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 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:

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

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. We are also very grateful to an anonymous referee who observed that our original formulation of Theorem 3.9 can be made more general.

Background. This paper is part of the research programme on computation in *sets with atoms*, whose original motivation was the observation [4, 7] 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 [5].

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 [6] and an imperative programming language [8, 17]. Furthemore, these programming languages have been implemented: the functional programming language as an extension of Haskell [16], and the imperative programming language as a C++ library [17]. In fact, [17] provides more than just a description of an implementation; it also shows how the programming language works for arbitrary logical structures with

*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). The work of Sz. Toruńczyk is supported by the National Science Centre of Poland grant 2016/21/D/ST6/01485.

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209190

a decidable first-order theory, e.g. Presburger Arithmetic, and not just homogeneous ones as assumed in [8].

Our point of departure is the programming language from [8], 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.

In [8, 17] 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 3.9 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 [13]: is there some logic which captures 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 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 polynomial time [3]. 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 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 $(5, 6)$ for $\{\{5, 6\}, 5\}$, 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. by 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 [8, 17]. 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. 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 [17], 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** c **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 $S \rightarrowtail [\![\texttt{I}]\!] \mapsto S'$ 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 $\mathrm{Split}(S, x, e)$ 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 $\mathrm{Split}(S, x, e)$ 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

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

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

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

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.

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 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 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 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 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 P is an instruction 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 $u_1, \ldots, u_n$ 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 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 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 [8] under a stronger assumption that $\mathbb{A}$ is *homogeneous* and effective, and for arbitrary effective atoms in [17], although for a slightly different semantics of while programs. We will show a partial converse to the above theorem in Theorem 3.9.

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 $\mathtt{time}$ and $\mathtt{space}$. The code of P' is the same as P, except that after each instruction we append the following code:

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

The idea is that the $\mathtt{time}$ variable stores an instruction counter represented as a von Neumann integer; and this counter is incremented after each instruction. The $\mathtt{space}$ 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 $\mathtt{time}$ and $\mathtt{space}$. For a while program P with n input variables and an n-tuple $\bar{x}$ of hereditarily definable sets $x_1, \ldots, x_n \in \mathrm{hdef}\mathbb{A}$, 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 $\mathtt{pair}$ which maps a pair of inputs x, y to the Kuratowski encoding of (x, y), that is $\{x, \{x, y\}\}$.
- The reverse operation, $\mathtt{unpair}$ 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 5 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 atoms that are *effectively atomic* structures, as defined below.

An *automorphism of* $\mathbb{A}$ is defined to be any permutation of its universe 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 a first-order formula $\varphi(x_1, \ldots, x_n)$ defines the orbit of $(a_1, \ldots, a_n) \in \mathbb{A}^n$ if the set of n-tuples that satisfy φ in $\mathbb{A}$ is equal to the orbit of $(a_1, \ldots, a_n)$ in $\mathbb{A}^n$. Countable structures with the property that for every tuple $(a_1, \ldots, a_n) \in \mathbb{A}^n$ the orbit of $(a_1, \ldots, a_n)$ is definable by some first-order formula are called *atomic structures*[1] in model theory [15]. Example atomic structures include $(\mathbb{N}, \leq)$ and $(\mathbb{N}, +)$, and all *oligomorphic* structures[2], i.e., structures in which there are only finitely many orbits of n-tuples, for each fixed n, such as $(\mathbb{N}, =)$, $(\mathbb{Q}, \leq)$, or the random graph.

Definition 3.8. A structure $\mathbb{A}$ is *effectively atomic* if it is effective and there is an algorithm which given an n-tuple $(a_1, \ldots, a_n)$ of elements of $\mathbb{A}$ outputs a first-order formula $\varphi(x_1, \ldots, x_n)$ defining the orbit of $(a_1, \ldots, a_n)$.

All the structures mentioned above are effectively atomic.

[1] There is no connection between the use of the term *atomic* here to our notion of *atoms*.

[2] Such structures are also called *ω-categorical*, due to the result of Engeler, Ryll-Nardzewski and Svenonius, cf. [15].

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.

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 3.9 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 3.9 (Computational completeness of definable while programs)**.** *Assume that $\mathbb{A}$ is an effectively atomic structure. 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, effective atomicity is 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. The proof of the theorem is deferred to the full version of the paper.

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. The rough idea why parameters are not counted 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 [9].

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.

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 *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 polynomial-time computation to our setting.

Fact 1. *Assume that the atoms $\mathbb{A}$ are $(\mathbb{N}, =)$. A class L of finite graphs is in 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, defined below. 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. We say that $\mathbb{A}$ admit *least supports* if for every $x \in \mathsf{hdef}\mathbb{A}$ there is a finite set of atoms, called its *least support*, which supports x and is contained in every support of x. Existence of least supports for $(\mathbb{N}, =)$ is shown It is shown in [18] that $(\mathbb{N}, =)$ admit least supports.

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 orbits 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 $\{\{\emptyset\}\}$ 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.

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 deferred to the full version of the paper.

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 $q_n \nsubseteq \text{Bound}(x)$ 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 $(\mathbb{N}, =)$ 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. □

5 Connections to Choiceless Polynomial Time and Abstract State Machines

This section is devoted to a brief discussion of related work, namely Choiceless Polynomial Time (čPT) and Abstract State Machines. We describe how these formalisms compare to ours, on a syntactic level. We observe that čPT is a fragment of definable while programs and observe that a converse implication to Theorem 4.9 would resolve an open problem concerning čPT. We defer to the recent survey [11] for an overview of čPT, and to [2] for a discussion of Abstract State Machines.

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 [19] 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 three comprehension terms, Step, Halt, and Out, and, given an input, proceeds by applying to the current value the term Step until Halt produces *true*, and the produced

output is obtained by applying 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 (or 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 (or čPT+C), which we believe is more programmer-friendly. Furthermore, it provides a new angle at attacking the open problem whether čPT+C captures polynomial time: although Rossman [19] proved that čPT+C cannot define all functions which are polynomial-time computable, for decision problems, the analogous question remains open (cf. Problem 3 in [13]). 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 polynomial time.

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 [14], parallel computation [1], distributed computation [10], quantum computation [12], 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.

[2] Andreas Blass, Yuri Gurevich, and Jan Van den Bussche. 2000. Abstract State Machines and Computationally Complete Query Languages. In *Abstract State Machines - Theory and Applications*, Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–33.

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

[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.

[5] Mikołaj Bojańczyk. 2017. Lecture Notes on Sets with Atoms. (2017). Available at https://www.mimuw.edu.pl/~bojan/.

[6] 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.

[7] 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.

[8] 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.

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

[10] Andreas Glausch and Wolfgang Reisig. 2009. *An ASM-Characterization of a Class of Distributed Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 50–64.

[11] Erich Grädel and Martin Grohe. 2015. *Is Polynomial Time Choiceless?* Springer International Publishing, Cham, 193–209.

[12] 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.

[13] 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.

[14] Yuri Gurevich. 1995. Specification and Validation Methods. Oxford University Press, Inc., New York, NY, USA, Chapter Evolving Algebras 1993: Lipari Guide, 9–36.

[15] Wiflrid Hodges. 1993. *Model Theory*. Cambridge University Press.

[16] 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*. 57–75.

[17] 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.

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

[19] Benjamin Rossman. 2010. *Choiceless Computation and Symmetry*. Springer Berlin Heidelberg, Berlin, Heidelberg, 565–580.