On computability and tractability for infinite sets*

Mikołaj Bojańczyk and Szymon Toruńczyk
University of Warsaw
{bojan,szymtor}@mimuw.edu.pl

Abstract

We propose a definition for computable functions on hereditarily definable sets. Such sets are possibly infinite data structures that can be defined using a fixed underlying logical structure, such as $(\mathbb{N}, =)$. We show that, under suitable assumptions on the underlying structure, a programming language called *definable while programs* captures exactly the computable functions. Next, we introduce a complexity class called *fixed-dimension polynomial time*, which intuitively speaking describes polynomial computation on hereditarily definable sets. We show that this complexity class contains all functions computed by definable while programs with suitably defined resource bounds. Proving the converse inclusion would prove that Choiceless Polynomial Time with Counting captures order-invariant polynomial time on finite graphs.

1 Introduction

The goal of this paper is to identify the notion of computability, including "polynomial-time computability", for hereditarily definable sets. Such sets are a generalisation of hereditarily finite sets. They are possibly infinite, but can be defined using set builder notation in terms of some underlying logical structure $\mathbb{A}$, called the *atoms* of the hereditarily definable set. We begin with some examples. Suppose that the underlying structure of atoms is the natural numbers with equality $(\mathbb{N}, =)$. One possible hereditarily definable set consists of all unordered pairs of atoms:

$$\{\{x, y\} : \text{ for } x, y \in \mathbb{A} \text{ such that } x \neq y\}.$$

We can use parameters from the atoms, e.g. as in the following hereditarily definable set:

$$\{x : \text{ for } x \in \mathbb{A} \text{ such that } x \neq 5\}.$$

Another example is the set $\mathbb{A}^2$ of all ordered pairs, encoded via Kuratowski pairing:

$$\{\{x, \{x, y\}\} : \text{ for } x, y \in \mathbb{A} \text{ such that true}\}$$

If the atoms have more structure, then this structure can be used in the hereditarily definable sets, e.g. if the atoms are the ordered rational numbers $(\mathbb{Q}, \leq)$ then an example of a hereditarily definable set is the set of all closed intervals with right endpoint ≤ 7:

$$\{\{y : \text{ for } y \in \mathbb{A} \text{ such that } x \leq y \wedge y \leq z\} : \text{ for } x, y \in \mathbb{A} \text{ such that } x \leq y \wedge y \leq 7\}$$

As mentioned above, we can use Kuratowski pairing, and therefore pairs and tuples are allowed in hereditarily definable sets, which allows us to talk about structures such as graphs, e.g. the directed clique on all atoms $(\mathbb{A}, \mathbb{A}^2)$. A formal definition of hereditarily definable sets is given in Section 2. Hereditarily definable sets are a flexible and easy to use formalism for representing some possibly infinite data structures. The goal of this paper is to define what it means for an operation on hereditarily definable sets to be computable. A second goal, and the main original contribution of this paper, is to propose a definition of "polynomial time" computation.

Acknowledgements. The authors would like to thank Andreas Blass, Anuj Dawar and Erich Grädel for helpful discussions, as well as the Simons Institute for hosting the semester *Logical Structures in Computation*, where these discussions were held.

Background. This paper is part of the research programme on computation in *sets with atoms*, whose original motivation was the observation [4, 6] that various automata models over infinite alphabets can be viewed as "finite" automata under a suitable relaxation of finiteness (called orbit finiteness, which is essentially the same thing as hereditary definability) and that standard algorithms over finite objects (such as graph reachability, automaton emptiness, or automaton minimisation) extend transparently to the setting of hereditarily definable sets. An extended description of this topic can be found in the lecture notes [3].

We would like to underline that our main focus is on hereditarily definable sets over atoms such as $(\mathbb{N}, =)$ or maybe $(\mathbb{Q}, <)$, which are the central examples in the theory of sets with atoms. Sometimes, we can prove results with fewer assumptions, e.g. oligomorphism, or a decidable first-order theory. Nevertheless, the number of assumptions grows toward the end of the paper, and the final results are only given for $(\mathbb{N}, =)$.

Computability. The first contribution of this paper is a discussion of computability over hereditarily definable sets. This is not the first approach to this question. There are, in fact, already two programming languages that manipulate hereditarily definable sets: a functional programming language [5] and an imperative programming language [7, 15]. Furthemore, these programming languages have been implemented: the functional programming language as an extension of Haskell [14], and the imperative programming language as a C++ library [15]. In fact, [15] provides more than just a description of an implementation; it also shows how the programming language works for arbitrary logical structures with a decidable first-order theory, e.g. Presburger Arithmetic, and not just homogeneous ones as assumed in [7].

Our point of departure is the programming language from [7], extended to logical structures that are not necessarily homogeneous, which we call here *definable while programs*. In such a program, there is only one data type for the variables, namely hereditarily definable sets. There are the standard instructions of while programs like **if** and **while**, and there is a nonstandard **for** $x \in X$ instruction which executes a block of code in parallel ranging over possibly infinitely many elements x of a hereditarily definable set X. These instructions can be nested arbitrarily. Our first contribution is a simplified model, equivalent to definable while programs, which we call definable state machines, which operate by performing a sequence of simple operations.

*The work of M. Bojańczyk was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC consolidator grant LIPA, agreement no. 683080).

LICS 2018, July 9–12, 2018, Oxford, Great Britain
2018.

In [7, 15] it was shown – including an implementation – that definable while programs can be executed on a "normal computer", i.e., a Turing machine. The next question we ask in this paper is: are definable while programs computationally complete? Could one add features, in a way which would allow new computable functions, but so that the programs could still be executed on a normal computer? The second contribution of this paper is Theorem 3.8 which shows that definable while programs are computationally complete in the following sense: if a function on hereditarily definable sets can be computed on a normal computer and it is equivariant (i.e. invariant under automorphisms of the atoms), then it can be computed by a definable while program.

Polynomial-time computation. The last and principal contribution of this paper is a study of what it means to compute a function on hereditarily definable sets in polynomial time. One natural idea would be to have a polynomial-time algorithm, in the usual sense, which inputs an expression such as

$$\alpha = \{\{x\} : \text{ for } x \in \mathbb{A} \text{ such that true}\} \tag{1}$$

and then produces the output (either a new expression, in case of functions from hereditarily definable sets to hereditarily definable sets, or a yes/no value for Boolean questions). An important difficulty is that such a function should not depend on the representation of the input. For example, the set defined by α can alternatively be represented by the expression

$$\beta = \{\{x, y\} : \text{ for } x, y \in \mathbb{A} \text{ such that } x = y\}. \tag{2}$$

Since these are the same sets, then the outputs should be the same sets. Unfortunately, deciding if two expressions represent the same hereditarily definable set is a PSPACE-complete problem, which shows that polynomial-time algorithms manipulating such expressions have very limited capabilities, and would only allow modelling the most basic functions like the identity function or constant functions. Even when this problem with ambiguity is eliminated by requiring the inputs to be of a very restricted form, e.g. unnested sets of tuples, guarded by quantifier-free formulas, certain basic problems, such as reachability in graphs, remains PSPACE-hard. It would be disappointing to have a polynomial-time complexity class that would not contain graph reachability. To work around the PSPACE-hardness, we use parametrised complexity. We identify a parameter for hereditarily definable sets, which we call dimension. Roughly speaking, the dimension of an expression is the number of variables that it uses. For example, the expression α defined above in (1) has dimension 1, even though suboptimal expressions, such as β, might need more variables. Our proposal for polynomial time is that the running time is bounded by a function $f(d, n)$ where d is the dimension of the input representation and n is the size of the input representation (the representation might have larger than necessary dimension and size); subject to the restriction that for every fixed d the function $f(d, \_)$ is polynomial, although the degree of the polynomial is allowed to depend on d. For this complexity class (of functions on hereditarily definable sets) we introduce the name *fixed-dimension polynomial*. In Section 4 we describe this complexity class, and show that it is robust and captures natural problems like graph reachability, automaton minimisation or emptiness for context free languages. We also rule out alternative definitions, including one where the degree of the polynomial is fixed independently of d.

Connection to finite model theory. A special case of a hereditarily definable set is one which is hereditarily finite, possibly using the atoms. For example, if the atoms are $(\mathbb{N}, =)$, then the undirected clique on vertices $\{1, 2, 7\}$ is an example of a hereditarily definable set which is also hereditarily finite. If an algorithm inputs a representation of a set as an expression, then the representation will necessarily have some ordering on the vertices, e.g. $\{1, 2, 7\}$ and $\{2, 1, 7\}$ are two different representations of the same set. This leads us to the central question in finite model theory [11]: is there some logic which captures order-invariant polynomial time, i.e. exactly those properties of finite structures (e.g. graphs) that can be computed in polynomial time in a way that is invariant under possible representations. This question can be viewed as part of our setting in the following way. In Fact 1 we show that a class $\mathcal{L}$ of finite graphs is in order-invariant polynomial time (in the sense of finite model theory) if and only if membership of a hereditarily definable set in $\mathcal{L}$ is in our complexity class of fixed-dimension polynomial time. The reason is that all finite graphs can be represented by expressions of dimension zero; and over fixed-dimension there is no difference between the two complexity classes. The message is that the setting of finite model theory can be viewed as the dimension zero case of our setting.

Resource bounded definable while programs. The main technical contribution of this paper is a study of resource bounded definable while programs. We show that if our definable while programs are restricted to hereditarily finite sets and equipped with polynomial bounds on the memory and time used, then they have the same expressive power as Choiceless Polynomial Time (ČPT), an important logic that is contained in order-invariant polynomial time [2]. Adding counting to the while programs leads to equivalence with the counting version of ČPT. What is more, the definition of polynomial resource bounds can be extended to possibly infinite hereditarily definable sets, and we show that while programs with such polynomial bounds are contained in the complexity class of fixed-dimension polynomial time. We do not know if they capture the entire complexity class, and we dare not make any such conjectures. Proving such a capture result would prove that ČPT with counting captures order-invariant polynomial time on finite structures, thus solving a central open problem in finite model theory.

2 Basic definitions

Suppose that $\mathbb{A}$ is a logical structure, whose elements will be called *atoms*. Call $\mathbb{A}$ *effective* if its universe is a decidable subset of 2^* and there is an algorithm which inputs a first-order formula φ, and a valuation of its free variables in $\mathbb{A}$, and decides whether the valuation satisfies φ in $\mathbb{A}$. Call a structure *effectively presentable* if it is isomorphic to some effective structure. Examples of effectively presentable structures include $(\mathbb{N}, =)$, the rational numbers with order, $(\mathbb{Q}, \leq)$, the random graph, Presburger arithmetic $(\mathbb{N}, +)$ and Skolem arithmetic $(\mathbb{N}, \times)$.

Set builder expressions and hereditarily definable sets. Fix a logical structure $\mathbb{A}$ for the atoms. Fix some countably infinite set of variables, which are meant to range over atoms. Define *set builder expressions over* $\mathbb{A}$ as follows by structural induction:

Atom. Every atom $a \in \mathbb{A}$ is a set builder expression, called an *atom expression*.

Variable. Every variable is a set builder expression, called a *variable expression.*

Set expression. Let $x_1, \ldots, x_n, y_1, \ldots, y_m$ be distinct variables, which contain the free variables in a first-order formula φ and an already defined set builder expression α. The formula φ is over the vocabulary of $\mathbb{A}$ which may use parameters from the atoms. Then

$$\{\alpha(x_1, \ldots, x_n, y_1, \ldots, y_m) : \text{ for } y_1, \ldots, y_m \in \mathbb{A} \text{ such that } \varphi(x_1, \ldots, x_n, y_1, \ldots, y_m)\} \quad (3)$$

is a set builder expression, called a *set expression.* The free variables are $x_1, \ldots, x_n$ and the variables $y_1, \ldots, y_m$ are called bound. The formula φ is called the *guard* of the expression. A special case of a set expression is when there are zero bound variables, i.e. $m = 0$, in which case we write it as a singleton $\{\alpha(x_1, \ldots, x_n)\}$.

Union expression. If $\alpha_1, \ldots, \alpha_n$ are set expressions, then $\alpha_1 \cup \cdots \cup \alpha_n$ is a set builder expression. Such an expression is called a *union expression.*

For a set builder expression α with free variables $x_1, \ldots, x_n$, the semantics of α is a function which takes n arguments $a_1, \ldots, a_n$ and produces the corresponding set $\alpha(a_1, \ldots, a_n)$, defined in the natural way, which is either an atom, a set of atoms, a set of sets of atoms, etc. If α has no free variables, then this function takes no arguments, and we say that α *defines* the set $\alpha()$. Note that the same set can be defined by different set builder expressions.

Definition 2.1 (Hereditarily definable sets)**.** A *hereditarily definable set* over a logical structure $\mathbb{A}$ is any atom or set defined by a set builder expression without free variables. We write $\mathsf{hdef}\mathbb{A}$ for the hereditarily definable sets over $\mathbb{A}$, and $\mathsf{setb}\mathbb{A}$ for the set builder expressions over $\mathbb{A}$ without free variables.

An atom $a \in \mathbb{A}$ can appear in a set builder expression in two ways: either as a subexpression of type "atom", or as a parameter in a guard in some subexpression of type "set expression". In either case the atom is called a *parameter* of the expression. Recall that set expressions can be singletons, which allows us to create hereditarily finite sets (a set is called hereditarily finite if it is finite, its elements are finite, and so on), e.g. $\{\{5\} \cup \{6\}\} \cup \{5\}$ is a hereditarily definable set with zero free or bound variables and parameters $5, 6$. This set is the same as $\{\{5, 6\}, 5\}$, which is the same as the Kuratowski pair $(5, 6)$. In future examples we will use the more convenient expressions $(5, 6)$ or $\{\{5, 6\}, 5\}$, but these should be seen as syntactic sugar. Using this syntactic sugar, we can write directed graphs as hereditarily definable sets, e.g. $(\{1, 2, 3, 7\}, \{(1, 2), (2, 3), (3, 7)\})$ is a hereditarily definable set which describes a directed path of length 3. In this example, the parameters are $1, 2, 3, 7$ and there are no free or bound variables.

The guards in a set builder expression are allowed to use quantifiers. For example if the atoms are Presburger arithmetic $(\mathbb{N}, +)$ then

$$\{x : \text{ for } x \in \mathbb{A} \text{ such that } \exists y\ y + y = x \wedge y + y \neq y\}$$

defines the set of nonzero even numbers. The quantified variables are also counted as bound variables, e.g. in the above set builder expression both variables x and y are bound. Another example of a hereditarily definable set over Presburger arithmetic is the configuration graph of any vector addition system (VAS), or of a Minsky machine.

3 Computable functions on hereditarily definable sets

We define two notions of computability of functions on hereditarily definable sets: by means of Turing machines, and by means of a programming language called definable while programs.

3.1 Computable functions

A set builder expression can be written down so that it can be input and output by algorithms; assuming that there is some way to represent the parameters. In particular, if $\mathbb{A}$ is an effective structure, then we can represent set builder expressions as bit strings and it makes sense to talk about algorithms that input or output set builder expressions.

Definition 3.1 (Computatable function on hereditarily definable sets)**.** Let $\mathbb{A}$ be an effective structure. A function $F : \mathsf{hdef}\mathbb{A} \to \mathsf{hdef}\mathbb{A}$ is called *computable* if there is a function $F' : \mathsf{setb}\mathbb{A} \to \mathsf{setb}\mathbb{A}$ which is computable in the normal sense (i.e. a Turing machine) such that for every $\alpha \in \mathsf{setb}\mathbb{A}$ representing a hereditarily definable set x, $F'(\alpha)$ is a set builder expression which defines the hereditarily definable set $F(x)$.

The above definition talks about total functions; the extension to partial functions (where the Turing machine does not terminate on inputs with undefined values) is defined in the natural way. Note that the notion above depends on a particular presentation of an effectively presentable structure $\mathbb{A}$. In particular, given two effective structures $\mathbb{A}, \mathbb{A}'$ and an isomorphism α between them, the functions computable in $\mathbb{A}$ may not correspond to the functions computable in $\mathbb{A}'$ via the isomorphism α, if the isomorphism is not computable.

Since hereditarily definable sets are closed under taking tuples, one can talk about computable functions that go from tuples of hereditarily definable sets to hereditarily definable sets. For example, the functions $x \cap y$, $x - y$ and $\bigcup x$ are computable, as is easy to see, and also follows from Theorem 3.5 below. Natural numbers can be viewed as special cases of hereditarily definable sets, e.g. by using von Neumann numerals $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}$, etc. (those should not be confused with natural numbers which may appear in atoms, e.g. if the atoms are $(\mathbb{N}, =)$). Using such an encoding, we say that a subset $L \subseteq \mathsf{hdef}\mathbb{A}$ is computable if its characteristic function (which is total) into the booleans $\{0, 1\}$ is computable.

3.2 Definable while programs

A disadvantage of Definition 3.1 is that that it requires computing on representations (i.e. set builder expressions); in particular each algorithm needs to explicitly implement parsing of the inputs, and operations like computing $x \cap y$ or testing $x = y$ on the level of set builder expressions. To avoid this, we will use *definable while programs* as proposed in [7, 15]. The idea is to add a layer of abstraction on top of set builder expressions which allows the programmer to work directly with hereditarily definable sets. Before describing the programming language, consider two examples.

Example 3.2. The code below uses the atoms $(\mathbb{Q}, <)$. After executing it, the variable `X` will store the hereditarily definable set of all intervals of the form $(-\infty; a)$, for $a \in \mathbb{Q}$. This example illustrates the two key properties of the programming language: variables store hereditarily definable sets, and the **for** loop may range across an infinite set.

```
A := 𝔸;
X := ∅;
for a in A do
  I := ∅;
  for b in A do
    if b < a then
      I := I ∪ {b};
  X := X ∪ {I}
```

Example 3.3. For the code below, the choice of $\mathbb{A}$ is not important. The program inputs a graph stored in variables V and E as well as a set of source vertices S, and computes in variable R the vertices reachable from the sources. The program terminates if and only if there is some n such that every vertex is connected to S by a path of length n, or by no path at all.

```
R := S;
Old := ∅;
while R != Old do
  Old := R;
  for v in R do
    for w in V do
      if {v,w} ∈ E then
        R := R∪{w}
```

Syntax. We now present the syntax of definable while programs. We now present the syntax of definable while programs. Fix a logical structure $\mathbb{A}$ of atoms. We assume that there is a countably infinite set of names for program variables. Program variables are untyped, i.e. every program variable stores a hereditarily definable set. Although cumbersome, one can encode other data structures using hereditarily definable sets, e.g. the natural numbers can be encoded by von Neumann numerals. A reasonable implementation, such as [15], has more features, such as booleans or integer arithmetic. Below we describe the possible instructions in a minimalistic version of the language.

Expressions. We consider expressions of two types: *terms* and *conditions*. A term e may be a variable, a constant $\emptyset$ or $\mathbb{A}$, interpreted as the hereditarily definable set that contains all elements in the universe of $\mathbb{A}$. Terms can be built up using Boolean operations and singleton, $e_1 \cup e_2, e_1 \cap e_2, e_1 - e_2, \{e_1\}$, with the expected semantics. Additionally, for each function symbol f of arity n in the signature of $\mathbb{A}$, there is a term $f(e_1, \ldots, e_n)$, which has the following semantics: if at least one e_i does not represent an atom, then $f(e_1, \ldots, e_n)$ evaluates to $\emptyset$; otherwise, if all expressions $e_1, \ldots, e_n$ evaluate to atoms, then $f(e_1, \ldots, e_n)$ evaluates to the value of f on the corresponding atom tuple.
A condition is a boolean combination using $\wedge, \vee, \neg$ of statements of the form $e_1 = e_2, e_1 \in e_2$, or $R(e_1, \ldots, e_n)$, where e_1, e_2 are terms and R is a relation symbol in the vocabulary of $\mathbb{A}$ of arity n, and $R(x_1, \ldots, x_n)$ denotes that the tuple $(x_1, \ldots, x_n)$ belongs to the interpretation of the symbol R in $\mathbb{A}$. (We adopt the convention that R is false when at least one of its arguments is not an atom.)
Assignment. If x is a program variable and e a term, then $x := e$ is an instruction, which loads the value of the expression e into the variable x.
Sequential composition. If I and J are already defined programs, then also I ; J is a program which first executes I and then J.

Control flow. Suppose that c is a condition, I and J are already defined programs. Then the following are programs:

if c **then** I **else** J **while** e **do** I

The for loop. The nonstandard construct in the programming language is the following **for** loop. Suppose that x is a variable, e a term, and I is an already defined program. Then the following is also a program:

for x **in** e **do** I

The idea behind this program is that it executes I in parallel for all elements of the set represented by e, with the results of the parallel executions being aggregated using set union.

We remark that our list of operations allowed in the expressions is redundant – a smaller, equivalent set of operations would allow only $\mathbb{A}$, $x \cup y$, $f(x_1, \ldots, x_n)$ and $\{x\}$ as terms and $x = y$ and $R(x_1, \ldots, x_n)$ as conditionals, where $x, y, x_1, \ldots, x_n$ are variables, and not expressions. However, we allow a more verbose syntax for convenience.

Semantics. We now present the formal semantics of definable while programs. A *program state* is a function which assigns hereditarily definable sets to the program variables appearing in the program. If γ is a program state and e is a term or a condition, then the semantics $[\![e]\!]_\gamma$ is defined in the natural way, by evaluating the expression e with values $\gamma(x)$ substituted for the variables x. Intuitively, the **for** loop splits a single program state into many parallel threads. This can be formalized by introducing *superstates*, which keep track of many threads simultaneously. A *superstate* S is an indexed family $(S_\tau)_{\tau \in T}$ of states; the elements of the indexing set T are called the *threads* of S. Intuitively speaking, the index τ is going to be a stack of hereditarily definable sets, corresponding to the values that are bound in successive nestings of the **for** loops.

The operational semantics of definable while programs is given by the rules listed in Figure 1 on page 5. The relation $S \rightarrowtail [\![\text{ I }]\!] \rightarrow S'$ denotes that performing the instruction I in superstate S results in superstate S'. This is a partial function from the first two arguments S and I to the third argument S'; it is partial because **while** loops might not terminate. The functions Split and Aggregate used in Figure 1 are explained below.

The intuition behind the operation $\text{Split}(S, x, e)$ is that it describes what will happen if all threads in a superstate S execute an a loop of the form **for** $x \in e$. Let S be a superstate, let x be a program variable and let e be an expression. Let $\text{Split}(S, x, e)$ be the superstate T defined as follows. The threads of T are pairs (τ, v) where τ is a thread of the superstate S and v is an element of the set represented by expression e in the program state corresponding to thread τ, i.e. $v \in [\![e]\!]_{S_\tau}$. The program state corresponding to thread (τ, v) in the superstate T is the following map from program variables to hereditarily definable sets:

$$y \quad \mapsto \quad \begin{cases} S_\tau(y) & \text{if } y \neq x \\ v & \text{otherwise} \end{cases}$$

The operation $\text{Aggregate}(S')$ performs an inverse operation to split; intuitively speaking it says what happens after finishing the execution of a **for** loop. The operation is only defined if S' is a superstate where every thread is of the form (τ, v), for some τ and v. The result of the operation $\text{Aggregate}(S')$ is a new superstate S defined as follows. The threads of S are threads τ such that (τ, v) is a thread of S' for some v. The value of a variable x in the program

$$\frac{}{\emptyset \rightarrowtail [\![\, \mathtt{I} \,]\!] \rightarrowtail \emptyset} \quad (\textit{no-threads}) \qquad \frac{S \rightarrowtail [\![\, \mathtt{I}_1 \,]\!] \rightarrowtail S' \qquad S' \rightarrowtail [\![\, \mathtt{I}_2 \,]\!] \rightarrowtail S''}{S \rightarrowtail [\![\, \mathtt{I}_1 ; \mathtt{I}_2 \,]\!] \rightarrowtail S''} \quad (\textit{sequencing}) \qquad \frac{}{S \rightarrowtail [\![\, \textbf{skip} \,]\!] \rightarrowtail S} \quad (\textit{skip})$$

$$\frac{}{S \rightarrowtail [\![\, \mathtt{x} := e \,]\!] \rightarrowtail S[x/e]} \quad (\textit{assignment}) \qquad \frac{S[c] \rightarrowtail [\![\, \mathtt{I} \,]\!] \rightarrowtail S_+ \qquad S[\neg c] \rightarrowtail [\![\, \mathtt{J} \,]\!] \rightarrowtail S_-}{S \rightarrowtail [\![\, \textbf{if } \mathtt{c} \textbf{ then } \mathtt{I} \textbf{ else } \mathtt{J} \,]\!] \rightarrowtail S_+ \cup S_-} \quad (\textit{if-then-else})$$

$$\frac{S[c] \rightarrowtail [\![\, \mathtt{I} \,]\!] \rightarrowtail S' \qquad S' \rightarrowtail [\![\, \textbf{while } \mathtt{c} \textbf{ do } \mathtt{I} \,]\!] \rightarrowtail S''}{S \rightarrowtail [\![\, \textbf{while } \mathtt{c} \textbf{ do } \mathtt{I} \,]\!] \rightarrowtail S'' \cup S[\neg c]} \quad (\textit{while}) \qquad \frac{\mathrm{Split}(S, x, e) \rightarrowtail [\![\, \mathtt{I} \,]\!] \rightarrowtail S'}{S \rightarrowtail [\![\, \textbf{for } \mathtt{x} \textbf{ in } e \textbf{ do } \mathtt{I} \,]\!] \rightarrowtail \mathrm{Aggregate}(S')} \quad (\textit{for})$$

Figure 1. Structural operational semantics of definable while programs. The notation used in the specific rules above is defined below. **(*no-threads*)**: $\emptyset$ denotes the superstate with empty set of threads. **(*assignment*)**: if e is a term and x is a variable, then by $S[x/e]$ we denote the superstate S' such that for every thread τ of S, $S'_\tau(x) = [\![e]\!]_{S_\tau}$ and $S'_\tau(y) = S_\tau(y)$ for $y \neq x$. **(*if-then-else*)** and **(*while*)**: if S is a superstate and c is a condition, then by $S[c]$ we denote the superstate obtained from S by restricting to those threads τ for which $[\![c]\!]_{S_\tau}$ evaluates to true.

state corresponding to thread τ in S is defined as follows. Consider the possible values of variable x in threads of S' that begin with τ, i.e.

$$\{S'_{(\tau,v)}(x) : v \text{ is such that } (\tau, v) \text{ is a thread of } S'\}. \tag{4}$$

If the set above contains one element, i.e. all threads beginning with τ agree on variable x, and this element is furthermore an atom a, then we define the value of variable x in thread τ of T to be a. Otherwise (i.e. either some thread beginning with τ stores a non-atom in variable x, or threads beginning with τ disagree on their contents) then the value of variable x in thread τ of T is defined to be the union of the set in (4), i.e. the set of elements that belong to at least one set from (4).

Example 3.4. Suppose that S' is a superstate where the threads are all pairs of atoms (a, b). Let S be the superstate $\mathrm{Aggregate}(S')$. The threads of S are individual atoms a.

- Assume that for program variable x, the program state indexed by (a, b) in S' stores the set $\{b\}$. Then the program state indexed by a in S stores the following set in variable x:
$$\mathbb{A} = \bigcup_{b \in \mathbb{A}} \{b\}$$
- Assume that for program variable y, the program state indexed by (a, b) in S' stores the atom b. Then the program state indexed by a in S stores the following set in variable y:
$$\emptyset = \bigcup_{b \in \mathbb{A}} b.$$
This set is empty because an atom has no elements.
- Assume that for program variable y, the program state indexed by (a, b) in S' stores the atom a. Then the program state indexed by a in S also stores a in variable z, because all threads beginning with a have the same value in variable z.

3.3 Functions computed by definable while programs

A definable while program $\mathtt{P}$ is an instruction $\mathtt{I}$ with a distinguished tuple of *input* variables $x_1, \ldots, x_n$ and a distinguished tuple of *output* variables $y_1, \ldots, y_m$. Such a program defines a partial function which maps n-tuples of sets to m-tuples of sets, as expected. Formally, for a given tuple $u_1, \ldots, u_m$ of sets, let $S^{\bar{u}}$ be the superstate with one thread denoted ε, such that $S^{\bar{u}}_\varepsilon$ is the program state which assigns u_i to the variable x_i, and $\emptyset$ to all remaining variables. If $S^{\bar{u}} \rightarrowtail [\![\, \mathtt{P} \,]\!] \rightarrowtail S$, then S also has only one thread ε, and we say that the *result* of the definable while program $\mathtt{P}$ on input $u_1, \ldots, u_n$ is the tuple of values $v_1, \ldots, v_m$, where $v_i = S_\varepsilon(y_i)$. We also say that the program $\mathtt{P}$ *computes* the partial function mapping a tuple $u_1, \ldots, u_n$ to the result $v_1, \ldots, v_m$. We will usually restrict to the case $n = m = 1$ for simplicity.

Note that according to the above definition, it makes sense to run definable while programs on any input sets, not necessarily hereditarily definable ones. It is not difficult to show that if the input sets are hereditarily definable, then the result (if defined) is a tuple of sets which are again hereditarily definable. Therefore, a definable while program with one input variable and one output variable induces a partial function $f : \mathrm{hdef}\mathbb{A} \to \mathrm{hdef}\mathbb{A}$. The following result shows that definable while programs compute functions which are computable in the sense of Definition 3.1.

Theorem 3.5. *Assume that $\mathbb{A}$ is effective. Then every partial function $f : \mathrm{hdef}\mathbb{A} \to \mathrm{hdef}\mathbb{A}$ which is computed by a definable while program over $\mathbb{A}$ is computable.*

Theorem 3.5 was shown in [7] under a stronger assumption that $\mathbb{A}$ is *homogeneous* and effective, and for arbitrary effective atoms in [15], although for a slightly different semantics of while programs. We will show a partial converse to the above theorem in Theorem 3.8.

3.4 Resource consumption

One of the principal goals of this paper is to define polynomial time computation for hereditarily definable sets, and therefore we need some way of bounding the resources of while programs. In this section, we begin by defining the resource consumption for a definable while programs as a hereditarily definable set. Later in Section 4 we discuss how to measure the resource consumption numerically, in the special case when $\mathbb{A}$ is $(\mathbb{N}, =)$.

Let $\mathbb{A}$ be an arbitrary logical structure. Suppose that P is a while program, which uses program variables $x_1, \ldots, x_n$ and no others. Define a new program P' as follows. It has the same program variables as P, plus two fresh program variables (initially storing empty sets) called time and space. The code of P' is the same as P, except that after each instruction we append the following code:

$$\texttt{time} := \texttt{time} \cup \{\texttt{time}\};$$
$$\texttt{space} := \texttt{space} \cup \{x_1, \ldots, x_n\}$$

The idea is that the time variable stores an instruction counter represented as a von Neumann integer; and this counter is incremented after each instruction. The space variable stores all sets that ever appeared during the computation. The input variables of P' are the same as of P, whereas the output variables are the variables time and space. For a while program P with n input variables and an n-tuple $\bar{x}$ of hereditarily definable sets $x_1, \ldots, x_n \in \mathsf{hdef}\mathbb{A}$, define the *time consumption* and the *space consumption* of P over $\bar{x}$ to be the pair of values produced by the program P' on input $x_1, \ldots, x_n$. The *resource consumption* of P over $\bar{x}$ is the union of the time consumption and the space consumption. We denote the time, space and resource consumption by $\mathrm{time}(\mathsf{P}, \bar{x})$, $\mathrm{space}(\mathsf{P}, \bar{x})$, and $\mathrm{resource}(\mathsf{P}, \bar{x})$, respectively. These values are undefined if the program does not terminate on $x_1, \ldots, x_n$. Note that the time consumption is a von Neumann encoding of a natural number, but space consumption has no immediate numerical meaning so far.

Constant time operations. We distinguish a special class of functions which can be defined by a while program without **while** loops. Say that a function f which maps n-tuples of hereditarily definable sets to m-tuples of hereditarily definable sets is a *constant time operation* if there is a definable while program P which does not use **while** loops and defines f, i.e., $f(u_1, u_2, \ldots, u_n) = (v_1, \ldots, v_m)$ if and only if P outputs $v_1, \ldots, v_m$ given input $u_1, \ldots, u_n$. Note that the time consumption of a constant time operation is bounded by a constant, as the name suggests.

Example 3.6. The following functions are constant time operations:

- The function pair which maps a pair of inputs x, y to the Kuratowski encoding of (x, y), that is $\{x, \{x, y\}\}$.
- The reverse operation, unpair which returns a pair x, y if the argument is the Kuratowski encoding of (x, y), and the empty set otherwise,
- The Cartesian product $x, y \mapsto x \times y$, as well as boolean operations $x, y \mapsto x \cup y$, $x, y \mapsto x \cap y$, $x, y \mapsto x - y$, and $x \mapsto \bigcup x$.

3.5 Definable state machines

We introduce a sequential model of computation equivalent to definable while programs, but more in the spirit of Turing machines and similar to abstract state machines introduced by Gurevich (see Section 4.2 for a discussion). A *definable state machine* M consists of four constant time operations Input, Output, Step, and Halt, where each takes one input and one output. For a hereditarily definable set x given on input, define the nth *state* q_n of the *run* of M on input x inductively: $q_0 = \mathrm{Input}(x)$ and, for $n \geq 0$, if q_n is defined and $\mathrm{Halt}(q_n) = \emptyset$, then $q_{n+1} = \mathrm{Step}(q_n)$. The machine *halts* on input x if the run is finite, in which case we define the *output* as $\mathrm{Output}(q_n)$, where q_n is the last state of the run.

Definable state machines can be seen as a special case of definable while programs with one input variable, one output variable and with only one while loop, of the form

$$I;\ \textbf{while}\ (x \neq \emptyset)\ \textbf{do}\ J;\ K,$$

where I, J, K are constant time operations. Conversely, we show that definable while programs can be simulated by definable state machines, preserving the used resources.

Theorem 3.7. *Fix a logical structure $\mathbb{A}$. For every definable while program P there is a definable state machine M such that for every hereditarily definable set x, M halts on x if and only if P halts on x. Moreover, if M halts on x, then the following properties hold:*

- *The output of M on x is equal to the output of P on x,*
- *The number of steps performed by M on input x is polynomial in $\mathrm{time}(\mathsf{P}, x)$,*
- *Each state is a subset of $L(\mathrm{space}(\mathsf{P}, x))$, where L is a constant time operation depending only on P.*

3.6 While programs are computationally complete

In this section, we restrict our attention to *oligomorphic* atoms, which we now define. An *automorphism of* $\mathbb{A}$ is defined to be any bijection of its universe with itself which preserves and reflects all relations and preserves the functions; these automorphisms form a group. If this group acts on a set X, then we say that two elements $x, y \in X$ are in the same orbit of the action if there is an atom automorphism π such that $\pi \cdot x = y$. This defines an equivalence relation on X, whose equivalence classes are called orbits of X. The orbit containing an element $x \in X$ is called the orbit of x in X.

For every number n, the automorphisms of $\mathbb{A}$ act on $\mathbb{A}^n$ componentwise. We say that $\mathbb{A}$ is *oligomorphic* if for every dimension n, there are finitely many orbits of $\mathbb{A}^n$ under this action. A theorem independently proved by Engeler, Ryll-Nardzewski and Svenonius (cf. Theorem 7.3.1 in [13]) says that for a countable structure, being oligomorphic is equivalent to having an *ω-categorical* theory (a theory is ω-categorical if any two countable models of the theory are isomorphic). Moreover, if $\mathbb{A}$ is oligomorphic, then every orbit of $\mathbb{A}^n$ is defined by a first-order formula with n free variables. For example, $(\mathbb{N}, =)$ and $(\mathbb{Q}, \leq)$ are oligomorphic, and $(\mathbb{N}, \leq)$ is not.

If x is a hereditarily definable set defined by a set builder expression α and π is an atom automorphism, then π can be applied to the atoms in x, to the atoms in the elements of x, etc. recursively, yielding another hereditarily definable set $\pi \cdot x$, which can be defined by the set builder expression α in which the parameters are mapped via π. Therefore, the group of atom automorphisms acts on hereditarily definable sets. We say that a finite set of atoms S *supports* a hereditarily definable set x if $\pi \cdot x = x$ holds for every atom permutation which fixes S pointwise. In particular, the parameters appearing in a set builder expression α support the hereditarily definable set defined by α.

A (possibly partial) function f from hereditarily definable sets to hereditarily definable sets is called *equivariant* if $\pi \cdot f(x) = f(\pi \cdot x)$ holds for every hereditarily definable x and atom automorphism π. The semantics of definable while programs is invariant under atom automorphisms, and hence we see that every f computed by such a program is equivariant. Theorem 3.8 below shows that f will also be computable in the sense of Definition 3.1, and furthermore, under suitable assumptions on the atoms, all equivariant computable functions are of this form.

Theorem 3.8 (Computational completeness of definable while programs)**.** *Assume that $\mathbb{A}$ is effective, oligomorphic and that, given $n \in \mathbb{N}$, one can compute a first-order formula with $2n$ free variables which defines the "same orbit" relation on $\mathbb{A}^n \times \mathbb{A}^n$. Then the following conditions are equivalent for every partial function $f : \mathsf{hdef}\mathbb{A} \to \mathsf{hdef}\mathbb{A}$.*

1. *f is computed by a while program over $\mathbb{A}$.*
2. *f is equivariant and computable.*

Recall that Theorem 3.5 shows the implication 1→2 when $\mathbb{A}$ is only assumed to be effective. The original contribution is the implication 2→1. Roughly speaking, the assumptions of the theorem (being oligomorphic and computing formulas for the "same orbit" relation) are used to give a while program which inputs a hereditarily definable set x and reverse engineers it to obtain a binary string describing a set builder expression for x. Nevertheless, we do not know if we really need the assumptions. The proof of the theorem is in the appendix.

4 Fixed-dimension polynomial functions on hereditarily definable sets

In the previous section we discussed computable functions on hereditarily definable sets, without bounding the resources used by the computation. We now turn to the main contribution of this paper: a proposal for "polynomial time" computation.

The first idea that comes to mind is to consider to take Definition 3.1 and simply add the requirement that F' is computable in polynomial time. This is not a good idea, as long as the atom structure is nontrivial. The reason is that when $\mathbb{A}$ has at least two elements, then even emptiness is hard: it is PSPACE-hard to check if a given set builder expression describes the empty set. This lower bound follows from a straightforward reduction from QBF. For this reason, all but the most trivial transformations on hereditarily definable sets are going to be PSPACE-hard if we measure running time in the traditional way. Our approach to this problem is to use the setting of parametrised complexity, where the running time of the algorithm is measured only when the value of a certain parameter is fixed. The parameter used is the dimension of a set builder expression, as defined below.

Definition 4.1 (Dimension and size of set builder expressions)**.** Define the *dimension* $\dim \alpha$ of a set builder expression α to be the number of distinct variables that it uses. This includes both the variables used in set expressions, as well as the quantified variables used in guards. Define the size $\|\alpha\|$ of a set builder expression α to be the number of distinct subexpressions in it plus the number of distinct subformulas of the formulas used in the guards.

In the above definition it is important that we count distinct variables, i.e., if the same variable is reused by binding it several times, then it only gets counted once. It is also important that dimension does not count parameters. In the appendix, we explain why parameters are not counted; the idea is that counting parameters would break the connections to finite model theory as stated in Fact 1 and Theorem 2. Note also how in the definition of $\|\alpha\|$ we count the number of distinct subexpressions and subformulas, as opposed to simply counting the number of symbols needed to write the expression down. The latter method can yield exponentially larger sizes, as witnessed by von Neumann numerals. By analogy, our method of counting the size is similar to circuit size as opposed to formula size.

Example 4.2. All of these examples are for $\mathbb{A}$ being the ordered rational numbers. The set builder expression

$$\{1, 2, \{2, 4\}, \{1, 2, \{5\}\}\}$$

has dimension zero, because it uses no variables. The following set builder expression has dimension 2, because it uses variables x, y, even though variable x gets bound a second time:

$$\{\{x\} \cup \{x : \text{ for } x \in \mathbb{A} \text{ such that } x \neq y\} : \text{ for } x, y \in \mathbb{A} \text{ such that } x \neq y \wedge x \neq 5\}.$$

The following expression has dimension 4 because of the variables used in the guards:

$$\{x : \text{ for } x \in \mathbb{A} \text{ such that } \exists y\ \exists z\ \exists u\ 5 < y < z < u < x\}.$$

In the example above, the guard could be replaced by the quantifier-free formula $5 < x$, reducing the dimension to 1.

Definition 4.3 (Fixed-dimension polynomial algorithm)**.** Let $\mathbb{A}$ be an effective structure. An algorithm which inputs and outputs set builder expressions is called *fixed-dimension polynomial* if there exist functions

$$f : \mathbb{N}^2 \to \mathbb{N} \qquad g : \mathbb{N} \to \mathbb{N}$$

with the following properties:

1. the function f is polynomial once the first argument is fixed.
2. if the input is $\alpha \in \mathsf{setb}\mathbb{A}$ then:
   - the running time of the algorithm is at most $f(\dim \alpha, |\alpha|)$;
   - the dimension of output expression is at most $g(\dim \alpha)$.

A total function $F : \mathsf{hdef}\mathbb{A} \to \mathsf{hdef}\mathbb{A}$ is called *fixed-dimension polynomial* if there is a fixed-dimension polynomial algorithm which inputs a set builder expression α and outputs a set builder expression representing the value of F on the set defined by α.

A typical example of f would be $(k, n) \mapsto n^k$. It is not hard to see that fixed-dimension polynomial functions are closed under compositions.

An algorithm which always returns 0 or 1 (encoded as $\emptyset$ and $\{\emptyset\}$) can be seen as a language recognizer. Note that a language of set builder expressions is recognized by a fixed-dimension polynomial algorithm if it belongs to the class XP from parametrised complexity, with the parameter being dimension. An alternative solution would be to use *fixed-dimension tractability*, i.e. algorithms with running time at most $f(\dim \alpha) \cdot |\alpha|^c$ for some computable $f : \mathbb{N} \to \mathbb{N}$ and some $c \in \mathbb{N}$. The following lemma shows that the alternate solution is a bad idea. For the definition of the W hierarchy and background on parametrised complexity, see [8].

Lemma 4.4. *If the* W *hierarchy does not collapse and $\mathbb{A}$ is infinite, then no fixed-dimension tractable algorithm can decide if a set builder expression defines the empty set.*

From now on we do not consider fixed-parameter tractable algorithms, and study only fixed-dimension polynomial ones.

Example 4.5. Assume that the atoms are $(\mathbb{N}, =)$. Then the following operations on pairs of hereditarily definable sets x, y are fixed-dimension polynomial: testing $x = y$, $x \in y$ and $x \subseteq y$, as well as computing $x \cap y, x \cup y, x - y$. These are special cases of Lemma 4.10 below, which states that every constant time operation is fixed-dimension polynomial. It is very important that we use the atoms $(\mathbb{N}, =)$. For some oligomorphic structures such as the random graph, set emptiness is NP-hard even for dimension 1 inputs (the proof is in the appendix).

When the atoms are $(\mathbb{N}, =)$, the following problems are also fixed-dimension polynomial for inputs consisting of hereditarily definable objects: graph reachability, automata emptiness, context-free grammar emptiness, automata minimisation. This follows from 4.11 below, as all these problems can be implemented by the usual fixpoint algorithms. Note that all these problems become undecidable when the atoms are Presburger arithmetic, or even $\mathbb{N}$ with the successor relation.

Connection to finite model theory. The central question in finite model theory is understanding which properties of structures (typically, graphs are considered without loss of generality) can be decided in polynomial time. More precisely, a class of graphs is said to be in *order-invariant polynomial time* if there is a polynomial-time algorithm (say, Turing machine), which decides membership given an incidence matrix of the graph, such that the algorithm gives the same answer for incidence matrices describing isomorphic graphs. The following observation relates order-invariant polynomial-time to our setting.

Fact 1. *Assume that the atoms $\mathbb{A}$ are $(\mathbb{N}, =)$. A class L of finite graphs is in order-invariant polynomial-time if and only if there is a fixed-dimension polynomial (equivalently, fixed-dimension tractable) algorithm deciding membership in $\{G \in L : \text{all vertices are from } \mathbb{A}\}$*

The reason for the above fact is that, when all vertices are from $\mathbb{A}$, then a graph has dimension zero; and for such inputs fixed-dimension polynomial (and tractable) collapses to polynomial.

4.1 Tractable while programs

In the previous section, we defined what it meant for a function $\mathsf{hdef}\mathbb{A} \to \mathsf{hdef}\mathbb{A}$ to be computable in fixed-dimension polynomial time. In this section we define a resource bounded version of while programs which can only compute fixed-dimension polynomial time functions. Such programs are easier to write because they directly talk about hereditarily definable sets and not their representations.

The results we present in this section assume that the atoms $\mathbb{A}$ are $(\mathbb{N}, =)$. One important property of these particular atoms is the existence of least supports, in the following sense: for every $x \in \mathsf{hdef}\mathbb{A}$ there is a finite set of atoms, called its *least support*, which is contained in every support of x. Existence of least supports for $(\mathbb{N}, =)$ is shown in [16].

Dimension and size of hereditarily definable sets. In Section 3.4 we have defined the resource consumption of a definable while program, which is a hereditarily definable set. When defining resource bounded while programs, we will want to say that, on input $x \in \mathsf{hdef}\mathbb{A}$, the resource consumption of a program is bounded by a polynomial in the size of x, whose degree is allowed to depend on the dimension of x. For this to make sense, we need to be able to talk about the dimension of x, as well as the size of x and its memory consumption, seen as natural numbers. In other words, we need notions of dimension and size for hereditarily definable sets themselves, and not for the set builder expressions defining them (as we have done before). One approach would be to use the dimension and size of expressions that are optimal in some sense. The following definition proposes a different approach; albeit one that strongly depends on the fact that the atoms $(\mathbb{N}, =)$ admit least supports.

Definition 4.6 (Dimension and size of hereditarily definable sets)**.** Let $\mathbb{A}$ be $(\mathbb{N}, =)$ and let $x \in \mathsf{hdef}\mathbb{A}$. Let x_* be the transitive closure of x, i.e. the set which contains all elements of x, all elements of all elements of x, and so on recursively. Define the *dimension* of x to be

$$\dim x \stackrel{\text{def}}{=} \max_{y \in x_*} |\sup(y) - \sup(x)|,$$

where $\sup(\cdot)$ denotes the least support. Define the *orbit size* of x, denoted by $\|x\|$, to be the number of elements $y \in x_*$ with respect to the group of those atom automorphisms which are the identity on the least support of x.

The following lemma is the key technical result used in the proof of our main result, Theorem 4.9 below. It shows that the size and dimension of a hereditarily definable set, as given above, is approximately the same as the optimal size and dimension of a set builder expression that defines it. Furthermore, the optimal expression can be computed in fixed-dimension polynomial time. Therefore, up to fixed-dimension polynomial corrections, there is a robust notion of "size" for hereditarily definable sets; in particular the alternative approach discussed before Definition 4.6 would be essentially equivalent.

Lemma 4.7. *There exists a function $f : \mathbb{N}^2 \to \mathbb{N}$ which is polynomial when the first coordinate is fixed with the following properties.*

1. *For every $x \in \mathsf{hdef}\mathbb{A}$ and $\alpha \in \mathsf{setb}\mathbb{A}$ defining x, $\dim x \leq \dim \alpha$ and $\|x\| \leq f(\dim \alpha, \|\alpha\|)$.*
2. *For every $x \in \mathsf{hdef}\mathbb{A}$ there exists some $\alpha \in \mathsf{setb}\mathbb{A}$ which defines it such that $\dim \alpha \leq 2 \dim x$ and $\|\alpha\| \leq f(\dim x, \|x\|)$. Furthermore, α can be computed in fixed-dimension polynomial time based on a set builder expression representing x.*

Resource bounded while programs. Having defined the resource consumption of a while program, as a hereditarily definable set, and knowing how to measure the size and dimension of a hereditarily definable set, we can introduce our proposal for resource bounded while programs.

Definition 4.8. Assume that $\mathbb{A}$ is $(\mathbb{N}, =)$. We say that a while program P with a single input variable is *fixed-dimension polynomial* if there is a function $f : \mathbb{N}^2 \to \mathbb{N}$, which is polynomial once the first argument is fixed, and a computable function $g : \mathbb{N} \to \mathbb{N}$, such that

$$\dim(\text{resource}(\mathsf{P}, x)) \leq g(\dim x) \quad \text{and}$$
$$\|\text{resource}(\mathsf{P}, x)\| \leq f(\dim x, \|x\|) \quad \text{for } x \in \mathsf{hdef}\mathbb{A}.$$

We can extend while programs with a counting expression, which is a term of the form $|x|$, whose semantics is the von Neumann numeral representing the size of x when x is a finite set, and the empty set if x is an infinite set. This operation can be simulated by a definable while program, but it would take exponential resources to do it, even for x of dimension zero. If such an operation is allowed, then we talk about *definable while programs with counting*. Theorem 3.7 remains valid for while programs with counting, where the definable state machines can use counting expressions in the operations Input, Output, Step, Halt. Definition 4.8 is easily extended to programs with counting.

The following theorem is our main result. Its proof is given in the appendix.

Theorem 4.9. *Assume that the atoms are* $(\mathbb{N}, =)$. *For every while program with counting which is fixed-dimension polynomial (in the sense of Definition 4.8), the function it computes is fixed-dimension polynomial (in the sense of Definition 4.3).*

We make no conjectures about the converse implication, when it comes to decision problems (for general computational problem, a negative result follows from Rossmann's result, see below). Theorem 4.9 follows rather easily from Theorem 3.7, Lemma 4.7, and Lemma 4.10 below. The proofs are in the appendix.

Lemma 4.10. *Assume that the atoms are* $(\mathbb{N}, =)$. *Every constant time operation is fixed-dimension polynomial.*

Fixpoint operations. As an example of a class of fixed-dimension polynomial computable functions we consider *bounded least fixpoint operations*, defined below.

Let Input, Bound, Step, Output be constant time operations. Define the function f which, given input x, proceeds in steps as follows. Let $q_0 = \text{Input}(x)$, and inductively define q_{n+1} as $q_n \cup \text{Step}(q_n)$. If $q_n \not\subseteq \text{Bound}(x)$ for some n, then $f(x)$ is undefined. Otherwise, $q_0 \subseteq q_1 \subseteq q_2 \subseteq \ldots \subseteq \text{Bound}(x)$. As the atoms are oligomorphic, it is easy to see that the sequence $q_0, q_1, \ldots$ must stabilize, i.e., there is an n such that $q_n = q_{n+1}$. Define $f(x)$ as $\text{Output}(q_n)$. This finishes the definition of the bounded fixpoint operation f defined by Input, Bound, Step and Output.

As an example, the program in Example 3.3 implementing graph reachability computes a function which is bounded fixpoint operation. Other examples include emptiness of context-free grammars and the reachability problem for tree automata. Clearly, every bounded fixpoint operation f is computable by a definable while program.

Lemma 4.11. *Assume that the atoms are* $(\mathbb{N}, =)$. *Every bounded least fixpoint operation is fixed-dimension polynomial.*

Proof. Let f be a least fixpoint operation given by the constant time operations Input, Bound, Step, Output. Let P be the natural implementation of f as a definable while program, obtained from implementations of the four operations.

We need to bound the time and space consumption of P on a given input x. To bound the time, it is sufficient to bound the number n for which stabilization occurs, i.e., $q_n = q_{n+1}$, since the output $f(x)$ is computed by a composition of $n+2$ constant time operations. Let S_0 be the set of atoms which occur as parameters in the definitions of the operations Input, Bound, Step, Output. It is easy to show by induction that if S is the least support of x, then for each i, the set q_i is again supported by $S \cup S_0$. Also, the set $\text{Bound}(x)$ is supported by $S \cup S_0$. As $q_i \subseteq \text{Bound}(x)$, it follows that q_i is a union of orbits of $\text{Bound}(x)$ under the action of the group

$$G = \{\pi : \pi \text{ is a permutation of } \mathbb{A} \text{ fixes all atoms from } S \cup S_0\}$$

As the sets q_i form an increasing chain, it follows that the moment of stabilization n is bounded by the number of orbits of $\text{Bound}(x)$ under the action of G. We will show that this number of orbits is fixed dimension polynomial.

Consider the operation g which maps an input x to the triple $(x, \text{Bound}(x), S_0)$. This is clearly a constant time operation, computable by a definable while program B obtained from the while program defining Bound. By Lemma 4.11, the operation g is fixed-dimension polynomial, so $\|\text{resource}(\mathsf{B}, x)\| \leq p(\dim x, \|x\|)$ for some function $p : \mathbb{N}^2 \to \mathbb{N}$ which is polynomial whenever the first coordinate is fixed. As $g(x) \subseteq \text{resource}(\mathsf{B}, x)$ and $S \cup S_0$ is the least support of $g(x)$, it follows that the number of orbits of $g(x)$ under the action of G is equal to $\|g(x)\|$, which is bounded by $\|\text{resource}(\mathsf{B}, x)\| \leq p(\dim x, \|x\|)$. Therefore, the number of steps performed by the least fixpoint computation of $f(x)$ is bounded by $p(\dim x, \|x\|)$. As each step is computed by a constant time operation, it follows from Lemma 4.11 that the running time of P on x is bounded by $p'(\dim x, \|x\|)$, for some function p' which is polynomial in the second component.

As for the space consumption, from the above discussions it follows that $\text{space}(\mathsf{P}, x) \subseteq \text{space}(\mathsf{B}, x)$ and the least support of $\text{space}(\mathsf{P}, x)$ is contained in the least support of $\text{space}(\mathsf{B}, x)$. In particular, $\|\text{space}(\mathsf{P}, x)\| \leq \|\text{space}(\mathsf{B}, x)\| \leq p(\dim x, \|x\|)$.

The existence of a computable bound on $\dim(\text{resource}(\mathsf{P}, x))$ is immediately obtained from the corresponding computable bounds for the operations Step, Bound, Input, and Output. □

4.2 Choiceless Polynomial Time and Abstract State Machines

In this section, we review connections to Choiceless Polynomial Time and Abstract State Machines.

Connection to Choiceless Polynomial Time. Recall that hereditarily definable sets of dimension zero are the same as hereditarily finite sets. Over hereditarily finite sets, we already have a proposal for polynomial time computation, namely čPT, or more accurately čPT+C. Let $\mathbb{A} = (\mathbb{N}, =)$. Below, we consider *finite* relational structures are over a fixed signature, and assume that their elements are elements of $\mathbb{A}$. In particular, they are hereditarily finite sets over $\mathbb{A}$. In this way, čPT and čPT+C take as their inputs finite relational structures, and output hereditarily finite sets. We omit the definitions here, and refer to [17] for a compact definition. We now briefly discuss the relationship to definable while programs. The definitions of čPT and čPT+C are based on the notion of *comprehension terms*. It is clear that constant time operations not using the constant $\mathbb{A}$ are equivalent in expressive power to comprehension terms. A čPT program is specified by by three *comprehension terms*, Step, Halt, and Out, and, given on input x, proceeds by applying to the current value the term Step until Halt produces *true*, and the produced output is obtained the term Out to the current value. Moreover, it is required that both the running time and the space consumption (defined in the same way as in our paper) are bounded by a polynomial in terms of the number of elements of the input structure. Note that if a finite relational structure $\mathbb{K}$ has n elements,

then $\|\mathbb{K}\|$ is bounded by $poly(n)$ for a polynomial depending only on the signature of $\mathbb{K}$.

The following fact therefore summarizes the correspondence between čPT and definable while programs.

Fact 2. *A partial function mapping finite structures over a fixed relational signature to hereditarily finite sets is in* čPT *if and only if it is computed by a definable while program* P *not using the constant* $\mathbb{A}$*, such that*

$$\|\text{resource}(\mathsf{P}, x)\| \leq poly(\|x\|).$$

The equivalence also holds if both formalisms are enriched with counting.

Proof sketch. By Theorem 3.7, we may replace definable while programs by definable state machines in the formulation. The statement then follows, as comprehension terms are equivalent to constant time operations, and the resource bounds are calculated in the same way, up to a polynomial. □

The above fact shows that čPT+C can be seen as the dimension zero case of resource-bounded definable while programs, as all the values occurring in the computation are zero-dimensional sets, i.e., hereditarily finite sets. Another use of the theorem is that it provides an alternative presentation of čPT+C, which we believe is more programmer-friendly. Furthermore, it provides a new angle at attacking the open problem whether čPT+C captures order-invariant polynomial time: although Rossman [17] proved that čPT+C cannot define all functions which are polynomial-time computable and order-invariant, for decision problems, the analogous question remains open (cf. Problem 3 in [11]). By Fact 2 and Fact 1, proving a converse implication in Theorem 4.9 for decision problems would provide a positive answer to the problem. Although probably proving this is not more feasible than resolving the open problem, it might be the case that *refuting* the converse implication is easier than separating čPT+C from order-invariant polynomial time.

Connection to Abstract State Machines Note that the syntax and semantics of definable while programs make sense even when we allow the inputs to be arbitrary sets, not just hereditarily definable ones. Call such unrestricted while programs *abstract while programs* over $\mathbb{A}$, where $\mathbb{A}$ is a fixed background logical structure. Unless specified otherwise, we assume $\mathbb{A} = \emptyset$, and then simply talk about *abstract while programs*. Similarly, allowing definable state machines to input arbitrary sets yields a model which we shall call *abstract state machines* (over $\mathbb{A}$). Note that Theorem 3.7 remains valid in this setting, so abstract while programs are equivalent to abstract state machines, and the equivalence preserves time and space resources.

Abstract state machines as defined above are very similar to the ASM's of Gurevich. Note that there are many variants of ASM's, aimed at modeling sequential computation [12], parallel computation [1], distributed computation [9], quantum computation [10], etc. Furthermore, many of those models are equipped with various features which are meant to make them useful in practice (such as interaction). Our abstract state machines are very closely connected to the parallel ASM's defined by Blass and Gurevich [1]. Our **for** operation corresponds to the operation **do-forall** of parallel ASM's. We omit the definitions here. We only remark that one difference is that in ASM's, states are required to be logical (first-order) structures, whereas in our machines, states are sets. As everything in mathematics, logical structures can be seen as sets. Conversely, a set x can be viewed as a relational structure $(x_*, \in, x)$, as follows. The universe is the *transitive closure* x_* of x, consisting of all elements of x, elements of elements of x, etc. The relation $\in$ is the binary membership relation among elements of x_*. The relation x is a unary predicate selecting those elements of x^* which are elements of x.

Another difference is that in parallel ASM's, the semantics of aggregation is based on multisets, rather than on sets.

References

[1] Andreas Blass and Yuri Gurevich. 2003. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic* 4, 4 (Oct. 2003), 578–651. https://doi.org/10.1145/937555.937561

[2] Andreas Blass, Yuri Gurevich, and Saharon Shelah. 1999. Choiceless polynomial time. *Annals of Pure and Applied Logic* 100, 1-3 (1999), 141–187.

[3] Mikołaj Bojańczyk. [n. d.]. Lecture Notes on Sets with Atoms. ([n. d.]). https://www.mimuw.edu.pl/~bojan/ Available at https://www.mimuw.edu.pl/~bojan/.

[4] Mikołaj Bojańczyk. 2011. Data Monoids. In *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*. 105–116. https://doi.org/10.4230/LIPIcs.STACS.2011.105

[5] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. 2012. Towards nominal computation. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 401–412. https://doi.org/10.1145/2103656.2103704

[6] Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. 2011. Automata with Group Actions. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. 355–364. https://doi.org/10.1109/LICS.2011.48

[7] Mikołaj Bojańczyk and Szymon Toruńczyk. 2012. Imperative Programming in Sets with Atoms. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*. 4–15. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.4

[8] Rodney G. Downey and M. R. Fellows. 2012. *Parameterized Complexity*. Springer Publishing Company, Incorporated.

[9] Andreas Glausch and Wolfgang Reisig. 2009. *An ASM-Characterization of a Class of Distributed Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 50–64. https://doi.org/10.1007/978-3-642-11447-2_4

[10] Erich Grädel and Antje Nowack. 2003. Quantum Computing and Abstract State Machines. In *Abstract State Machines 2003*, Egon Börger, Angelo Gargantini, and Elvinia Riccobene (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–323.

[11] Martin Grohe. 2008. The Quest for a Logic Capturing PTIME. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. 267–271. https://doi.org/10.1109/LICS.2008.11

[12] Yuri Gurevich. 1995. Specification and Validation Methods. Oxford University Press, Inc., New York, NY, USA, Chapter Evolving Algebras 1993: Lipari Guide, 9–36. http://dl.acm.org/citation.cfm?id=233976.233979

[13] Wiflrid Hodges. 1993. *Model Theory*. Cambridge University Press. https://books.google.pl/books?id=Rf6GWut4D30C

[14] Bartek Klin and Michal Szynwelski. 2016. SMT Solving for Functional Programming over Infinite Structures. In *Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016. (EPTCS)*, Robert Atkey and Neelakantan R. Krishnaswami (Eds.), Vol. 207. 57–75. https://doi.org/10.4204/EPTCS.207.3

[15] Eryk Kopczynski and Szymon Toruńczyk. 2017. LOIS: syntax and semantics. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 586–598. http://dl.acm.org/citation.cfm?id=3009876

[16] Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA.

[17] Benjamin Rossman. 2010. *Choiceless Computation and Symmetry*. Springer Berlin Heidelberg, Berlin, Heidelberg, 565–580. https://doi.org/10.1007/978-3-642-15025-8_28

The following sections are appendices containing the proofs omitted from the paper.

A While programs

A.1 Representing states and superstates.

We make several simple but fundamental observations concerning the semantics of while programs, in particular, that states can be represented using hereditarily definable sets, as follow.

A *hereditarily definable function* $f : X \to Y$ between hereditarily definable sets X, Y is a hereditarily definable subset of $X \times Y$ which is (the graph of) a function. In particular, we may consider a hereditarily definable set of hereditarily definable functions, etc. We assume that program variables are identified with natural numbers, encoded as hereditarily definable sets via von Neumann's encoding representing n as $\mathrm{vN}(n) = \{\mathrm{vN}(0), \dots, \mathrm{vN}(n-1)\}$. We suppose that the program uses n variables, and the set of its variables is equal to $\mathrm{vN}(n)$. In particular, all program states are hereditarily definable functions, as they map (finitely many) variables to hereditarily definable states.

We represent a superstate $S = (S_\tau)_{\tau \in T}$ as a function from $S : T \to \{S_\tau : \tau \in T\}$, where $S(\tau) = S_\tau$. Let the initial thread ε be represented by $\emptyset$. In particular, the initial superstate $S^{\bar{x}}$ (with one thread and program state assigning $\bar{x}$ to the input variables and $\emptyset$ to the remaining variables) is a hereditarily definable set. By induction on the structure of the instruction I we easily obtain the following.

Lemma A.1. *Let* I *be an instruction and S, S', S'' superstates. Then $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$ and $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S''$ implies $S' = S''$. Moreover, S and S' have the same sets of threads, i.e.,* $\mathrm{dom}\ S = \mathrm{dom}\ S'$. *Furthermore, if S is hereditarily definable, then S' is hereditarily definable, too.*

In particular, since $S^{\bar{x}}$ is hereditarily definable when $\bar{x}$ is a tuple of hereditarily definable sets, we may always assume that all superstates are hereditarily definable.

A.2 Proof of Theorem 3.7

We start by observing that the operations used in the rules of the semantics of while programs (cf. Fig 1) are constant time operations.

Lemma A.2. *Fix a finite, hereditarily definable set of variables V. Let c be a condition, e an expression, and x a variable. Then the following operations, defined in Figure 1, are constant time operations:*

- $S \mapsto S[c]$,
- $S \mapsto S[x/e]$,
- $S \mapsto \mathrm{Split}(S, x, e)$,
- $S \mapsto \mathrm{Aggregate}(S)$.

Proof. By writing while programs without **while** loops. □

A *derivation* is a finite, ordered, rooted tree whose nodes are labeled by assertions of the form $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$ where S, S' are superstates and I is an instruction, with the property that if a node is labeled by an assertion c then the sequence $c_1, \dots, c_n$ of labels of its children (from left to right) is such that $\frac{c_1\ c_2\ \dots\ c_n}{c}$ is a rule of the semantics of while programs. We say that the derivation is a *derivation of* the assertion labeling the root. By definition, $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$ if and only if there is a derivation of $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$. Note that our semantics is such that for any superstate S and instruction I, there is at most one derivation of an assertion of the form $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$. We say that a derivation *uses superstates* satisfying a property $\mathcal{P}$ if all superstates occurring in the labels of the nodes of the derivation satisfy property $\mathcal{P}$.

For a superstate S, let $\mathrm{mem}(S)$ denote the set of all values of all program variables in all program states of the superstate S, i.e., $\mathrm{mem}(S) = \bigcup_x \{\gamma(x) : \gamma \in \mathrm{rng}\ S\}$, where the union ranges over all program variables x.

Lemma A.3. *Let* P *be a while program and x an input such that* P *terminates on input x. Then the derivation $S^x \rightarrowtail\!\!\llbracket\ \mathtt{P}\ \rrbracket\!\!\to\ S'$ uses superstates S with* $\mathrm{mem}(S) \subseteq \mathrm{space}(\mathsf{P}, x)$. *Moreover, the number of nodes of the derivation is bounded by poly*$(\mathrm{time}(\mathsf{P}, x))$.

Lemma A.4. *Fix a while program* P. *There is a constant time operation K such that for every input x for which* P *terminates, the derivation of $S^x \rightarrowtail\!\!\llbracket\ \mathtt{P}\ \rrbracket\!\!\to\ S'$ uses superstates with $S \subseteq K(\mathrm{space}(\mathsf{P}, x))$.*

Proof. We show that there is a constant time operation D such that $\mathrm{dom}\ S \subseteq D(\mathrm{space}(\mathsf{P}, x))$, and a constant time operation R such that $\mathrm{rng}\ S \subseteq R(\mathrm{space}(\mathsf{P}, x))$. This will imply $S \subseteq D(\mathrm{space}(\mathsf{P}, x)) \times R(\mathrm{space}(\mathsf{P}, x))$, yielding the lemma.

Assume that the program P uses n program variables, corresponding to the elements of $\mathrm{vN}(n)$. For a hereditarily definable set X, let $R(X)$ be the set of all valuations from $\mathrm{vN}(n)$ to X. Note that R is a constant time operation. By definition of $\mathrm{mem}(S)$ and of R we immediately get that for every superstate S, $\mathrm{rng}\ S \subseteq R(\mathrm{mem}(S))$. In particular, since by Lemma A.3, $\mathrm{mem}(S) \subseteq \mathrm{space}(\mathsf{P}, x)$ for every superstate S in the derivation of $S^x \rightarrowtail\!\!\llbracket\ \mathtt{P}\ \rrbracket\!\!\to\ S'$, we get that every such superstate S satisfies $\mathrm{rng}\ S \subseteq R(\mathrm{space}(\mathsf{P}, x))$.

We now proceed to proving that $\mathrm{dom}\ S \subseteq D(\mathrm{space}(\mathsf{P}, x))$ for every superstate S appearing in the derivation, where D is some constant time operation.

Observe that by the semantics of while programs (Fig. 1), particularly, the definition of the Split operation, for every superstate S appearing in the derivation, every thread $\tau \in \mathrm{dom}\, S$ is a pair (τ', v), which consists of a thread τ' from some other superstate, and some value v. Define the *length* of a thread inductively as follows: the length of the initial thread ε is 0; the length of a thread of the form (τ', v) is one plus the length of the thread τ.

Define the *for-depth* of an instruction I as the maximal nesting of **for** loops in I. Formally, this is defined by induction on the structure of I, as expected.

Let Y^d denote the set of all tuples of elements of Y of length d, where a tuple of length d is encoded as a pair (t, y), for t a tuple of length $d-1$ and $y \in Y$, and ε is the unique tuple of length 0. By $Y^{\leq d}$ we denote the set of all tuples of length at most d. Note that for every fixed d, the function $Y \mapsto Y^{\leq d}$ is a constant time operation.

The following lemma follows by an easy induction on the size of the derivation, and the only non-trivial case to check in the inductive step is the *for* rule.

Lemma A.5. *If* I *is an instruction of for-depth d,* $\mathrm{dom}\ S \subseteq Y^{\leq l}$ *and every superstate T in the derivation of $S \rightarrowtail\!\!\llbracket\ \mathtt{I}\ \rrbracket\!\!\to\ S'$ satisfies* $\mathrm{mem}(T) \subseteq Y$, *then it also satisfies* $\mathrm{dom}\ (T) \subseteq Y^{\leq l+d}$.

Let d_{P} denote the for-depth of the program P; this is a fixed constant. In particular, taking $l = 0$ and $d = d_{\mathsf{P}}$, by Lemma A.3 we get that any derivation of $S^x \rightarrowtail\!\!\llbracket\ \mathtt{P}\ \rrbracket\!\!\to\ S'$ uses superstates S such that $\mathrm{dom}\ S \subseteq D(\mathrm{space}(\mathsf{P}, x))$, where $D(Y) = Y^{\leq d_{\mathsf{P}}}$. This ends the proof of Lemma A.4. □

Proof of Theorem 3.7. To prove Theorem 3.7, we first consider the usual recursive procedure $\mathtt{sem}(S, \mathtt{I})$ which inputs a superstate S and an instruction $\mathtt{I}$ and returns the superstate S' such that $S \rightarrowtail\!\!\![\![\mathtt{P}]\!]\!\!\rightarrow S'$. The procedure recursively applies the rules of the semantics in a left-most fashion: when applying a rule with two premises, first compute the derivation of the left premise, and then of the right premise. Assuming that $\mathtt{P}$ terminates on x, when executing the procedure $\mathtt{sem}$ on the superstate S^x and $\mathtt{P}$, the procedure $\mathtt{sem}$ is invoked (by recursive calls) a number of times polynomial in $\mathrm{time}(\mathtt{P}, x)$, by Lemma A.3. In each recursive call, the algorithm needs to compute a bounded number of constant time operations listed in Lemma A.2, or to test whether the current superstate is empty (to determine whether the *no-threads* rule should be applied). Moreover, in each recursive call, the current superstate S is contained in $K(\mathrm{space}(\mathtt{P}, x))$, by Lemma A.4.

To minimize the use of the call stack, we optimize the recursion in the case of the *while* rule. To evaluate

$$\mathtt{sem}(S, \textbf{while } c \textbf{ do } \mathtt{I}),$$

the procedure executes the following pseudocode which accumulates in a variable $\mathtt{T}$ the threads which have already terminated the **while** loop (the pseudocode describes a usual while program operating on expressions representing hereditarily definable sets, and not a definable while program):

```
T:=S[¬c];
while (S[c]≠ ∅) do
  S:=sem(S[c], I);
  T:=T∪S[¬c];
return T;
```

With this adaptation, we see that when evaluating $\mathtt{sem}(S, \mathtt{I})$ for any instruction $\mathtt{I}$, each recursive call invokes $\mathtt{sem}(S', \mathtt{I}')$ with an instruction $\mathtt{I}'$ which is strictly contained in $\mathtt{I}$. Therefore, the depth of the recursion remains bounded by a constant c depending only on the program $\mathtt{P}$.

To finish the proof of Theorem 3.7, we convert the recursive algorithm to a sequential computation on a hereditarily definable state machine, by implementing a stack for storing the local variables for each recursive call. Recall that we only use stacks of length bounded by c, for some constant c. A stack storing values $x_1, \ldots, x_\ell \in X$, where $\ell \leq c$, is represented as a relation $\sigma \subseteq X \times \{1, \ldots, c\}$, namely $\sigma = \{(x_i, i) : 1 \leq i \leq \ell\}$ (where $1, \ldots, c$ are represented as von Neumann numerals).

Lemma A.6. *Let T begin a constant time operation with k input variables and m output variables. For a stack σ with at most c, at least k elements, let $\mathtt{apply}_T(\sigma)$ denote the stack obtained from σ by popping the top k elements $\bar{x}$ and pushing the l elements of the result $T(\bar{x})$, and $\mathtt{apply}_T(\sigma) = \emptyset$ if σ has less than k elements. Then $\mathtt{apply}_T$ is a constant time operation.*

Proof. By writing a while program without **while** loops. □

When simulating the recursive algorithm, the stack stores either superstates S or symbols from a finite alphabet Σ depending only on $\mathtt{P}$ (which consists of subinstructions $\mathtt{I}$ of $\mathtt{P}$, encoded as hereditarily definable sets). Let $L(X) = (K(X) \cup \Sigma) \times \{1, \ldots, c\}$, where K is the constant time operation from Lemma A.4. Since the height of the control stack is bounded by a constant c, at each moment of the computation, the stack is a subset of $L(\mathrm{space}(\mathtt{P}, x))$. Moreover, the operation L is a constant time operation. This proves Theorem 3.7. □

B Oligomorphic atoms

B.1 Orbits and types

We recall some basic facts concerning orbits and logical types. See [13] for an extensive overview, and [3] in the specific context of sets with atoms.

Let S be a finite set or tuple of atoms. An *S-automorphism* is an atom automorphism which fixes S pointwise. Recall that S *supports* x if $\pi \cdot x = x$ for every S automorphism π.

If the structure $\mathbb{A}$ is *oligomorphic*, then one can show that for every hereditarily definable set x and its support S, the set of S-automorphisms acts on the set x, inducing finitely many orbits. Each of these orbits is called an *S-orbits* contained in x. On the other hand, if $y \in x$, then the *S-orbit of y* is the S-orbit contained in x to which y belongs to.

A countable relational structure $\mathbb{A}$ is *homogeneous* if every partial isomorphism $f : \mathbb{A} \to \mathbb{A}$ between finite substructures of $\mathbb{A}$ extends to an automorphism $\pi : \mathbb{A} \to \mathbb{A}$.

If $\bar{a}$ is a tuple of elements of $\mathbb{A}$, then the *atomic type* of $\bar{a}$ is the conjunction of all literals formed by (possibly negated) atomic relations in the vocabulary of $\mathbb{A}$, which hold of the elements of $\bar{a}$. More generally, if $\bar{u} \in \mathbb{A}$ is a tuple of parameters, then the $\bar{u}$-atomic type of $\bar{a}$ is defined as above, but in the structure $\mathbb{A}$ extended by constant symbols for the elements of $\bar{u}$.

It follows from the definition of homogeneity that two tuples $\bar{a}$ and $\bar{b}$ are in the same orbit of the action of $\mathbb{A}$ if, and only if, the atomic type of $\bar{a}$ is equal to the atomic type of $\bar{b}$, up to syntactic equivalence.

It follows that every homogeneous structure over a finite signature is oligomorphic.

If S is a finite set or tuple of elements of $\mathbb{A}$, then we say that two tuples of atoms $\bar{a}$ and $\bar{b}$ are in the *same S-orbit* if there is an atom permutation π which maps $\bar{a}$ to $\bar{b}$ and fixes S pointwise. Generalizing the above observation, assuming $\mathbb{A}$ is homogeneous, $\bar{a}$ and $\bar{b}$ are in the same S-orbit if and only if they have the same S-atomic types.

B.2 Proof of Theorem 3.8

The proof of the implication from 1 to 2 in Theorem 3.8 is omitted here, and can be found e.g. in [15]. We only mention that another approach would be to use Theorem 3.7, and prove the following.

Lemma B.1. *Assume that $\mathbb{A}$ is effective. Then every constant time operation is computable.*

The rest of Section B.2 is devoted to proving the implication from 2 to 1 of Theorem 3.8. Let f be as in the assumptions of the theorem, and let $r : \mathrm{setb}\mathbb{A} \to \mathrm{setb}\mathbb{A}$ be a representation of f. We need to show that if f is computable under representation r then it can be computed by a while program. The key observation is the following lemma, which says that a while program can reverse the representation function r, at least up to automorphisms.

Lemma B.2. *Assume that the atoms are effectively oligomorphic. Let $\bar{a}$ be a tuple of atoms. There is a definable while program which inputs a hereditarily definable set x and outputs a set builder expression α and a tuple of natural numbers $\bar{n}$ such that*

$$\pi(x) = \alpha(r(\bar{n})) \qquad \text{for some } \bar{a}\text{-automorphism } \pi.$$

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

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

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

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

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

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

We now complete the proof of the implication from 2 to 1 in Theorem 3.8. Suppose that f is a function from hereditarily definable sets to hereditarily definable sets which is supported by a tuple of atoms $\bar{a}$, and assume that f is computable under representation r. We present below a while program which computes f. Assume that on input we have a hereditarily definable set x. Use Lemma B.2 to compute α and $\bar{n}$. By condition 2 in the theorem, we can compute a set builder expression β and a tuple of natural numbers $\bar{m}$ such that

$$f(\alpha(r(\bar{n}))) = \beta(r(\bar{m})) \tag{5}$$

Using the same ideas as in Lemma B.2, a while program can compute the $\bar{a}$-orbit of the tuple $r(\bar{m}\bar{n})$, call it y. Compute the set

$$z = \{\beta(\bar{c}) : \bar{b}\bar{c} \in y \text{ are such that } \alpha(\bar{b}) = x\}$$

We claim that z has only one element, namely $f(x)$. By definition $\beta(\bar{c}) \in z$ if and only if $\pi(\bar{b}\bar{c}) = r(\bar{n}\bar{m})$ for some $\bar{a}$-automorphism π and $\bar{b}$ such that $x = \alpha(\bar{b})$. From (5) and invariance of f under $\bar{a}$-automorphisms, we conclude that

$$f(\alpha(\bar{b})) = \beta(\bar{c}).$$

Because the left side is equal to $f(x)$, it follows that $z = \{f(x)\}$. To extract $f(x)$ from the set z, we use a simple while program without **while** loops. This completes the implication from 2 to 1 in Theorem 3.8.

C Tractability – basics

In this section, $\mathbb{A}$ is $(\mathbb{N}, =)$, unless stated otherwise.

C.1 Basic tests

We prove the following.

Proposition 1. *The membership, inclusion and equality problems for hereditarily definable sets are fixed-dimension polynomial.*

The proposition is a consequence of the following results.

Lemma C.1. *Let α, β be set builder expressions with free variables contained in $\bar{x}$. The set of $\bar{x}$-tuples of atoms which satisfy the inclusion $\alpha \subseteq \beta$ is definable by a formula of first-order logic which can be computed based on α and β in polynomial time. Likewise for $\in$ or $=$ instead of $\subseteq$. The formulas use the same parameters and the same variables as the expressions α, β.*

Proof. We construct first-order formulas by induction on the size of the set builder expressions. The constructed formulas will have the property that the bound variables are exactly those which appear in α and β.

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

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

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

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

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

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

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

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

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

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

In particular, applying the above lemma to two set builder expressions α, β without free variables, we obtain a first-order sentence φ which holds if and only if $\alpha \subseteq \beta$ (or $\alpha \in \beta$). To prove Proposition 1, it remains to test whether φ holds in $\mathbb{A}$, i.e., decide the model checking problem for first-order logic in $\mathbb{A}$.

Lemma C.2. *Let $n, k \in \mathbb{N}$ be numbers, and let $\bar{a}$ be an n-tuple of $\mathbb{A}$. The set of $\bar{a}$-atomic types of k-tuples of atoms can be enumerated in time $poly(n)^k$.*

Proof. The atomic $\bar{a}$-atomic types of single elements can be enumerated in time $poly(n)$: those types are simply $x = a_1, x = a_2, \ldots, x = a_n$ and $x \neq a_1 \wedge \cdots \wedge x \neq a_n$. The lemma then follows by induction on k. □

Lemma C.3. *There is an algorithm which, given a first-order formula φ using d variables and p parameters, produces in time $poly(\varphi) \cdot poly(p)^{poly(d)}$ a quantifier-free formula equivalent in $\mathbb{A}$ to φ.*

Proof. The idea is to perform quantifier elimination, in a bottom up fashion, and in each step replace the computed quantifier-free subformula by an equivalent formula of small size, to avoid blowup. We use two kinds of transformations of subformulas: elimination of quantifiers and reduction of size. The details follow.

Observe that a formula of the form

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

is equivalent in $\mathbb{A}$ to the formula

$$\psi' = \bigvee_{\tau} \alpha_\tau(\bar{y}),$$

where:

- τ ranges over the set of all atomic types of $x\bar{y}$ tuples of atoms over the parameters of the formula α,
- for a fixed atomic type τ, α_τ is the formula $\tau'(\bar{y}) \to \alpha'$, where τ' is the projection of the atomic type τ of $x\bar{y}$ tuples to an atomic type of $\bar{y}$ tuples, and the formula α' is obtained from α by replacing every predicate involving x by its boolean value, as specified by the atomic type τ.

That the above formula is equivalent to ψ follows from homogeneity of $\mathbb{A}$. We now analyse the time required for computing ψ' from ψ. By Lemma C.2, the set of all atomic types of $x\bar{y}$ tuples of atoms over the parameters of α can be enumerated in time $poly(p)^d$. Therefore, the above translation of the formula ψ into ψ' can be done in time $|\psi| \cdot poly(p)^d$.

Next, observe that any quantifier-free formula γ with free variables $\bar{y}$ is equivalent to a disjunction of atomic types of $\bar{y}$-tuples. As there are at most $poly(p)^d$ such types, the formula γ is equivalent to a formula γ' of size at most $poly(p)^d$. Moreover, the formula γ' can be computed from γ in time $poly(\gamma) \cdot poly(p)^d$, by taking the disjunction over all atomic types τ of $\bar{y}$-tuples such that $\tau \models \gamma$. As mentioned above, enumerating types can be done in the required time, and checking that $\tau \models \gamma$ is done in time $poly(\gamma)$.

Let φ be a formula using at most d variables, and which uses only existential quantifiers and no universal quantifiers (by de Morgan's law we can assume this). We process the subformulas of φ in a bottom-up fashion, by replacing each subformula by an equivalent one of size at most $poly(p)^d$, as follows. In case of a subformula of the form ψ as above, we replace ψ by the formula ψ' as described above in the first conversion. In case of a subformulas γ of the form $\alpha \vee \beta$, $\alpha \wedge \beta$ or $\neg\alpha$, we compute the equivalent formula γ' as described above in the second conversion. The invariant is thus maintained.

The overall running time of the algorithm is $poly(\varphi) \cdot poly(p)^{poly(d)}$. □

Corollary C.4. *There is an algorithm which, given a first-order sentence φ with p parameters and d variables, decides whether it holds in $\mathbb{A}$ in time $poly(\varphi) \cdot poly(p)^{poly(d)}$.*

The following result allows us, among others, to determine whether a given expression describes a Kuratowski encoding of a pair, and, if so, extract its components.

Lemma C.5. *There is an algorithm running in time $poly(\alpha) \cdot poly(p)^{poly(d)}$ which, given a set builder expression α using p parameters does the following. If α defines a finite set, then the algorithm outputs a list of its elements (without repetitions), represented by expressions; otherwise, it outputs ∞.*

Proof. The algorithm is as follows. If α is an atom or a variable, then it does not define a set; the case when it is a union expression reduces to the case of testing each of its components. Hence, it suffices to consider the case when α is a set expression without free variables (otherwise, it does not define a set). Let α be of the form $\{\beta(\bar{x}) : \bar{x} \in \mathbb{A}, \varphi(\bar{x})\}$. Then α defines an infinite set if and only if there is some assignment of the variables $\bar{x}$ in atoms which satisfies φ and which involves some atom different from all the parameters. To check this, it suffices to model-check the formula $\psi = \exists \bar{x}. \bigvee_{x \in \bar{x}} \bigwedge_{a \in \bar{a}} (x \neq a).\varphi(\bar{x})$, where $\bar{a}$ is the tuple of parameters. The sentence ψ has size polynomial in φ and quantifier depth bounded by $\dim \alpha$, so by Corollary C.4, can be verified in time $poly(\varphi)^{poly(d)}$.

If α defines a finite set, then its elements can be enumerated by listing all expressions $\beta(\bar{v})$, where $\bar{v}$ ranges over all valuations $\bar{v}$ for $\bar{x}$ satisfying φ and involving just parameters (there is a constant number of those). Repetitions can be removed, since testing equality of sets defined by expressions can be performed in the specified time by Lemma C.1 and Corollary C.4. □

Corollary C.6. *In the time specified in Lemma C.5, one can decide whether a given set builder expression defines a Kuratowski encoding of some pair, and, if it does, outputs the corresponding pair of elements.*

C.2 Proof of Lemma 4.4

Proof. Let $\mathbb{A}$ be an infinite structure. Consider the following *model checking* problem over finite graphs: given a finite (simple, undirected) graph G and first-order sentence φ, decide whether G satisfies φ. When parametrized by the size of the formula φ, it is known that this problem is not fixed-parameter tractable (fpt), unless the W hierarchy collapses [8].

The following defines an fpt reduction from the model checking problem to the emptiness problem of hereditarily definable sets. Given a formula φ in the language of graphs and a finite graph G of size n we construct a set builder expression α such that α evaluates to $\emptyset$ iff G satisfies φ.

Let $a_1, \ldots, a_n$ be n distinct atoms, and convert the formula φ to a formula ψ using equality only, by recursively replacing each subformula $\forall x.\gamma$ by $\forall x.((x = a_1) \vee (x = a_2) \vee \ldots \vee (x = a_n)) \to \gamma$, and dually, each subformula $\exists x.\gamma$ by $\exists x.((x = a_1) \vee (x = a_2) \vee \ldots \vee (x = a_n)) \wedge \gamma$, and finally, each atomic formula $E(x, y)$ by a disjunction $\bigvee_{ij \in E(G)} (x = a_i \wedge y = a_j)$.

Let α be the set builder expression $\{\emptyset : \neg\psi\}$, with parameters $a_1, \ldots, a_n$. The size of α is $f(\varphi) \cdot poly(G)$, and its dimension is $g(\varphi)$, for some computable functions f, g. Clearly, $[\![\alpha]\!] = \emptyset$ if G satisfies φ, and $[\![\alpha]\!] = \{\emptyset\}$ otherwise.

Therefore, this defines an fpt reduction from model checking to emptiness. In other words, given an fpt algorithm for the emptiness problem, composing it with the above reduction would yield an fpt algorithm for the model checking problem, implying the collapse of the W hierarchy. □

C.3 Proof of Fact 1

Proof. To prove the right-to-left implication, we observe is that if G is a graph where all vertices are atoms, then it can be defined by a set-builder expression whose dimension is 0, like any hereditarily finite set.

To prove the left-to-right implication, we observe that there is a fixed-dimension polynomial-time algorithm which transforms

a set builder expression which represents a hereditarily finite set into another expression which represents the same set, but which has dimension 0. This follows from Lemma C.5. □

C.4 Another definition of dimension

Call the all-dimension of an expression to be its dimension plus the number of parameters appearing in it. The following lemma shows that such a definition is not a good idea.

Lemma C.7. *Let L be an isomorphism closed class of finite undirected graphs and let $\mathbb{A}$ be an infinite effective structure. Then L is decidable if and only if the following family of hereditarily definable sets is fixed all-dimension tractable:*

$$\{G : G \text{ is a graph in } L \text{ where all vertices are from } \mathbb{A}\} \subseteq \mathsf{hdef}\mathbb{A}.$$

Proof. Clearly, if the above class is recognized by any Turing machine M (not necessarily satisfying any time bounds), then L is decidable: given an encoding of a graph G with n vertices in binary, choose arbitrary atoms $a_1, \ldots, a_n$ to represent its vertices, and build an expression α describing a graph isomorphic to G with vertices $a_1, \ldots, a_n$. By definitionn $G \in L$ if and only if M accepts α.

For the other implication, assume that L is decidable. We first note that, by Corollary C.4, the model checking problem of first-order sentences in $\mathbb{A}$ is fixed-parameter tractable, where the parameter is the number of variables plus the number of parameters of the formula. Next, testing whether two expressions without free variables define equal sets is fixed-parameter tractable, too: by Lemma C.1, this question reduces in polynomial time to model-checking sentences in $\mathbb{A}$. The same holds for testing containment or membership of sets. By Lemma C.5, there is an algorithm deciding whether an input expression α defines a finite set, and if so, enumerates its elements (without repetitions) in time polynomial in the size of α (for fixed all-dimension); moreover, the resulting set has bounded size (for fixed all-dimension).

Given an expression α, we wish to determine whether it defines a finite graph. Using Corollary C.6 we first check that α is the Kuratowski encoding of a pair (V, E). Next, check that V is a finite set, that each element of V is an atom, and that each element of E is a pair (u, v) where u, v belong to V. For each such pair, mark that u and v are adjacent by placing a 1 on the (u, v) position in an $V \times V$ matrix. In this way, if all the steps succeeded, we have computed the adjacency matrix of a finite graph described by α, in polynomial time (for fixed all-dimension). Moreover, the graph has only at most a bounded number of vertices, and therefore, testing whether it belongs to the class L can be performed in constant time, given the adjacency matrix. □

D Proof of Lemma 4.7

Section D is devoted to the proof of Lemma 4.7. Throughout this section, $\mathbb{A}$ is assumed to be $(\mathbb{N}, =)$.

Orbit dag. An *orbit dag* is a finite directed acyclic graph (dag) D, with possibly parallel edges, which satisfies the following properties:

- there is one *root* vertex, i.e., vertex with no ingoing edges,
- each vertex v is labeled by a finite sets of atoms λ_v,
- the label of a leaf (vertex with no outgoing edges) contains at most one atom,
- every edge e from v to w is labeled by a mapping $\lambda_e : \lambda_w \to \mathbb{A}$.

For an edge e from v to w we call the atoms in $\mathrm{rng}\, \lambda_e \cap \lambda_v$ *free parameters* for the edge e, and the atoms in $\mathrm{rng}\, \lambda_e - \lambda_w$ are called *bound parameters*. See Figure 2 for an example orbit dag.

Definable set to orbit dag. Given a set x, we define an orbit dag *associated* to x, as follows. Recall that x_* denotes the transitive closure of x.

Choose a set V of representatives of the equivalence relation on x_* of being in the same $\emptyset$-orbit (i.e., $u, v \in x_*$ are equivalent if there is an atom permutation π such that $\pi(u) = v$). The vertex set of the constructed dag is $V \cup \{x\}$. Let v be a vertex which is a nonempty set with least support S, and let $o_1, \ldots, o_k$ be the S-orbits of v. Define the label λ_v of the vertex v to be the least support of v, and create outgoing edges $o_1, \ldots, o_k$, where the edge o_i leads to the unique element $w \in V$ whose $\emptyset$-orbit contains o_i, and is labeled by the restriction of π to λ_w, where π is an arbitrarily chosen permutation such that $\pi(w) \in o_i$.

This ends the description of the orbit dag associated to x. The construction depends on the choice of representatives V and the choices of the edge labels. However, this dependence is inessential for our purposes, so we allow ourselves to speak of *the* orbit dag associated to x. Note that it has finitely many vertices and edges, and that x is the unique vertex with no incoming edge.

Example D.1. Let $x = \{\{a, b\} : a, b \in \mathbb{A}, a = 1 \vee a = 2\}$. The orbit dag D associated to x is depicted in Figure 2. The expressions

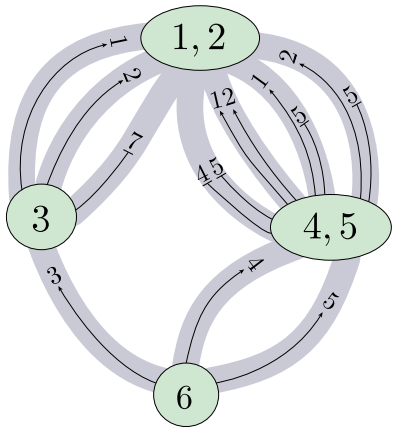

Figure 2. The edges are the thick lines, directed from top to bottom. The root vertex is the set x, its left child is the set $\{3\}$ and its right child is the set $\{4, 5\}$. The leaf is the atom 6. The underlined parameters are the bound parameters. The thin edges ending with an arrowhead indicate that a parameter is mapped to a free parameter; those edges can be thought of as variable substitution. The remaining edges indicate that a parameter is mapped to a bound parameter; those edges will substituted by bound variables.

constructed as in the proof of Lemma D.2 to the vertices and edges

of the orbit dag D depicted in Figure 2 are, from bottom to top:

$$\begin{aligned}
\alpha_6 &= 6,\\
\alpha_{e_3} &= \{3\},\\
\alpha_{\{3\}} &= \{3\},\\
\alpha_{e_1} &= \{\{1\}\},\\
\alpha_{e_2} &= \{\{2\}\},\\
\alpha_{e_7} &= \{\{x_7\} : x_7 \in \mathbb{A}, (x_7 \neq 1) \wedge (x_7 \neq 2)\},\\
\alpha_{e_4} &= \{4\},\\
\alpha_{e_5} &= \{5\},\\
\alpha_{\{4,5\}} &= \{4\} \cup \{5\},\\
\alpha_{e_{2,5}} &= \{\{2\} \cup \{x_5\} : x_5 \in \mathbb{A}, (x_5 \neq 1) \wedge (x_5 \neq 2)\},\\
\alpha_{e_{1,5}} &= \{\{1\} \cup \{x_5 : x \in \mathbb{A}\} : x_5 \in \mathbb{A}, (x_5 \neq 1) \wedge (x_5 \neq 2)\},\\
\alpha_{e_{1,2}} &= \{\{1\} \cup \{2\}\},\\
\alpha_{e_{4,5}} &= \{\{x_4\} \cup \{x_5\} : x_4, x_5 \in \mathbb{A}, (x_4 \neq 1) \wedge (x_4 \neq 2)\\
&\qquad \wedge (x_5 \neq 1) \wedge (x_5 \neq 2) \wedge (x_4 \neq x_5)\},\\
\alpha_x &= \alpha_{e_1} \cup \alpha_{e_2} \cup \alpha_{e_7} \cup \alpha_{e_{2,5}} \cup \alpha_{e_{1,5}} \cup \alpha_{e_{1,5}} \cup \alpha_{e_{4,5}}.
\end{aligned}$$

Note that the set builder expression α_x describes the set x.

Orbit dag to expression. For each vertex v of an orbit dag D we define an expression α_v inductively, as follows. The invariant is that α_v uses as parameters those atoms which are in the label of v.

If v is a leaf then its label contains at most one atom. If $\lambda_v = \{a\}$, let $\alpha_v = a$; otherwise, let $\alpha_v = \emptyset$. Suppose that v is a vertex such that the expression α_w has been already defined for each successor w of v. Fix an edge e with source v and target w. Let $\alpha'_w(\bar{x})$ be the expression obtained from α_w as follows. First, rename the parameters in the label of w via the mapping f. Next, replace the tuple $\bar{a}$ of parameters which are bound with respect to e by a tuple of fresh variables $\bar{x}$. Define α_e as the set expression $\{\alpha'_w(\bar{x}) : \bar{x} \in \mathbb{A}, \tau(\bar{x})\}$, where $\tau(\bar{x})$ is the $\lambda(v)$-atomic type of the tuple $\bar{a}$, i.e., the conjunction of all equalities and inequalities between the elements of $\bar{a}$ and of $\lambda(v)$. Finally, define α_v as the union of the expressions α_e, for e ranging over the edges leaving v. This ends the inductive definition.

The set builder expression α_D defined by the orbit dag D is the expression α_v, where v is the root of D. The following lemma is immediate by construction.

Lemma D.2. *The expression α_D can be computed in polynomial time from a given orbit dag D.*

Lemma D.3. *Let x be a hereditarily definable set and let $v \in x_*$. Then $\dim v \leq \dim x$.*

Proof. Let $u \in v_*$ be such that the set C of atoms in the least support of u which do not belong to the least support of v has cardinality equal to $\dim v$. Let π be an atom permutation which fixes the least support of v pointwise and maps the set U to any set disjoint from those atoms which are in the least support of x but not in the least support of v. Then $\pi(v) = v$, so $\pi(u) \in v_*$. Now we have that $\pi(u) \in x_*$ and $\pi(C)$ is a set of atoms which are in the least support of $\pi(u)$ but not in the least support of x. It follows that $\dim x \geq |\pi(C)| = |C| = \dim v$. □

Lemma D.4. *Let x be a hereditarily definable set and D be the orbit dag associated to x. Then the set builder expression α_D corresponding to D describes the hereditarily definable set x. Moreover,*

1. *$\|\alpha_D\| \leq f(\dim x, \|x\|)$ and $\|x\| \leq f(\dim x, |D|)$, where $|D|$ is the number of vertices and edges in D, and $f(k,n) = poly(n)^{poly(k)}$;*
2. *Up to a renaming bound variables, $\dim \alpha_D \leq 2 \dim x$.*

Proof. Let D be the orbit dag associated to x. Recall that its vertices are hereditarily definable sets. Inductively, we show that the set builder expression α_v associated to a vertex v is such that α_v defines the hereditarily definable set v, and that α_v has at most $\dim x$ free variables.

In the base case, if v is a leaf, then α_v defines v and $\dim \alpha_v = \dim v$. Assume that v is a vertex such that for every successor w of v, the inductive assumption holds. Let e be an edge outgoing from v to some successor w, with label π. Note that e is an S-orbit of v, where S is the least support of v. By construction, the set builder expression α_e describes the set e. Suppose that $\dim v = k$. In particular, since $\pi(w) \in v$, the least support of $\pi(w)$ has at most k elements which are not in the least support of v, i.e., the edge e has at most k free parameters. Therefore, the expression α_e has at most $k = \dim v$ free variables. Finally, by definition, α_v is the union of all expressions α_e, for e ranging over all S-orbits of v. In particular, the set builder expression α_v describes the set v, and has at most $\dim v \leq \dim x$ (by Lemma D.3) free variables, ending the inductive proof.

In particular, the set builder expression associated to the root vertex describes the set x, and every its subexpression has at most $\dim x$ free variables. Therefore, up to a renaming of bound variables, $\dim \alpha_D \leq 2 \dim x$.

The size of the expression α_D is polynomial in the size of the description of the orbit dag D by Lemma D.2. The orbit dag D associated to x has at most $\|x\|$ vertices and between a vertex v and its successor w there are at most $f(|\lambda_w - \lambda_v|, |\lambda_v|)$ edges, where $f(k,n) = poly(n)^{poly(k)}$ is the bound from C.2 on the number of $\bar{a}$-atomic types of k-tuples of variables and $\bar{a}$ is a tuple of atoms of length n. Since $|\lambda_w - \lambda_v|$ is bounded by $\dim w \leq \dim x$ and $|\lambda_v| \leq \|x\|$, it follows that D has at most $\|x\| \cdot f(\dim x, \|x\|)$ edges. A similar argument shows that $\|x\| \leq f(\dim x, |D|)$. This proves the lemma. □

Fixed-dimension polynomial computability of orbit dags. We show in Lemma D.8 that the orbit dag associated to x and the corresponding set builder expression can be computed from a set builder expression α defining x in fixed-dimension polynomial time. For this we need the following three lemmas.

Lemma D.5. *The function mapping a hereditarily definable set x to its least support is fixed-dimension polynomial.*

Proof. Suppose that a hereditarily definable set x is given, and represented by a set builder expression α without free variables, and using some set of parameters P. Since P is a support of x, it follows that $S \subseteq P$.

For $b \in \mathbb{A}$, denote by $\alpha[a/b]$ the expression obtained by replacing the parameter a by the parameter a in the definition.

Claim 1. *Let $a \in P$ and $U = P - \{a\}$. The following conditions are equivalent.*

1. *a does not belong to the least support of x;*
2. *every element $b \neq a$ in the U-orbit of $a \in \mathbb{A}$ is such that $\alpha[a/b]$ defines x;*
3. *some element $b \neq a$ in the U-orbit of $a \in \mathbb{A}$ is such that $\alpha[a/b]$ defines x.*

We show the implications in a cyclic fashion.

(1)→(2). Take any b in the U-orbit of a, and let π be an automorphism fixing U which maps a to b. Then π fixes x, since by assumption, U supports x. On the other hand x is the element defined by α and is therefore is mapped by π to the element x' defined by the expression α' obtained from α by renaming the parameters according to π. But α' is $\alpha[a/b]$. This proves that $x = x$, as required.

For the implication (2)→(3), it suffices to observe that the U-orbit of $a \in \mathbb{A}$ does not contain only a – otherwise, U would support a, and hence a would have two disjoint supports, $\{a\}$ and $P - \{a\}$, implying that the least support of a is empty, i.e., a is invariant under all automorphisms, and no such element exists in $\mathbb{A} = (\mathbb{N}, =)$.

(3)→(1). Suppose that b is as in condition (3). Since a and b are in the same U-orbit of $\mathbb{A}$, there is an automorphism π which is the identity on U and maps a to b. In particular, $\pi(x)$ is the set defined by the expression α with all parameters translated via π; this set is equal to x by assumption. Since $\pi(x) = x$, it follows that π maps the least support of x to the least support of x. Therefore, a cannot belong to the least support of x, as $b = \pi(a)$ does not even belong to its superset P.

This proves the claim.

Using the second condition in the claim, we may compute the least support of a set x represented by an expression α, as follows: for each $a \in P$ (where P are the parameters in α), pick an arbitrary $b \in \mathbb{A} - P$ (hence in the U-orbit of a, since $\mathbb{A} = (\mathbb{N}, =)$), and test whether $\alpha[a/b]$ and α define the same sets. If so, a is not in the least support of x. By the above claim, this allows to compute the least support of x, proving LemmaD.5. □

Lemma D.6. *1) There is an algorithm which runs in fixed-dimension polynomial time and inputs two expressions $\alpha(\bar{x}), \beta(\bar{y})$ and a tuple of atoms $\bar{s}$, and returns a formula $\varphi(\bar{x}, \bar{y})$ using at most $\dim \alpha + \dim \beta$ variables such that, for given valuations $\bar{a}$ and $\bar{b}$, $\varphi(\bar{a}, \bar{b})$ holds if and only if $\alpha(\bar{a})$ and $\beta(\bar{b})$ are in the same $\bar{s}$-orbit.*

2) There is a fixed-dimension polynomial function which, given a hereditarily definable set v and a tuple of atoms $\bar{s}$, computes the $\bar{s}$-orbit of v.

3) The function which, given a hereditarily definable set X and its support S, computes the set of S-orbits of X, is fixed-dimension polynomial.

Proof. 1) Define

$$\varphi(\bar{x}, \bar{y}) = \exists \bar{z}. \tau_{\bar{s}}(\bar{z}, \bar{x}) \wedge \varphi_{=}(\bar{z}, \bar{y}),$$

where $\tau_{\bar{s}}(\bar{z}, \bar{x})$ is the formula which says that $\bar{x}$ and $\bar{z}$ have the same atomic types over $\bar{s}$, and $\varphi_{=}$ is the formula which says that $\alpha(\bar{z})$ and $\beta(\bar{y})$ define the same elements, given by Lemma C.1.

2) Let $v = \alpha(\bar{a})$ for some set builder expression $\alpha(\bar{x})$ without parameters. Let $\tau(\bar{x})$ be the $\bar{s}$-atomic type of $\bar{a}$. We claim that the set U defined by the expression $\{\alpha(\bar{x}) : \bar{x} \in \mathbb{A}, \tau(\bar{x})\}$ is the $\bar{s}$-orbit of v. Clearly, $v \in U$. Furthermore, if π is an atom permutation fixing $\bar{s}$, then $\pi(\bar{a})$ has the same $\bar{s}$-atomic type as $\bar{a}$, and therefore, $\alpha(\pi(\bar{a})) = \pi(\alpha(\bar{a})) = \pi(v)$ belongs to U. Hence, U contains the $\bar{s}$-orbit of v. Conversely, if $u \in U$, then $u = \alpha(\bar{u})$ for some tuple of atoms $\bar{u}$ satisfying $\tau(\bar{u})$, i.e., such that there is a permutation π fixing $\bar{s}$ which maps $\bar{a}$ to $\bar{u}$. Then $u = \pi(a)$, i.e., u belongs to the $\bar{s}$-orbit of a.

3) Let X be a nonempty hereditarily definable set. Let $\sim$ denote the equivalence relation on X whose equivalence classes are the S-orbits of X. In other words, $x \sim y$ if and only if there exists an atom automorphism π such that π fixes S pointwise, and $\pi(x) = y$. We claim that, given X and S, the relation $\sim$ can be computed in fixed-dimension polynomial time. Let α be the expression defining X, and suppose $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$, where each α_i is a set expression of the form

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

For the set defined by the above expression α_i, compute its set U_i of $\bar{s}$-orbits, using the formula φ defined in the first part of the lemma for the expressions $\beta(\bar{x})$ and $\beta(\bar{y})$:

$$U_i = \{\{\beta(\bar{x}) : \bar{x} \in \mathbb{A}, \varphi(\bar{x}) \wedge \varphi_\beta(\bar{x}, \bar{y})\} : \bar{y} \in \mathbb{A}, \varphi(\bar{y})\}.$$

The set $U_1 \cup \cdots \cup U_n$ is the set of all $\bar{s}$-orbits of X. It is finite, so, by Lemma C.5, it can be effectively enumerated. □

Lemma D.7. *There is an algorithm running in fixed-dimension polynomial time which inputs a set builder expression α and, if α represents a nonempty set x, outputs a set builder expression describing some hereditarily definable set y which belongs to x.*

Proof. If α is a union expression, then choose any nonempty component of this union, using Proposition 1 to test emptiness. Therefore, we may assume that α is a set expression, of the form $\{\beta(\bar{x}) : \bar{x} \in \mathbb{A}, \varphi(\bar{x})\}$. Using quantifier elimination (Lemma C.3), replace φ by a quantifier-free formula which is a disjoint union of atomic types. Choose one of those types $\tau(\bar{x})$ arbitrarily, and produce a sequence $\bar{a}$ of atoms satisfying $\tau(\bar{a})$. Then $\beta(\bar{a})$ defines a set which belongs to the set x defined by α. □

Lemma D.8. *There is an algorithm which inputs a set builder expression α and computes the orbit dag associated to the corresponding set in fixed-dimension polynomial time.*

Proof. Fix a set builder expression α. Let α_d denote the union of expressions $\{\beta(\bar{x}) : \bar{x} \in \mathbb{A}\}$, where $\beta(\bar{x})$ ranges over all subexpressions of α of nesting depth d.

The algorithm computes an increasing sequence of dags $D_0 \subseteq D_1 \subseteq \ldots \subseteq D_h$, where h is the nesting depth of α. The invariant is that D_d is the orbit dag associated to the set defined by the expression α_d, with the root removed.

In the base case, α_0 defines the set $\mathbb{A}$ and D_0 consists of a single vertex which is an arbitrarily chosen atom. The dag D_{d+1} is constructed by extending D_d as follows. Using the third part of Lemma D.6, compute the set of all $\emptyset$-orbits of the set defined by the expression α_{d+1}, and remove those orbits which already occurred as orbits of α_d. For each remaining orbit, using Lemma D.7 choose a representative v, and add v as a vertex to D_{d+1}. Compute the least support S of v using Lemma D.5, and label v by a tuple $\bar{a}_v$ enumerating S. Compute the $\bar{a}_v$-orbits $o_1, \ldots, o_k$ of v using the last part of Lemma D.6. For every o_i, find the unique vertex w of D_d whose $\emptyset$-orbit contains o_i, by using the second part of Lemma D.6 and Proposition 1. Create a new edge from v to w in D_{d+1}. We label the edge as follows. Compute the $\emptyset$-orbit U of the pair $(w, \bar{a}_w)$, and, using Lemma D.7, pick any element $(v, \bar{e})$ in $(U \cap o_i) \times \mathbb{A}^{|\bar{a}_w|}$. Label the edge from v to w by the tuple $\bar{e}$. By construction, $\bar{e} = \pi(\bar{a}_w)$ for some atom permutation π such that $\pi(w) \in o_i$.

If h is the nesting depth of α, then α_h is the set builder expression $\{\alpha\}$, and D_h has a vertex v which is in the same orbit as the set defined by α. Actually, when choosing the representative of this

orbit, we could have chosen the set defined by α, so lets assume that v is equal to the set defined by α.

It follows by construction that the dag consisting of those edges and vertices, which can be reached in D_d from v by a directed path, is the orbit dag associated to the set defined by α.

We analyse the running time of the algorithm. The algorithm processes each subexpression of the expression α a polynomial number of times, each time performing a constant number of fixed-dimension polynomial operations. In effect, the algorithm runs in fixed-dimension polynomial time. □

Corollary D.9. *There is an algorithm running in fixed-dimension polynomial time which inputs a set builder expression representing a set x and computes the expression α_x corresponding to the orbit dag associated to x.*

Proof of Lemma 4.7. Before proving Lemma 4.7, we prove the following.

Lemma D.10. *Let α be a set builder expression defining a set x. Then x is supported by the set of parameters appearing in α. Moreover, α can be converted to an expression β with $\dim \beta \leq \dim \alpha$, such that β defines x and uses as parameters exactly the atoms in the least support of x.*

Proof. The first part of the lemma is clear, since the semantics of set builder expressions is invariant under atom permutations, i.e., $\alpha(\pi \cdot \bar{a}) = \pi(\alpha(\bar{a}))$. It follows that if π fixes $\bar{a}$ then it must also fix $\alpha(\bar{a})$, hence $\alpha(\bar{a})$ is supported by $\bar{a}$.

For the second part of the lemma, we proceed by induction on the nesting depth of the expression α. The base case is when α is a parameter; then there is nothing to do.

Let $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$ be an expression of nesting depth at least one, defining a set x whose least support is S. We construct an expression β defining x with $\dim \beta \leq \dim \alpha$, which only uses parameters appearing in S.

Let P be the set of parameters occurring in the expression α and not in S.

Consider an element $x \in X$. There is an atom permutation π which fixes S pointwise and maps the least support of x to a set disjoint from P. Denoting $\pi(x)$ by y, from S-invariance of X we conclude that $y \in X$. In particular, y belongs to the set defined by some expression $\alpha_i = \{\beta(\bar{x}) : \bar{x} \in \mathbb{A}, \varphi(\bar{x})\}$, and there is some tuple $\bar{a}$ such that $\varphi(\bar{a})$ holds and $y = \beta(\bar{a})$. Let $\beta_{\bar{a}}$ be the expression obtained from β by replacing the variables $\bar{x}$ by the corresponding values from $\bar{a}$. Then $\beta_{\bar{a}}$ defines the set $y = \beta(\bar{a})$ and has nesting depth smaller than α. Since the least support of y is disjoint from P and contained in the union of S and the tuple $\bar{a}$, by inductive assumption applied to $\beta_{\bar{a}}$, there is an expression β' which defines y and uses parameters from S and from $\bar{a}$, and with $\dim \beta' \leq \dim \beta_{\bar{a}}$. Let $\gamma(\bar{x})$ be the expression obtained by replacing the parameters from $\bar{a}$ occurring in β' by corresponding variables in $\bar{x}$. More precisely, if $\bar{x}$ is a tuple of variables $x_1, \ldots, x_k$ and $\bar{a}$ is a tuple of atoms $a_1, \ldots, a_k$, and a parameter a appears in γ, then we replace a by an arbitrary variable x_i such that $a = a_i$. Then $\gamma(\bar{a}) = v$, and $\gamma(\bar{x})$ uses at most as many variables other than $\bar{x}$ as $\beta(\bar{x})$ does.

Let $\tau(\bar{x})$ denote the S-atomic type of $\bar{a}$. Consider the set builder expression

$$\eta_x = \{\gamma(\bar{x}) : \bar{x} \in \mathbb{A}, \tau(\bar{x})\},$$

and let E_x be the set it defines.

By construction, η_x uses only parameters from S, and therefore, by the first part of the lemma, the set E_x is S-invariant. Moreover, since τ is an S-atomic type, E_x is the smallest S-invariant set containing y (since $y = \gamma(\bar{a})$). As $y \in X$ and X is S-invariant, it follows that $E_x \subseteq X$. Also, since E_x is S-invariant and π fixes S pointwise, $x \in E_x$. Finally, note that the expression η_x has dimension at most $\dim \alpha$. This is because η_x is obtained from α_i by replacing $\beta(\bar{x})$ by $\gamma(\bar{x})$, and γ uses as at most as many variables other than $\bar{x}$ as β.

As the set X has finitely many S-orbits, the family of all sets E_x constructed as above, ranging over all $x \in X$, is in fact finite. Hence, there are finitely many expressions η_x defining all elements of this family, and each of them uses only parameters from S. Moreover, the union of these expression defines a set which contains X (by construction, since $x \in E_x$ for all $x \in X$), and is contained in X (as $E_x \subseteq X$), hence is equal to X. This ends the inductive proof of the second part of the lemma. □

Proof of Lemma 4.7. We start by showing that $\dim x \leq \dim \alpha$, for every $\alpha \in \text{setb}\mathbb{A}$ defining x. By the second part of Lemma D.10, we may assume that α only uses as parameters those atoms which are in the least support of x.

We proceed by induction on the depth of the set x, which is defined as 0 if x is an atom or the empty set, and one plus the largest depth of an element in x otherwise (this is always a finite number, if x is a hereditarily definable set, as proved easiy by induction on the nesting depth of an expression defining x).

If x has depth 0, then $\dim x = 0$ so there is nothing to prove. Suppose that x had depth $d + 1$, and let α be an expression defining x.

Let y be an element of x_*. Then, by the semantics of hereditarily definable sets, y is of the form $\beta(\bar{a})$, for some subexpression β of α, and some valuation $\bar{a}$ of its free variables. By the first part of Lemma D.10, the least support of y is contained in the set of parameters appearing in $\bar{a}$ and in α (as β is a subexpression of α). Therefore, the set difference of the least support of y and the least support of x (which is equal to the parameters appearing in α by assumption) is contained in $\bar{a}$. Hence, the size of this difference is at most as large as the length of the tuple $\bar{a}$, which is at most $\dim \alpha$. This ends the proof that $\dim x \leq \dim \alpha$.

We now prove that $\|x\| \leq h(\dim \alpha, \|\alpha\|)$ for some function h which is polynomial in the second argument. Let D be the orbit dag D associated to x, and let α_D be the set builder expression defined by D. By Lemma D.8, D can be computed from α in fixed-dimension polynomial time, and by Lemma D.2, α_D can be computed in polynomial time from D. In particular, $\|\alpha\|_D \leq g(\dim \alpha, \|\alpha\|)$, for some function g which is polynomial in the second argument. Furthermore, $\|\beta\| \leq \mathit{poly}(|D|)$ by Lemma D.2, where $|D|$ is the number of vertices and edges of D. By Lemma D.4, $\|x\| \leq f(\dim x, |D|)$, where f is polynomial in the second argument. Altogether, $\|x\| \leq h(\dim \alpha, \|\alpha\|)$ for some function h which is polynomial in the second argument, as required.

The second part of Lemma 4.7 follows by taking $\alpha = \alpha_D$, which satisfies the required conditions by Lemma D.4. □

E Proof of Lemma 4.10

Atoms are $(\mathbb{N}, =)$. We show that every constant time operation is fixed-dimension polynomial. We start with the following observation.

Fact 3. *Every constant time operation is defined by a definable while program which, apart from sequencing, uses only the following constructs:*

- *assignments of the form* $x := y$ *or* $x := x \cup \{y\}$, $x := \mathbb{A}$ *or* $x := |y|$ *(if counting is allowed),*
- **for** *loops of the form* **for** x **in** y,
- *conditionals of the form* **if** $(x = y)$ *or* **if** $(R(x_1, \ldots, x_n))$,

where $x, y, x_1, \ldots, x_n$ *are variables.*

Proof. By implementing constant time operations for the expressions $\cup, \cap, -$ and conditionals by instructions which only use the constructs listed above. □

In the rest of this section, all instructions are of the form specified in the fact above, and $\mathbb{A}$ is $(\mathbb{N}, =)$.

Lemma E.1. *Let* x, y *be variables. Then the following operations are fixed-dimension polynomial (cf. Figure 1):*

0. $S \mapsto S$ *if* S *is a hereditarily definable superstate and* $S \mapsto \emptyset$ *otherwise; below we assume that* S *is a hereditarily definable superstate;*
1. $S \mapsto S[x = y]$;
2. $S \mapsto S[x/y]$;
3. $S \mapsto S[x/(x \cup \{y\})]$;
4. $S \mapsto S[x/|y|]$;
5. $S \mapsto \mathrm{Split}(S, x, y)$;
6. $S \mapsto \mathrm{Aggregate}(S)$;
7. $x_0, \ldots, x_{n-1} \mapsto S^{\bar{x}}$, *where* $x_0, \ldots, x_{n-1}$ *is an* n*-tuple of variables and* $S^{\bar{x}}$ *is a superstate with one thread* ε *and program state mapping the* i*th variable to* x_i,
8. *a reverse operation, which maps a program state of the form* $S^{\bar{x}}$ *to the* n*-tuple of hereditarily definable sets* $x_0, \ldots, x_{n-1}$.

Proof. Let α be an expression representing the superstate S. Compute the orbit dag D associated to S by applying Lemma D.8. In each case, we modify D to obtain an orbit dag such that the set builder expression obtained using Lemma D.2 describes the resulting superstate.

Note that S is a superstate if and only if D has the following structure:

- The elements of S are pairs (τ, γ), where τ is a thread and γ is a program state;
- Each program state γ is a finite set of pairs (n, x), where n is a variable (encoded as a natural number via the von Neumann encoding) and x is a hereditarily definable state.

A pair (a, b) is encoded via the Kuratowski encoding as $\{a, \{a, b\}\}$. If a vertex v of the orbit dag D represents a pair (a, b), then v has two successors in D, namely a and $\{a, b\}$, and a is a successor of $\{a, b\}$.

Based on these observations, we see that one can determine in polynomial time (with respect to D) whether S is a superstate, and distinguish in polynomial time the vertices of D which are the threads τ, as well as the vertices of D which are program states γ (note that these sets need not be disjoint), and, given a vertex which is a program state γ and a variable x, one can compute in polynomial time the value $\gamma(x)$.

We now describe the operations listed in the lemma, performed on the level of orbit dags.

1. The resulting orbit dag is obtained from D by removing those edges which lead from the root of D (representing S) to a pair (τ, γ), where γ is such that $\gamma(x) \neq \gamma(y)$.

2. The resulting orbit dag is obtained from D by performing the following steps for every vertex v (τ, γ) which is a successor of the root. First, create a duplicate v by creating a new vertex v' which has the same successors as v, and only the root as a predecessor (this is because v may be a successor of some nodes other than the root, so we should not modify it). So far, v' still represents the same set as v, i.e., $\alpha_v = \alpha_{v'}$, using the notation from Lemma D.2. Remove the edge from the root to v, effectively disowning the pair (τ, γ) (and its entire $\bar{a}$-orbit, where $\bar{a}$ is the least support of S) from S. Similarly, replace γ in the pair v' by its duplicate γ' constructed as above. γ' has n successors (the same ones as γ) where n is the number of program variables. One of those successors is the pair $(x, \gamma(x))$, and another is the pair $(y, \gamma(y))$, where x and y are the program variables considered in the instruction $S[x/y]$. Remove the edge from γ' to the successor $(x, \gamma(x))$ and instead, create an edge to a new element which is the Kuratowski encoding of $(x, \gamma(y))$ (more specifically, create a new successor $(x, \gamma(y))$ of γ', whose successors are x and a new vertex $\{x, \gamma(y)\}$, which in turn has successors x and $\gamma(y)$).

3. This case is very similar as the previous one.

4. This case is again similar to the previous ones, augmented by the following modification. We need to determine the (von Neumann encoding of) the cardinality of $\gamma(y)$. To this end, we observe that $\gamma(y)$ is finite iff each of its successors has least support contained in the least support of $\gamma(y)$, and, in case it is finite, its cardinality is equal to the number of successors (the argument is similar to the argument in Lemma C.5). Once the cardinality n is computed, we can create (in polynomial time) nodes representing the von Neumann encoding of n.

5. The split operation is again quite similar to the previous ones. This time, however, we need to create multiple duplicates of each pair (τ, γ) which is a successor of the root. The duplicate of (τ, γ) is obtained by replacing the first coordinate τ by the pair (τ, v), for every possible successor v of $\gamma(y)$.

6. The aggregation operation is done in the reverse manner, where all vertices of the form $((\tau, v), \gamma)$, which share the same τ are replaced (as successors of S) by a single vertex (τ, γ'), where γ' is a new vertex with successors $(x, \gamma'(x))$, for each variable x, and $\gamma'(x)$ has as successors all the successors of all elements $\gamma(x)$, for γ such that $((\tau, v), \gamma)$ is a successor of the root of D, unless all the $\gamma(x)$ are the same leaf of D (i.e., represent the same atom), in which case $\gamma'(x)$ is this leaf.

7 and 8 are done by simple manipulations.

All these operations can be performed in polynomial time on the given orbit dag D. □

We now prove Lemma 4.10.

Proof of Lemma 4.10. Let $\mathtt{I}$ be a while program without **while** loops and using n variables. Without loss of generality, we may assume that it is an instruction of the form described in Fact E.1. As in the proof of Theorem 3.7, computing the result of $\mathtt{I}$ on an n-tuple of sets $x_0, \ldots, x_{n-1}$ (represented by expressions) amounts to composing several operations: first compute the superstate $S^{\bar{x}}$, as in item 7 of Lemma E.1, and then perform a sequence of operations as in items 1-6 of the lemma, or unions and emptiness tests, in the case

of *if-then-else* instructions. Finally, perform an operation as in item 8 of the lemma. The total number of operations is bounded by a constant depending on I only. Since fixed-dimension polynomial functions are closed under composition, the lemma follows. □

Example E.2. We show that the emptiness problem for set builder expressions is NP-hard for some effective oligomorphic structures.

We recall the construction of the Rado graph. There are countably many vertices. For each pair of distinct vertices v, w, randomly and independently create an edge between them with probability $\frac{1}{2}$. It is known that with probability 1, the resulting graph is isomorphic to a single graph R, called the Rado graph. The Rado graph satisfies the following extension property: for every two finite, disjoint sets of vertices X, Y, there exists a vertex x which is connected to all vertices in X and to no vertex in Y. Moreover, R is oligomorphic.

We choose one of many known effective representations of R. The vertices V of R are (binary encodings of) natural numbers $1, 2, 3, \ldots$. An edge connects i with j (where $i < j$) if the ith bit of the binary representation of j is 0. The resulting graph (V, E) is isomorphic to the Rado graph.

Let $\mathbb{A}$ be the Rado graph $R = (V, E)$. We now show a reduction from the boolean satisfiability problem to the emptiness problem for hereditarily definable sets of dimension 1.

Let α be the a boolean formula with variables $x_1, \ldots, x_n$. Choose n arbitrary distinct vertices of the random graph, say $v_1, \ldots, v_n$. Define a formula φ over the signature of $\mathbb{A}$ by replacing each variable x_i by the atomic formula $E(t, x_i)$, where t is a variable free in φ. Then the following set X is nonempty if and only if α is satisfiable:

$$X = \{t \text{ for } t \in \mathbb{A} \text{ such that } \varphi\}.$$

Indeed, if X is nonempty, then there is a vertex t such that $\varphi(t)$ holds. Then, by construction, setting the variable x_i to 1 if and only if v_i is a neighbour of t yield a satisfying assignment to α. Conversely, if given a satisfying assignment to α, by the extension property, there exists a vertex t of the random graph which is connected precisely to those vertices v_i, for which the corresponding variable x_i is assigned the value *true*.

This shows that emptiness of expressions of dimension 1 is NP-hard when $\mathbb{A}$ is the Rado graph.

F Proof of Theorem 4.9

We prove Theorem 4.9. Let P be a definable while program (with or without counting), and let M be the definable state machine constructed in Theorem 3.7 (or its extension to while programs with counting).

Recall that for a given input x, the machine M computes a sequence of states $q_0, q_1, \ldots, q_n$ of length polynomial in $\text{time}(\mathsf{P}, x)$, and that each state q_i is a subset of $L(\text{space}(\mathsf{P}, x))$, for some constant time operation L. Let $\bar{a}$ be the least support of the input x. Observe that each state q_i is supported by $\bar{a}$, since the semantics of the definable state machine is invariant under atom automorphisms. Therefore, q_i is a $\bar{a}$-invariant subset of $L(\text{space}(\mathsf{P}, x))$. In particular, $\dim q_i \leq \dim L(\text{space}(\mathsf{P}, x))$ and $\|q_i\| \leq \|L(\text{space}(\mathsf{P}, x))\|$.

We now simulate M by a fixed-dimension polynomial algorithm which inputs an expression α describing a hereditarily definable set x. The algorithm mimics the computation of M, by computing set builder expressions representing the states $q_0, q_1, \ldots, q_n$, and in each step, optimizes the current expression using Lemma 4.7. In particular, by the above observations and by Lemma 4.7, we maintain the invariant that in each step, the expression α describing the current state satisfies $\dim \alpha \leq 2 \dim r$ and $\|\alpha\| \leq f(\dim r, \|r\|)$, where $r = \text{space}(\mathsf{P}, x)$. Therefore, the algorithm performes a polynomial number of constant time operations on expressions satisfying the above inequalities. Hence, its total runtime is $\mathit{poly}(\text{time}(\mathsf{P}, x)) \cdot f(\dim r, \|r\|) \leq g(\dim r, \|r\|)$, for some function g, as required. Moreover, the output $\mathit{Output}(q_n)$ also satisfies the required inequalities.

This finishes the proof of Theorem 4.9.