Logspace Reducibility via Abstract State Machines

Erich Grädel and Marc Spielmann

Mathematische Grundlagen der Informatik,
RWTH Aachen, D-52056 Aachen, Germany
{graedel, spielmann}@informatik.rwth-aachen.de

Abstract. We present a notion of logspace reducibility among structures that is based on abstract state machines (ASM). Our reductions are logspace computable (due to the syntactic restrictions we impose on ASM-programs) and are equally expressive as the logic (FO+DTC) enriched with the ability to handle sets. On ordered structures they precisely capture LOGSPACE. Our work continues that of Blass, Gurevich and Shelah on the choiceless fragment of PTIME. Indeed, our reductions can be seen as the choiceless fragment of LOGSPACE.

1 Introduction

Abstract State Machines (ASM) have become a successful methodology for the specification and verification of hardware and software systems. Aside this, ASM provide a computation model which is also very interesting under theoretical aspects. In this paper we study *applications of ASM to the theory of computation.* More precisely, we investigate logspace computability via ASM.

Logspace computability is an important level of complexity, for several reasons:

– It can be viewed as the natural notion of computability with 'very little' memory.
– Logspace computable functions can be computed in parallel very efficiently (i.e., in polylogarithmic time) with a reasonable amount of hardware (i.e., by circuits of polynomial size).
– Logspace reductions are widely accepted as a natural basis for completeness results for important complexity classes like P, NP, and PSPACE. Indeed, most of the complete problems for these classes are complete with respect to logspace reductions [Pap94, GHR95].

The standard computation model in complexity theory is the Turing machine. By definition, a function is logspace computable if it is computable by a Turing machine using on inputs of length n at most $O(\log n)$ cells of its work tapes. However, Turing machines work on strings, whereas many problems arising in computer science and logic have inputs that are naturally viewed as structures rather than strings. Furthermore, in most cases algorithms on structures should treat isomorphic structures in the same way. Although we can encode structures by strings, there is no easily computable string representation of *isomorphism classes* of structures. (This problem was addressed by Chandra and Harel in the context of database queries [CH82]. It disappears when the structures are linearly ordered.). The situation calls for computation models that work directly on structures rather than strings, and in particular, for a notion of *logspace computability on structures* and for a *reduction theory among structures* that is based on such a notion.

There are several computation models on structures in the literature. Some of them have been developed in the context of database theory (see [AHV95]), most notably the *generic machines* of Abiteboul and Vianu. Another model, which we will use in this paper, is provided by *abstract state machines*, formerly known as *evolving algebras* [Gur91, Gur95, Gur97], which has become the foundation of a successful methodology for specification and verification of software and hardware systems [BH98]. One can view a Turing machine program as a description of how to modify the current configuration of a Turing machine to obtain the next configuration. Similarly, an ASM-program describes how to modify the current state of an ASM to obtain the next state. The main difference is that the states of an ASM are mathematical structures rather than strings. Thus, an ASM takes a structure as input and modifies it step by step, until it stops and outputs the resulting structure. (Aside from this basic model, there do exist more general types of ASM, e.g., real-time ASM and recursive ASM, which have been proved useful for specifying and verifying dynamic systems. These will not be considered in the present paper.)

Both generic machines and ASM (and a number of other models as well) are *computationally complete*: they can calculate all computable (isomorphism-invariant) functions on finite structures. Hence, the notion of (isomorphism-invariant) computable functions on structures is well-understood. The situation becomes much more intriguing when we impose complexity bounds. It is not at all clear whether there exists a computation model on structures (for instance a subclass of abstract state machines) that describes precisely the class of all polynomial-time computable functions, or the class of all logspace computable functions on structures. The straightforward approach, namely to impose appropriate time and space restrictions on one of the known (computationally complete) machine model on structures, does not work. The first problem is that honest time measures for such a model are not always obvious and (natural) space measures may not exist at all. But more importantly, the computational completeness of generic machines or ASM does not scale down to lower complexity levels. For instance, the class of functions computable by ASM in honest polynomial time is a strict subclass of the class of isomorphism-invariant functions that are polynomial-time computable in the usual sense, i.e. on string encodings of structures.

The problem of whether there exists a computation model describing precisely the logspace computable isomorphism-invariant functions has been formulated in different terms as the problem of whether there exists a logic for

LOGSPACE, one of the main open problems of finite model theory. This problem has been made precise by Gurevich [Gur88] (for polynomial time rather than logarithmic space, but the situation is similar). Let us briefly review the situation for LOGSPACE.

It is a well-known result due to Immerman [Imm87] that logspace computability is closely related to the logic (FO+DTC), first-order logic (FO) augmented with the deterministic transitive closure operator DTC. A query on ordered finite structures is logspace computable iff it is expressible in (FO+DTC). The DTC operator assigns to any definable binary relation E the transitive closure of the deterministic part E_{det} of E; E_{det} is obtained by removing all edges from E that start at a point with out-degree > 1. An advantage of this description of logspace computability is that once a query is formulated in (FO+DTC), it is guaranteed to be computable in logspace. There is no need to analyze the storage requirements of an algorithm.

On the other side, handling (FO+DTC) is not as straightforward as, say, a programming language and requires a certain familiarity with logic. Moreover, expressions in (FO+DTC) tend to be rather complex and hard to read when describing non-trivial queries. More to the point, if no linear order is available, (FO+DTC) fails to express all logspace queries. In fact, some very simple problems, which are obviously logspace computable, are not expressible in (FO+DTC). A standard example of such a problem, often called PARITY, is the question whether the cardinality of a given structure is even. Gurevich [Gur88] conjectured that there is no logic (and hence no computation model on structures) that captures PTIME or LOGSPACE on arbitrary finite structures.

A natural model-theoretic notion of reductions among structures is provided by (first-order) interpretations (see [Hod93, Chapter 5]). Informally, a structure $\mathcal{A}$ is interpreted in a structure $\mathcal{B}$ by a sequence of first-order formulae that define an isomorphic copy of $\mathcal{A}$ inside $\mathcal{B}$. It is well-known that first-order interpretations are weaker than logspace reductions. One way to enhance their power is to consider (FO+DTC)-interpretations instead. While these still do not capture all logspace computable functions from structures to structures they give us a better lower bound for the desired class of logspace reductions among structures.

After these introductory remarks, we are now in a position to formulate some necessary conditions for the notion of reductions among structures that we want to put forward:

1. Reductions should be computable in logarithmic space.
2. Reasonable closure properties should be satisfied. In particular, the class of reductions should be closed under composition.
3. On ordered structures, our notion should capture all logspace computable functions from structures to structures.
4. On unordered structures, our reductions should have at least the power of (FO+DTC)-interpretations.

Our work is based on a recent paper by Blass, Gurevich, and Shelah [BGS97], who introduced and investigated the *choiceless fragment of* PTIME, denoted $\tilde{\text{C}}$PTIME. An important feature of algorithms on ordered structures (or strings) is the ability to make choices. Out of any set of (otherwise indistinguishable) elements the algorithm can select, say, the smallest, and proceed from there. Typical examples are graph algorithms. Consider, for instance, an algorithm deciding reachability in digraphs. A common way to determine whether a target node t is reachable from a source node s in a given digraph G is to construct the set of all reachable nodes from s. To this end the algorithm maintains an auxiliary set X of 'endpoints'. Initially, $X := \{s\}$. At each iteration step the algorithm 'chooses' a node in X, e.g., by selecting the smallest one, adds its neighbors to the set of reachable nodes, and updates the set of endpoints accordingly. The algorithm terminates once the target node has been found reachable, or the set of reachable points has become stable. It is easy to see that in this example explicit choice is not really needed. Instead of choosing one particular endpoint one can process all of them in parallel. But this seems not always possible. For example, in matching algorithms choice is used in a much more sophisticated way, and there is no known efficient algorithm for perfect matching that avoids explicit choice.

The main idea in [BGS97] is to use hereditarily finite sets and parallel execution to replace explicit choice. Abstract state machines serve as a convenient vehicle to define the choiceless fragment of PTIME in a precise and convincing way. The clean description via ASM makes the class amenable to combinatorial and model-theoretic tools that are used in [BGS97] to determine the power of $\tilde{\mathrm{C}}$PTIME.

In this paper we continue the work of Blass, Gurevich and Shelah. We study reductions among structures that are defined by means of ASM-programs. Motivated by the approach in [BGS97], we equip our model with the ability to handle small sets, which, for instance, serve as a convenient reservoir for new elements. (Notice that while reducing an instance of one problem to an instance of another problem one often has to invent new elements, e.g., new nodes in a graph.) The syntactic restrictions we impose on ASM-programs will guarantee that all functions computed by such programs are logspace computable (by a Turing machine). On ordered structures, moreover, a function is logspace computable iff it can be computed by such a program. Due to the ability to handle sets, our programs are 'choiceless': they can form a set without ever actually choosing one *particular* element of the set. Indeed, the class of properties computable by our programs can be seen as the *choiceless fragment of* LOGSPACE, which we denote $\tilde{\mathrm{C}}$LOGSPACE. Taking the results of [BGS97] into account, we obtain the following relations between standard, choiceless, and descriptive complexity classes:

$$\begin{array}{ccc} \textsc{Logspace} & \subseteq & \textsc{Ptime} \\ \cup^1 & & \cup^1 \\ \tilde{\mathrm{C}}\textsc{Logspace} & \subset^2 & \tilde{\mathrm{C}}\textsc{Ptime} \\ \cup^3 & & \cup^2 \\ (\mathrm{FO+DTC}) & \subset^3 & (\mathrm{FO+LFP}) \end{array}$$

where the inclusions marked 1, 2, and 3 are proper due to the following problems:

(1) PARITY

(2) Small Subset Parity (see Corollary 24)
(3) Reachability in double graphs (see Lemma 11)

We introduce two different variants of ASM-programs to define reductions among structures. Programs of the basic variant, which we call *nullary programs*, are essentially just basic ASM-programs (in the sense of [Gur97]) where every dynamic function is nullary. Nullary programs suffice to capture Logspace on ordered input, but are fairly weak without access to an ordering. Programs of the more powerful variant, called *bounded memory programs*, are sequences of nullary programs where the nullary programs may also occur parameterized by means of `do-forall`-clauses (see [Gur97]). Bounded memory programs have two main advantages over nullary programs. Firstly, even for functions that can be computed by means of nullary programs, bounded memory programs admit presentations that are more succinct and easier to understand. Secondly, bounded memory programs are strictly more expressive than both nullary programs and the logic (FO+DTC).

At this point, the reader may wonder why we use the term "bounded memory programs" rather than, say, "logspace programs". Recall that ASM have been put forward as a model for describing algorithms *on their natural level of abstraction* [Gur95]. The term "logspace" refers to the bit-level. When we describe common logspace algorithms at their natural abstraction level (e.g., by ASM), we see that most of them actually use a bounded number of memory locations (variables) each of which stores an object that is identified by a logarithmic number of bits (e.g., an element of the input structure or a natural number polynomially bounded in the cardinality of the input structure). Hence we find 'bounded memory' programs more adequate for the spirit of ASM.

The contents of the paper is as follows: In Section 2 we recall some terminology from finite model theory and set theory, latter mostly adopted from [BGS97]. In Section 3 we introduce *nullary programs*, our basic variant of programs. In Section 4 we prove that all functions computable by nullary programs are logspace computable. To this end, we describe computations of nullary programs in terms of a set-theoretic extension of the logic (FO+DTC). *Bounded memory programs*, our more powerful variant of programs, are defined in Section 5. In Section 6 we introduce the complexity class $\tilde{\text{C}}$Logspace and show that it is a robust class which also has a logical characterization. Finally, we separate Ptime and Logspace on the choiceless level.

2 Preliminaries

Global relations. Let C be a class of structures over some vocabulary σ. A k-ary *global relation* (or *query*) ρ on C is a mapping that assigns to every structure $\mathcal{A} \in C$ a (local) relation $\rho^{\mathcal{A}} \subseteq A^k$ (where A is the universe of $\mathcal{A}$), such that isomorphisms between structures in C are preserved, i.e., every isomorphism between two structures $\mathcal{A}$ and $\mathcal{B}$ in C is also an isomorphism between $(A, \rho^{\mathcal{A}})$ and $(B, \rho^{\mathcal{B}})$. For example, every first-order formula $\varphi(x_1, \dots, x_k)$ over σ (all of whose free variables occur among $x_1, \dots, x_k$) defines a k-ary global relation on the class of all σ-structures: φ maps a σ-structure $\mathcal{A}$ to $\varphi^{\mathcal{A}} := \{(a_1, \ldots, a_k) \in A^k : \mathcal{A} \models \varphi[a_1, \ldots, a_k]\}$.

Deterministic transitive closure logic. Let FO denote first-order logic with equality. *Deterministic transitive closure logic*, (FO+DTC), is obtained from FO by adding to the usual formula-formation rules of FO the rule

– If φ is a formula, $\bar{x}$ and $\bar{y}$ are two k-tuples of variables, and $\bar{u}$ and $\bar{v}$ are two k-tuples of terms, then $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi](\bar{u}, \bar{v})$ is a formula.

For any $2k$-ary relation R, let $DTC(R)$ denote the transitive reflexive closure of the deterministic part R_{det} of R. (R_{det} is the $2k$-ary relation obtained from R by removing all edges $(\bar{a}, \bar{b}) \in R$ for which there exists an edge $(\bar{a}, \bar{b}') \in R$ such that $\bar{b} \neq \bar{b}'$.) The semantics of $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi](\bar{u}, \bar{v})$ is as follows. Regard $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi]$ as a new $2k$-ary relation symbol whose interpretation is $DTC(R)$, where R is the $2k$-ary relation defined by $\varphi(\bar{x}, \bar{x}')$. As an example, consider a finite digraph $G = (V, E)$ with two distinguished nodes s and t. $G \models [\mathrm{DTC}_{x,y}\ E(x, y)][s, t]$ iff there exists a path in G (composed from E-edges) that starts at node s, ends at node t, and each node on the path (except t) has out-degree 1. For a more formal definition of the semantics of DTC-formulas the reader is referred to [EF95].

Fact 1 ([Imm87]). *A global relation on ordered structures is definable in the logic* (FO+DTC) *iff it is logspace computable.*

Hereditarily finite sets. Let A be a finite set of atoms (i.e., elements that are not sets, also called urelements in set theory). The set $\mathrm{HF}(A)$ of all *hereditarily finite objects* built from the atoms in A is the least set such that

– $A \subseteq \mathrm{HF}(A)$,
– every finite subset of $\mathrm{HF}(A)$ is an element of $\mathrm{HF}(A)$.

Thus, an object $X \in \mathrm{HF}(A)$ is either a finite set or an atom from A. A set X is *transitive*, if whenever $Z \in Y \in X$, then $Z \in X$. The transitive closure $\mathrm{TC}(X)$ of an object X is the least transitive set with $X \in \mathrm{TC}(X)$. For instance, $\mathrm{TC}(A) = A \cup \{A\}$. Observe that for every object $X \in \mathrm{HF}(A)$, $\mathrm{TC}(X)$ is finite and in $\mathrm{HF}(A)$. By the *size* of an object X we mean the cardinality of $\mathrm{TC}(X)$. For example, the size of A is $|A|+1$. Notice that the size of X is a bound for both the cardinality of $A \cap \mathrm{TC}(X)$ and the length k of chains $Y_1 \in Y_2 \in \ldots \in Y_k \in X$. The maximum k is also called the *rank* of X. We write $\mathrm{HF}_s(A)$ for the restriction of $\mathrm{HF}(A)$ to objects of size at most s.

3 Nullary programs

In this section we introduce *nullary programs*, a restricted model for logspace computability on structures. On *ordered* input structures nullary programs suffice to describe all logspace computable functions.

States. Let σ and τ be disjoint finite relational vocabularies. We extend $\sigma \cup \tau$ to a *program vocabulary* $\Upsilon(\sigma, \tau)$ containing, in addition to $\sigma \cup \tau$,

- a unary relation symbol *Universe*,
- the set theoretic symbols $\in, Atoms, unique, \varnothing$, where $\in$ is a binary relation symbol, *Atoms* is a unary relation symbol, *unique* is a unary function symbol, and $\varnothing$ is a constant symbol,
- a number of constant symbols $f_1, \ldots, f_d$.

Our programs will take finite σ-structures as inputs and compute finite τ-structures as outputs. Therefore, we refer to σ as the *input vocabulary* and to τ as the *output vocabulary.* The universe of the output will be determined by the interpretation of the symbol *Universe*, and may grow during the course of a computation by writing to *Universe*. In the following we treat *Universe* as a special kind of output symbol.

Definition 2. Let $\mathcal{A}$ be a finite σ-structure whose universe A is a set of atoms. On input $\mathcal{A}$ every nullary program over $\Upsilon(\sigma, \tau)$ (defined below, see Definition 6) starts its computation in the *initial state* $\mathcal{S}(\mathcal{A})$, which is a $\Upsilon(\sigma, \tau)$-structure defined as follows:

- the universe of $\mathcal{S}(\mathcal{A})$ is $\mathrm{HF}(A)$,
- $\mathcal{A}$ is embedded in $\mathcal{S}(\mathcal{A})$, i.e., $R^{\mathcal{S}(\mathcal{A})} = R^{\mathcal{A}}$ for every input relation $R \in \sigma$,
- the interpretation of $\in$ and $\varnothing$ are the obvious ones; *Atoms* is interpreted as the set A; for every singleton $X \in \mathrm{HF}(A)$, $unique(X)$ is the unique $Y \in X$; for all other $X \in \mathrm{HF}(A)$, $unique(X) := \varnothing$,
- every output relation (including *Universe*) is empty,
- each f_i is interpreted as $\varnothing$.

$\square$

Definition 3. Every *state* $\mathcal{S}$ of a nullary program over $\Upsilon(\sigma, \tau)$ on input $\mathcal{A}$ is a $\Upsilon(\sigma, \tau)$-structure with universe $\mathrm{HF}(A)$. During a computation the universe $\mathrm{HF}(A)$ and the interpretations of the input symbols and the set-theoretic symbols remain unchanged; these symbols are static. In contrast, $f_1, \ldots, f_d$ and all output symbols (including *Universe*) are dynamic in the sense that their interpretation may or may not change from one state to the next. While there are few restrictions on how to change the interpretation of $f_1, \ldots, f_d$, every output relation is write-only. Once a tuple is put into an output relation it remains there for the rest of the computation. We call $f_1, \ldots, f_d$ *dynamic functions.* $\square$

Notice that because tuples cannot be removed from the output, the output is in general relational. Since we will later also consider composition of programs, it is natural to restrict attention to relational input and output vocabularies. However, this restriction has no practical impact. For instance, if you like your favorite function F to be part of the input structure, simply include the graph G_F of F. Your program may then contain a term of the form $unique\{y \in Atoms : G_F(x, y)\}$ whose semantics will be $F(x)$.

The logic (FO+BS). The ASM-programs that we will put forward are based on a restricted variant of first-order definability over states. Below we define a fragment of FO over states which is not as expressive as full FO over states. The fragment is nevertheless strictly more expressive than FO over the input structures, as it can handle 'bounded' sets. To emphasize this, we denote the fragment (FO+BS) where "BS" alludes to "bounded sets".

For any vocabulary σ let $\sigma^+ := \sigma \cup \{\in, \mathit{Atoms}, \mathit{unique}, \varnothing\}$.

Definition 4. The *terms* and *formulae* of (FO+BS) over σ are defined by simultaneous induction:

T1 As usual, terms are built from variables and constants, and are closed under application of function symbols (from σ^+).

T2 If $t_1, \dots, t_k$ are terms and s is a natural number, then $\{t_1, \dots, t_k\}_s$ is a term. ($\{t_1, \dots, t_k\}_s$ denotes the set whose elements are denoted by $t_1, \dots, t_k$, provided that this set has size $\leq s$).

T3 Choose a term t, a variable x, a formula φ, and a natural number s. Let r be either a set symbol in σ^+ or a term with no free occurrence of x. Then $\{t : x \in r : \varphi\}_s$ is a term (denoting the set of all $t(x)$ with x from range r satisfying condition $\varphi(x)$, provided that this set has size $\leq s$).

F Atomic formulae are defined as usual (from $=$ and the relations in σ^+). Formulae are either atomic formulae or built from previously defined formulae by means of negation and disjunction.

The free and bound variables of terms and formulae are defined in the obvious way. In particular, a variable occurs free in $\{t : x \in r : \varphi\}_s$ if it occurs free in t, r or φ and is different from x. x itself occurs bound. $\square$

The semantics of a term t with respect to a state $\mathcal{S}$ with universe $\mathrm{HF}(A)$ is clear in the case where t is of type T1. If $t = \{t_1, \dots, t_k\}_s$ and the set $\{t_1^{\mathcal{S}}, \dots, t_k^{\mathcal{S}}\}$ has size $\leq s$, then $t^{\mathcal{S}}$ is this set. Otherwise, $t^{\mathcal{S}} := \varnothing$. When $t = \{t_0 : x \in r : \varphi\}_s$ and the set $\{t_0^{\mathcal{S}}[X] : X \in r^{\mathcal{S}} : \mathcal{S} \models \varphi[X]\}$ has size $\leq s$, let $t^{\mathcal{S}}$ be this set. Otherwise, $t^{\mathcal{S}} := \varnothing$. The semantics of a formula φ with respect to $\mathcal{S}$ is standard.

We did not mention quantification in the definition of (FO+BS). But note that (FO+BS) can define a guarded form of quantification. For example, let $(\exists x \in r)\varphi$ abbreviate the formula $\varnothing \in \{\varnothing : x \in r : \varphi\}$. (Technically, the set term in this formula needs a subscript to bound its size; 2 will do. To ease notation we frequently omit the subscript at a set term when the set's description implies an obvious bound on its size.) By using *Atoms* for the guard r in $(\exists x \in r)\varphi$, we can simulate in (FO+BS) quantification over elements of the input structure. This implies that, with respect to definability over the input structures, (FO+BS) indeed extends FO.

Lemma 5. *For every* FO*-formula* $\psi(\bar{x})$ *over* σ *there exists a* (FO+BS)*-formula* $\varphi(\bar{x})$ *over* σ *such that for every finite* σ*-structure* $\mathcal{A}$ *and all* $\bar{a} \in \mathcal{A}^k$, $\mathcal{A} \models \psi[\bar{a}]$ *iff* $\mathcal{S}(\mathcal{A}) \models \varphi[\bar{a}]$.

Syntax and semantics of nullary programs.

Definition 6. Fix a program vocabulary $\Upsilon = \Upsilon(\sigma, \tau)$ with dynamic functions $f_1, \dots, f_d$. We define *nullary rules* over Υ inductively:

- **Updates:** For every dynamic function $f \in \Upsilon$, every k-ary output relation $R \in \Upsilon$, and all (FO+BS)-terms $t_0, t_1, \ldots, t_k$ over $\sigma \cup \{f_1, \ldots, f_d\}$ the two assignments $f := t_0$ and $R(t_1, \ldots, t_k) := \mathit{true}$ are (*atomic*) nullary rules. (Frequently, we will abbreviate $R(t_1, \ldots, t_k) := \mathit{true}$ to $R(t_1, \ldots, t_k)$.)
- **Conditional:** If φ is a (FO+BS)-formula over $\sigma \cup \{f_1, \ldots, f_d\}$ and Π a nullary rule, then (`if` φ `then` Π) is a nullary rule (with *guard* φ).
- **Parallel execution:** If Π_0 and Π_1 are nullary rules, then $\frac{\Pi_0}{\Pi_1}$ is a nullary rule (for brevity sometimes written as $\Pi_0 || \Pi_1$).

The free and bound variables of a nullary rule are defined in the obvious way. A *nullary program* is a nullary rule without free variables. $\square$

The semantics of nullary programs is (almost) standard. Consider a nullary program Π over $\Upsilon(\sigma, \tau)$ and let $\mathcal{S}$ be a state of Π. We denote the *sequel of $\mathcal{S}$ with respect to Π* (see [Gur97]) by $\Pi(\mathcal{S})$. The *run of Π on $\mathcal{S}$* is the maximal sequence $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots$ of states such that $\mathcal{S}_0 = \mathcal{S}$, $\mathcal{S}_{i+1} = \Pi(\mathcal{S}_i)$, and $\mathcal{S}_{i+1} \neq \mathcal{S}_i$ for every state $\mathcal{S}_{i+1}$ in the sequence. Note that a run is either finite or infinite, and that a finite run ends with a state $\mathcal{S}_i$ satisfying $\mathcal{S}_i = \Pi(\mathcal{S}_i)$. The run of Π on a finite σ-structure $\mathcal{A}$ is the run of Π on $\mathcal{S}(\mathcal{A})$. In case the latter is finite and ends with a state $\mathcal{S}$, the *output* of Π on $\mathcal{A}$ is, by definition, the τ-reduct of the substructure of $\mathcal{S}$ induced by $\mathit{Universe}^{\mathcal{S}}$. This convention enables us to define the universe of the output by throwing into the unary output relation *Universe* all those objects which we want to be present.

Obviously, the output of Π on $\mathcal{A}$ is—if it exists—a τ-structure. But is it finite? To see this, observe that for every nullary program Π there exists an upper bound s of the size of the objects that Π can touch. ($s \geq 1$ is the maximum of all explicit size bounds of set terms occurring in Π.) Thus, the universe of the output structure is a subset of $\mathrm{HF}_s(A)$. For every finite A, $\mathrm{HF}_s(A)$ is finite.

Note also that the output of a nullary program in general cannot serve as input to another program, because its universe may contain elements that are sets. To avoid problems when composing programs later on we will from now on tacitly assume that the output of a nullary program has its non-atomic elements converted to genuine atoms.

Example 7. Consider the binary global relation ρ_{DTC} which maps every finite ordered digraph $G = (V, E, <)$ to $DTC(E)$. (Recall that $DTC(E)$ denotes the deterministic transitive closure of G's edge relation E.) We present a nullary program Π_{DTC} with input vocabulary $\{E, <\}$ and output vocabulary $\{DTC\}$ that computes ρ_{DTC}. On input G, Π_{DTC} outputs the graph $(V, DTC(E))$.

Let us first concentrate on an instance of the problem: write a nullary program Π which, given a node $\mathit{start\_node} \in V$, outputs all nodes on the deterministic E-path that starts at *start_node*. Here is a possible solution. In the first step, Π initializes a nullary dynamic function *pebble* with *start_node*. Then, in every following step, Π outputs *pebble* and moves *pebble* along the deterministic path by executing the update $\mathit{pebble} := \mathit{succ}(\mathit{pebble})$, where $\mathit{succ}(x)$ abbreviates the term $\mathit{unique}\{y \in \mathit{Atoms} : Exy\}$.

But how do we ensure termination of this process if the path leads into a cycle? Every cycle in G has at most $|V|$ nodes. Thus it suffices to set up a counter which triggers termination after $|V|$ steps. Let *counter* be a nullary dynamic function and let *least* denote the least node in G w.r.t. to the ordering $<$ of the nodes. (Note that *least* is definable as $unique\{x \in Atoms : \neg(\exists y \in Atoms)\, y < x\}$.) Π initializes *counter* with *least* and executes in every step the update $counter := counter + 1$, where $counter + 1$ stands for the term $unique\{x \in Atoms : x > counter \wedge (\forall y \in Atoms)(y > counter \rightarrow y \geq x)\}$.

Π as defined below outputs a pair $(start\_node, a)$ for every node a on the deterministic path that starts at *start_node*. It becomes idle (i.e., it stops) when *pebble* has no unique E-successor or when *counter* assumes $\varnothing$ after $|V|$ steps.

$\Pi :=$ **if** $mode = initialize\_pebble$ **then**
  $pebble := start\_node$
  $counter := least$
  $mode := move\_pebble$
if $mode = move\_pebble$ **then**
  if $pebble \neq \varnothing \wedge counter \neq \varnothing$ **then**
    $DTC(start\_node, pebble)$
    $pebble := succ(pebble)$
    $counter := counter + 1$
  else
    $mode := next\_path$

From Π one easily obtains the desired nullary program Π_{DTC}. Π_{DTC} systematically varies *start_node* over all nodes in V and calls for each instance the above program Π (see line (1) below). When Π terminates, Π_{DTC} resumes its computation in line (2).

$\Pi_{DTC} :=$ **if** $mode = initial$ **then**
  $start\_node := least$
  $mode := initialize\_pebble$
Π (1)
if $mode = next\_path \wedge start\_node \neq \varnothing$ **then** (2)
  $Universe(start\_node)$
  $start\_node := start\_node + 1$
  $mode := initialize\_pebble$

□

It is worth noticing that there is no obvious way to define a nullary program computing $DTC(E)$ without access to an ordering of the nodes. Without such an ordering we can neither count—and this way detect a cycle—nor search systematically through all nodes. The example reveals two defects of nullary programs when no order is available: (1) How can we ensure **termination** of nullary programs? (2) How do we perform **systematic search** of the universe? In Section 5 we are going to cure both defects by upgrading nullary programs to bounded memory programs.

4 Logical description of computations

In this section we prove that nullary programs are logspace computable. To this end we describe computations of nullary programs in terms of (FO+BS+DTC), i.e., the closure of (FO+BS) under (a bounded version of) the DTC-operator. Logspace computability of nullary programs then follows from the observation that every sentence in (FO+BS+DTC) can be evaluated in logarithmic space. On ordered input structures the other direction holds also: every logspace computable function can be computed by a nullary program. Thus, on ordered input nullary programs are as powerful as logspace Turing machines.

In the remainder of this section let σ denote a finite relational vocabulary.

Definition 8. The *hereditarily finite extension* $\mathcal{A}^+$ of a finite σ-structure $\mathcal{A}$ is a σ^+-structure defined as follows:

- the universe of $\mathcal{A}^+$ is $\mathrm{HF}(A)$,
- $R^{\mathcal{A}^+} = R^{\mathcal{A}}$ for every $R \in \sigma$, and
- the interpretation of $\in, Atoms, unique, \varnothing$ are as in Definition 2.

Let $\varphi(x_1, \dots, x_k)$ be a (FO+BS)-formula over σ. The *global relation defined by* φ, also denoted φ, maps the hereditarily finite extension $\mathcal{A}^+$ of a finite σ-structure $\mathcal{A}$ to the (possibly infinite) relation $\varphi^{\mathcal{A}^+} := \{(X_1, \dots, X_k) \in \mathrm{HF}(A)^k : \mathcal{A}^+ \models \varphi[X_1, \dots, X_k]\}$. For any finite σ-structure $\mathcal{A}$, let $\varphi^{\mathcal{A}}$ denote $\varphi^{\mathcal{A}^+}$. $\square$

We extend the logic (FO+BS) to the logic (FO+BS+DTC) by adding a new formula-formation rule to Definition 4:

F2 If s is a natural number, φ is a formula, $\bar{x}$ and $\bar{y}$ are two k-tuples of variables, and $\bar{u}$ and $\bar{v}$ are two k-tuples of terms, then $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi]_s(\bar{u}, \bar{v})$ is a formula.

The semantics of a DTC-formula $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi]_s(\bar{u}, \bar{v})$ is similar to that of a DTC-formula in (FO+DTC), except that now, in order to reach $\bar{v}$ from $\bar{u}$ via a deterministic φ-path, we may compose this path from φ-edges which connect points in $\mathrm{HF}_s(A)^k$ rather than A^k only. (Recall that $\mathrm{HF}_s(A)$ is the set of objects in $\mathrm{HF}(A)$ of size $\leq s$.) More precisely, consider a hereditarily finite extension $\mathcal{A}^+$ of some finite σ-structure $\mathcal{A}$ and interpretations $\bar{X} \in \mathrm{HF}(A)$ of the free variables in $[\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi]_s(\bar{u}, \bar{v})$. Let R_s denote the restriction of $\varphi^{(\mathcal{A}^+, \bar{X})}$ to $\mathrm{HF}_s(A)$, where the $2k$-ary relation $\varphi^{(\mathcal{A}^+, \bar{X})}$ is the image of $(\mathcal{A}^+, \bar{X})$ under the global relation φ. By definition, $(\mathcal{A}^+, \bar{X}) \models [\mathrm{DTC}_{\bar{x},\bar{y}}\ \varphi]_s(\bar{u}, \bar{v})$ iff $(\bar{u}^{(\mathcal{A}^+, \bar{X})}, \bar{v}^{(\mathcal{A}^+, \bar{X})}) \in DTC(R_s)$.

Definition 9. Let C be a class of finite σ-structures and let Π be a nullary program over $\Upsilon(\sigma, \{R\})$ that halts on all $\mathcal{A} \in C$. The *global relation computed by* Π *on* C, also denoted Π, maps every $\mathcal{A} \in C$ to the relation $\Pi^{\mathcal{A}} := R$, where (U, R) is the output of Π on input $\mathcal{A}$. $\square$

Lemma 10. *Every global relation computable by a nullary program is definable in* (FO+BS+DTC).

Lemma 11. *There is a class C of finite graphs and a nullary program Π so that the global relation computed by Π on C is not definable in* (FO+DTC).

Proof. (Sketch.) For every finite digraph $G = (V, E)$ let $2G := (2V, 2E)$ be its doubled version, where $2V := V \times \{0, 1\}$ and $2E := \big\{\big((a, i), (b, j)\big) : (a, b) \in E, 0 \leq i, j \leq 1\big\}$. Consider a graph G consisting of two disjoint (directed) cycles of the same even diameter. Suppose that in only one of the cycles there is a node labeled *start* and a node labeled *goal*. The distance between *start* and *goal* is maximal. Let C_1 be the collection of all $2G$. Now, modify G to G' by moving the label *goal* to the other cycle. Let C_2 be the collection of all $2G'$. There is a path from node $(start, 0)$ to node $(goal, 0)$ in every graph in C_1. No graph in C_2 has this property. Due to Immerman [Imm92] (see also [GM95]) there is no (FO+DTC)-sentence φ such that $2G \models \varphi$ for all $2G \in C_1$ and $2G' \not\models \varphi$ for all $2G' \in C_2$. Nevertheless, the nullary program Π displayed below accepts every graph in C_1 and rejects every graph in C_2. Hence, let $C := C_1 \cup C_2$.

$\Pi :=$ **if** $mode = initial$ **then**
    $pebbles := \{(start, 0)\}_2$
    $mode := move\_pebble$
  if $mode = move\_pebble$ **then**
    if $(goal, 0) \notin pebbles \wedge (start, 1) \notin pebbles$ **then**
      $pebbles := \{x \in Atoms : (\exists y \in pebbles) 2E(y, x)\}_3$
    if $(goal, 0) \in pebbles$ **then** *Accept*
    if $(start, 1) \in pebbles$ **then** *Reject*

□

The last two lemmas imply that (FO+BS+DTC) is more expressive than (FO+DTC). The next lemma shows that (FO+BS+DTC) is not too expressive.

Lemma 12. *Every nullary global relation definable by a sentence in the logic* (FO+BS+DTC) *is logspace computable.*

Logspace computability of every nullary program now follows by Lemma 10. The converse, i.e., that every logspace computable global relation is computable by a nullary program, does not hold in general (see Theorem 22). The situation changes if we restrict attention to ordered structures.

On ordered input structures.

Lemma 13. *Every logspace computable global relation on ordered structures can be computed by a nullary program.*

Proof. (Sketch.) Due to Fact 1 it suffices to show that every global relation definable in (FO+DTC) can be computed by a nullary program. We did most of the work in Example 7. By induction on the construction of a (FO+DTC)-formula $\varphi(\bar{x})$ one can define a nullary rule Π_φ which accepts an input $(\mathcal{A}, \bar{a})$ iff $\mathcal{A} \models \varphi[\bar{a}]$. The desired nullary program computing $\varphi^{\mathcal{A}}$ on input $\mathcal{A}$ then simply runs Π_φ for all possible $\bar{a} \in A^k$ and writes the accepted $\bar{a}$ into an output relation R. The details are left to the reader. □

As pointed out in the discussion following Example 7, there is no obvious way to tell whether a given nullary program Π halts on all structures of a given class C. However, if the input is ordered then one can set up a counter (like *counter* in Example 7) that terminates Π once the maximal number of possible configurations of Π has been reached.

Lemma 14. *Every nullary program can be altered by a syntactic manipulation so that it halts on all ordered inputs and computes exactly the same output as the original program, whenever the latter halts.*

In the next section we will upgrade nullary programs to bounded memory programs. This will improve the handling of programs in practice as well as their expressive power (when no order is present).

5 Bounded memory programs

Nullary programs considered so far do not properly reflect two important properties of logspace computable functions, namely that such functions are closed under composition and 'distributed execution'. Nullary programs cannot simply be composed, because output relations must not occur in guards. But what do we mean with distributed execution? Consider, e.g., a logspace Turing machine $M(x)$ which takes (encodings of) finite structures as input together with a node x of the input structure as parameter. For two different nodes a and b of the input structure the computations of $M(a)$ and $M(b)$ do not interfere with each other. Thus, in order to compute $M(a)$ for every node a of the input structure, we may execute all the instances of $M(x)$ in parallel on distributed processors. This distributed execution is still in (sequential) logspace. We obtain the same result with a logspace Turing machine N that enumerates all nodes a in some order and simulates $M(x)$ with $x = a$.

We add both composition and distributed execution to nullary programs.

Definition 15. Let $\Upsilon^*(\sigma, \tau)$ denote a program vocabulary where σ and τ are not necessarily disjoint, i.e., where some relation symbols may be input and output at the same time. We define *distributed programs* over $\Upsilon^*(\sigma, \tau)$ inductively:

- Every nullary program is a distributed program.
- **Distributed execution:** Let $\bar{x}$ be a tuple of variables and let Π be a nullary rule all of whose free variables but none of whose bounded variables occur among $\bar{x}$. For each x_i in $\bar{x}$ choose a closed range term r_i, i.e., either a closed (FO+BS)-term over σ or a set symbol in σ^+. Let $\Pi_{\bar{x}}$ denote the result of replacing in Π every occurrence of a dynamic function f with the term $f(\bar{x})$. Then (`do forall` $\bar{x} \in \bar{r}\ \ \Pi_{\bar{x}}$) is a distributed program.
- **Guarded distributed execution:** Let $\bar{x}, \bar{r}$, and Π be as in the previous rule. In addition, let $\alpha_1, \ldots, \alpha_n$ be atomic FO-formulae over τ all of whose free variables occur among $\bar{x}$. Suppose that Π satisfies for each α_i, if $\alpha_i = R(\bar{t})$ and $R(\bar{t}')$ is an atomic subrule of Π then $\bar{t} = \bar{t}'$. (This ensures that all updates in Π of an output relation R occurring in some α_i affect only the tuple $\bar{t}$ specified by α_i.) Then the following is a distributed program:

```
do forall x̄ ∈ r̄
unless α1 ∨ ... ∨ αn
    Π_x̄
```

– If Π_0 and Π_1 are distributed programs, then $\Pi_0||\Pi_1$ is a distributed program.

(Observe that a distributed program obtained from a nullary rule Π by applying (guarded) distributed execution may have a different vocabulary than Π. This is because transforming Π to $\Pi_{\bar{x}}$ increases the arity of every dynamic function. Notice also that every distributed program has the form $\Pi_1||\ldots||\Pi_p$, where each Π_i is either a nullary program or a (guarded) distributed execution of a nullary rule. Since $||$ is associative and commutative, we can view every distributed program as a set of programs Π_i.)

Let $\Pi = \{\Pi_1, \ldots, \Pi_p\}$ be a distributed program over some vocabulary $\Upsilon^*(\sigma', \tau')$ and let σ and τ be disjoint finite relational vocabularies. A (σ, τ)-*stratification* of Π is a partition $(\Pi_1^*, \ldots, \Pi_q^*)$ of Π such that

– each Π_i^* is a distributed program over some vocabulary $\Upsilon^*(\sigma_i, \tau_i)$ where σ_i and τ_i are disjoint,
– $\sigma_i = \sigma \cup \tau_1 \cup \ldots \cup \tau_{i-1}$,
– $\tau \subseteq \tau_1 \cup \ldots \cup \tau_q$,
– if $\Pi_k \in \Pi_i^*$ is a guarded distributed execution then $\Pi_i^* = \{\Pi_k\}$.

A *bounded memory program* Π over (σ, τ) is a (σ, τ)-stratified distributed program $(\Pi_1^*, \ldots, \Pi_q^*)$. Each Π_i^* is called a *stratum* of Π. □

Let us first concentrate on bounded memory programs definable without using guarded distributed execution. One can show that every stratum Π_i^* of such a program is equivalent to a distributed program of the form (`do forall` $\bar{x} \in \bar{r}\ \ \Pi_{\bar{x}}$), where $\Pi_{\bar{x}}$ was obtained from some nullary rule Π by replacing in Π every occurrence of a nullary dynamic function symbol f with the term $f(\bar{x})$ (proof omitted). We will often write $f_{\bar{x}}$ instead of $f(\bar{x})$ to indicate that f originated from a nullary function.

Informally, the semantics of a bounded memory program $\Pi = (\Pi_1^*, \ldots, \Pi_q^*)$ is the sequential execution of its strata. Stratum Π_{i+1}^* starts on the halting state of Π_i^* and uses, aside from the input structure, the output relations of all previous strata as input. To get an idea of the semantics of a single stratum Π_i^* assume that

$\Pi_i^* =$ `do forall` $x \in r_x, y \in r_y$
$\qquad\qquad \Pi_{xy}$

The semantics of Π_i^* is the parallel execution of instances of Π_{xy}, where there is one instance Π_{XY} for each pair $(X, Y) \in r_x^{\mathcal{S}} \times r_y^{\mathcal{S}}$. That is, if Π_{XY} denotes the 'nullary program' Π_{xy}, where x and y are interpreted as the objects X and Y, respectively, then one step of Π_i^* can be thought of as one step of $||_{X,Y}\Pi_{XY}$, where X varies in $r_x^{\mathcal{S}}$ and Y varies in $r_y^{\mathcal{S}}$. There is no direct interference between different instances of Π_{xy} because each instance Π_{XY} got its dynamic functions 'tagged' with X, Y. Π_i^* halts when either all instances halt or one instance produces an inconsistency.

To define the semantics of bounded memory programs formally, we first define the semantics of distributed programs. Suppose that Π is a distributed program over $\Upsilon^*(\sigma,\tau)$, where $\Upsilon^*(\sigma,\tau)$ may now contain dynamic function symbols of arity > 0. As in the case of nullary programs, a state $\mathcal{S}$ of Π is a $\Upsilon^*(\sigma,\tau)$-structure with universe $\mathrm{HF}(A)$ for some finite set A of atoms. The sequel $\Pi(\mathcal{S})$ of $\mathcal{S}$ with respect to Π is defined as usual (see [Gur97]). The only new case is when Π was obtained by means of guarded distributed execution.

$$\begin{pmatrix} \texttt{do forall } \bar{x} \in \bar{r} \\ \texttt{unless } \varphi \\ \quad \Pi_{\bar{x}} \end{pmatrix} (\mathcal{S}) \quad := \quad \begin{pmatrix} \texttt{do forall } \bar{x} \in \bar{r} \\ \quad \texttt{if } \neg\varphi \texttt{ then} \\ \quad\quad \Pi_{\bar{x}} \end{pmatrix} (\mathcal{S})$$

Now consider a bounded memory program $\Pi = (\Pi_1^*, \ldots, \Pi_q^*)$ over (σ, τ), where $\Pi_1^* \cup \ldots \cup \Pi_q^*$ is a distributed program over $\Upsilon^*(\sigma', \tau')$. Like a nullary program, Π takes a finite σ-structure as input and, in case it halts, yields a finite τ-structure as output. W.l.o.g., we can assume that $\sigma \subseteq \sigma'$ (since any $R \in \sigma - \sigma'$ does not appear in Π). In the initial state $S(\mathcal{A})$ of Π on an input $\mathcal{A}$ all symbols in $\sigma' - \sigma$ are considered to be output symbols. Hence, $S(\mathcal{A})$ is a $\Upsilon^*(\sigma, \tau' - \sigma)$-structure and is defined as in Definition 2. In order to define the run of Π on $\mathcal{A}$, let us first consider the case $q = 2$, i.e., $\Pi = (\Pi_1^*, \Pi_2^*)$. For simplicity, let us also assume that no dynamic function of stratum Π_1^* occurs in stratum Π_2^*, and vice versa. (This can be achieved by renaming dynamic functions in a suitable way.) The *run of* (Π_1^*, Π_2^*) *on* $\mathcal{A}$ is either

- the infinite run of Π_1 on $\mathcal{S}(\mathcal{A})$, or otherwise
- the composed run $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k, \mathcal{S}_{k+1}, \ldots$, where $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k$ is the finite run of Π_1 on $\mathcal{S}_0 := \mathcal{S}(\mathcal{A})$, and $\mathcal{S}_k, \mathcal{S}_{k+1}, \ldots$ is the run of Π_2 on $\mathcal{S}_k$.

This generalizes to the case $q > 2$ in the obvious way. If the run of Π on $\mathcal{A}$ is finite, then the output of Π is defined as for nullary programs.

Lemma 16. *The class of partial functions computed by bounded memory programs is closed under composition.*

Why use guarded distributed quantification? In the introduction we formulated four necessary conditions for reductions among structures. One of them was that on unordered structures our reductions should have at least the power of (FO+DTC)-interpretations. The question is whether we can compute any (FO+DTC)-definable global relation on unordered structures with a bounded memory program. Recall Example 7 and the subsequent discussion concerning the two shortcomings of nullary programs on unordered structures, namely the problems of *termination* and *systematic search*. We already fixed the problem of systematic search by adding `do-forall` to nullary programs. For example, the following bounded memory program (resembling Π in Example 7) computes $DTC(E)$ on unordered graphs (V, E), although it may not terminate.

$$
\begin{array}{l}
\Pi_{DTC} := \mathtt{do\ forall}\ x \in V \\
\quad \mathtt{if}\ \mathit{mode}_x = \mathit{initial}\ \mathtt{then} \\
\quad\quad \mathit{pebble}_x := x \\
\quad\quad \mathit{mode}_x := \mathit{move\_pebble} \\
\quad \mathtt{if}\ \mathit{mode}_x = \mathit{move\_pebble} \wedge \mathit{pebble}_x \neq \varnothing\ \mathtt{then} \\
\quad\quad DTC(x, \mathit{pebble}_x) \\
\quad\quad \mathit{pebble}_x := \mathit{succ}(\mathit{pebble}_x)
\end{array}
$$

Observe that an instance of the nullary body of Π_{DTC} may run into a cycle, thus prevent Π_{DTC} from halting. It is still not clear how to ensure termination without counting configurations.

Let us modify Π_{DTC} a little bit. The resulting program $\bar{\Pi}_{DTC}$ (see below) is a bounded memory program which uses guarded distributed execution to detect and terminate every instance of the nullary body of Π_{DTC} that 'hangs' in a cycle. In particular, $\bar{\Pi}_{DTC}$ halts on all inputs. We assume that *Cycle* is a new unary output relation.

$$
\begin{array}{l}
\bar{\Pi}_{DTC} := \mathtt{do\ forall}\ x \in V, y \in V \\
\quad \mathtt{unless}\ \mathit{Cycle}(x) \\
\quad\quad \mathtt{if}\ \mathit{mode}_{xy} = \mathit{initial}\ \mathtt{then} \\
\quad\quad\quad \mathit{pebble}_{xy} := x \\
\quad\quad\quad \mathit{mode}_{xy} := \mathit{move\_pebble} \\
\quad\quad \mathtt{if}\ \mathit{mode}_{xy} = \mathit{move\_pebble} \wedge \mathit{pebble}_{xy} \neq \varnothing\ \mathtt{then} \\
\quad\quad\quad DTC(x, \mathit{pebble}_{xy}) \\
\quad\quad\quad \mathit{pebble}_{xy} := \mathit{succ}(\mathit{pebble}_{xy}) \\
\quad\quad \mathtt{if}\ \mathit{pebble}_{xy} = y \wedge \neg \mathit{reached}_{xy}\ \mathtt{then}\ \mathit{reached}_{xy} := \mathit{true} \\
\quad\quad \mathtt{if}\ \mathit{pebble}_{xy} = y \wedge \ \ \mathit{reached}_{xy}\ \mathtt{then}\ \mathit{Cycle}(x)
\end{array}
$$

Let Π_{xy} denote the nullary body of $\bar{\Pi}_{DTC}$. The new guard "`unless` *Cycle*(x)" for Π_{xy} in $\bar{\Pi}_{DTC}$ ensures that only those instances of Π_{xy} contribute in the next computation step, for which *Cycle*(x) does **not** hold. All other instances of Π_{xy} are *disabled*. Here is the idea behind $\bar{\Pi}_{DTC}$. Fix a node $a \in V$ and concentrate on the deterministic E-path starting at a. We run Π_{ab} for every node $b \in V$ in parallel. Each b can be seen as a probe. When pebble_{ab} is placed on b the first time, we set a dynamic function $\mathit{reached}_{ab}$ to *true*, indicating that b has been touched once. If b is pebbled a second time we know that the deterministic path starting at a leads to a cycle through b. In that case there will be no further new output to $DTC(a, y)$, so that we can stop all Π_{ac} whose first subscript a is the same as that of Π_{ab}, which detects the cycle. As a stop signal for each Π_{ac} we set *Cycle*(a) to *true*. On the other hand, if there is no b such that Π_{ab} places pebble_{ab} on b twice, then the deterministic path starting at a does not lead to a cycle. All Π_{ac} will come to a halt simultaneously when the path ends.

The cycle detection technique described above is also the main crux in the proof of the following lemma.

Lemma 17. *Every global relation definable in* (FO+DTC) *is computable by a bounded memory program.*

6 Choiceless Logspace

In this section we define Choiceless Logarithmic Space ($\tilde{\text{C}}$LOGSPACE) as the class of graph properties decidable by means of bounded memory programs. $\tilde{\text{C}}$LOGSPACE is a fragment of all logspace computable graph properties and can be seen as the LOGSPACE counterpart of Choiceless Polynomial Time ($\tilde{\text{C}}$PTIME), the choiceless fragment of all polynomial-time computable graph properties. $\tilde{\text{C}}$PTIME was recently defined by Blass, Gurevich, and Shelah [BGS97] using polynomial-time bounded ASM-programs. Our programs are 'choiceless' in the sense that they can form and handle bounded sets of objects (like nodes in an input graph) without ever actually choosing one particular element of a set. Essentially this capability makes bounded memory programs more expressive than the logic (FO+DTC) (cf. the proof of Lemma 11). We conclude the section with the observation that bounded memory programs are not as powerful as the polynomial-time bounded programs of [BGS97]. This separates PTIME and LOGSPACE on the choiceless level.

There are two points to clarify in order to make the upcoming definition of $\tilde{\text{C}}$LOGSPACE reasonable:

- $\tilde{\text{C}}$LOGSPACE is supposed to be fragment of LOGSPACE. Thus we have to show that every bounded memory program actually is logspace computable.
- We cannot simply define $\tilde{\text{C}}$LOGSPACE by means of *all* bounded memory programs (which use, e.g., a Boolean output constants *Accept* to indicate acceptance), since some of these programs may not halt on all inputs and thus do not decide their input. The question is (still) how to ensure termination of bounded memory programs?

We will first solve the problem of termination by describing a syntactic manipulation of arbitrary bounded memory programs, so that the resulting "standard" bounded memory programs halt on all inputs.

Termination of bounded memory programs. We lift the cycle-detection construction leading from Π_{DTC} to $\bar{\Pi}_{DTC}$ in the previous section to arbitrary strata of the form

$$
\begin{aligned}
\Pi^* := {}& \texttt{do forall } \bar{x} \in \bar{r} \\
& \quad \texttt{unless } \varphi \\
& \qquad \Pi_{\bar{x}}
\end{aligned}
$$

where $\Pi_{\bar{x}}$ was obtained from some nullary rule Π. (On can show that every stratum of a bounded memory program can be replaced with a stratum of that form.) Let $f_1, \dots, f_d$ be an enumeration of the dynamic functions in Π. Consider an instance $\Pi_{\bar{X}}$ of $\Pi_{\bar{x}}$. The sequel of a state $(\mathcal{S}, \bar{X})$ of $\Pi_{\bar{X}}$ is entirely determined by the input embedded in $\mathcal{S}$ and the *dynamic part* $f_1^{\mathcal{S}}(\bar{X}), \dots, f_d^{\mathcal{S}}(\bar{X})$ of $(\mathcal{S}, \bar{X})$. We can detect a cycle in the computation of $\Pi_{\bar{X}}$ if we run $\Pi_{\bar{X}}$ in parallel for every possible dynamic part, and stop the computation once we pass the same dynamic part twice. That is, for all possible values $Y_1, \dots, Y_d$ of $f_1^{\mathcal{S}}(\bar{X}), \dots, f_d^{\mathcal{S}}(\bar{X})$ in all possible states of $\Pi_{\bar{X}}$ we run an instance of $\Pi_{\bar{X},y_1,\dots,y_d}$ in parallel. ($\Pi_{\bar{x},y_1,\dots,y_d}$ is obtained from $\Pi_{\bar{x}}$ by replacing every occurrence of a $f_i(\bar{x})$ with $f_i(\bar{x},y_1,\dots,y_d)$.) If one of these instances, say, $\Pi_{\bar{X},Y_1,\dots,Y_d}$, finds for the second time that the current values of $f_1,\dots,f_d$ match its private values $Y_1,\dots,Y_d$, then $\Pi_{\bar{X},Y_1,\dots,Y_d}$ may stop all other instances of $\Pi_{\bar{X},y_1,\dots,y_d}$. Here is the standard form of stratum Π^* that detects repeating configurations and thus terminates on all inputs:

$$
\begin{array}{rl}
\bar{\Pi}^* := & \texttt{do forall } \bar{x} \in \bar{r}, y_1,\dots,y_d \in t \\
& \quad \texttt{unless } \varphi \vee \mathit{Cycle}(\bar{x}) \\
& \quad\quad \Pi_{\bar{x}\bar{y}} \\
& \quad\quad \texttt{if } \bigwedge_i f_{i\bar{x}\bar{y}} = y_i \wedge \neg \mathit{reached}_{\bar{x}\bar{y}} \texttt{ then } \mathit{reached}_{\bar{x}\bar{y}} := \mathit{true} \\
& \quad\quad \texttt{if } \bigwedge_i f_{i\bar{x}\bar{y}} = y_i \wedge \ \ \mathit{reached}_{\bar{x}\bar{y}} \texttt{ then } \mathit{Cycle}(\bar{x})
\end{array}
$$

where the range term t is such that $f_1^{\mathcal{S}}(\bar{X}),\dots,f_d^{\mathcal{S}}(\bar{X}) \in t^{\mathcal{S}}$ for all possible states $(\mathcal{S},\bar{X})$ of $\Pi_{\bar{X}}$ (on any input).

Unfortunately, we may not be able to find such a range term t for every stratum Π^*. This is because the range of a dynamic function f_i can in general be any subset of $\mathrm{HF}_s(A)$, where $s \geq 1$ is the maximum of all explicit size bounds of set terms occurring in Π^*. For instance, the range of f_i might be $\{\{a\} : a \in A\} \subseteq \mathrm{HF}_2(A)$ and it is easy to see that there is no (FO+BS)-term denoting (a superset of) $\{\{a\} : a \in A\}$. The next lemma provides a way out of this dilemma.

Lemma 18. *For every $s \geq 1$ there exist* (FO+BS)*-terms $\mathit{object}_s(x,y_1,\dots,y_s)$ and Forms_s over $\{\varnothing\}$ such that for every state $\mathcal{S}$ with universe* $\mathrm{HF}(A)$*,* $\mathrm{HF}_s(A) = \{\mathit{object}_s(F,\bar{m})^{\mathcal{S}} : F \in \mathit{Forms}_s^{\mathcal{S}}, \bar{m} \in A^s\}$.

Proceeding toward a standard form of stratum Π^* that does not depend on the existence of a range term t, modify the definition of $\bar{\Pi}^*$ as follows: (1) Replace "$y_1,\dots,y_d \in t$" with "$F_1,\dots,F_d \in \mathit{Forms}_s, \bar{m}_1,\dots,\bar{m}_d \in \mathit{Atoms}$", where each F_i is a new variable and each $\bar{m}_i$ is a tuple of s new variables. (2) Replace "$\bigwedge_i f_{i\bar{x}\bar{y}} = y_i$" with "$\bigwedge_i f_{i\bar{x}\bar{y}} = \mathit{object}_s(F_i,\bar{m}_i)$", where $\bar{y}$ now abbreviates the tuple $F_1,\dots,F_d,\bar{m}_1,\dots,\bar{m}_d$ of variables. The newly defined $\bar{\Pi}^*$ halts on all inputs and computes the same output as Π^*.

Lemma 19. *Every bounded memory program can be altered by a syntactic manipulation so that it halts on all inputs and computes exactly the same output as the original program, whenever the latter halts.*

Using Lemma 18 we can also show the next theorem, which in turn implies logspace computability of bounded memory programs (recall Lemma 12).

Theorem 20. *A nullary global relation is definable by a sentence in the logic* (FO+BS+DTC) *iff it is computable by a bounded memory program.*

Choiceless Logarithmic Space. We call bounded memory programs altered according to Lemma 19 *standard.* A *bounded memory acceptor Π* is a standard bounded memory program over $(\sigma,\{\mathit{Accept}\})$. *$\Pi$ accepts* a finite σ-structure $\mathcal{A}$ if Π outputs *Accept* on input $\mathcal{A}$. Otherwise *Π rejects $\mathcal{A}$.*

Definition 21. A class C of finite σ-structures is in $\tilde{\mathrm{C}}$LOGSPACE iff it is closed under isomorphisms and there exists a bounded memory acceptor Π such that Π accepts every structure in C and rejects every finite σ-structure not in C. $\square$

The following theorem summarizes our main results of last three sections (taking into account the main result of [BGS97]).

Theorem 22. (FO+DTC) $\subsetneq$ (FO+BS+DTC) $=$ $\tilde{\mathrm{C}}$LOGSPACE $\subsetneq$ LOGSPACE.

We conclude this section by showing that the same problem which separates $\tilde{\mathrm{C}}$PTIME and (FO+LFP) also separates $\tilde{\mathrm{C}}$PTIME and $\tilde{\mathrm{C}}$LOGSPACE. This problem, which we call SMALL SUBSET PARITY, is the following one. Given a finite structure $\mathcal{A} = (A, S)$ with S a subset of A such that $|S|! \leq |A|$, decide whether $|S|$ is even. It is easy to see that SMALL SUBSET PARITY is in $\tilde{\mathrm{C}}$PTIME [BGS97]. Here we prove that SMALL SUBSET PARITY is not in $\tilde{\mathrm{C}}$LOGSPACE, implying that $\tilde{\mathrm{C}}$LOGSPACE is a proper subclass of $\tilde{\mathrm{C}}$PTIME.

First, let us recall some definitions from [BGS97]. Consider a finite relational σ-structure $\mathcal{A}$ whose universe A consists of atoms.

- A set X of atoms is called a *support* of an object $Y \in \mathrm{HF}(A)$, if every automorphism of $\mathcal{A}$ that pointwise fixes X also fixes Y. For example, $A_Y := A \cap \mathrm{TC}(Y)$ is the *trivial support* of Y.
- Let $k \geq 1$ be a natural number. Call an object $Y \in \mathrm{HF}(A)$ *k-symmetric*, if every $Z \in \mathrm{TC}(Y)$ has a support of cardinality $\leq k$. Obviously, any Y is $|A_Y|$-symmetric.
- Let $\bar{\mathcal{A}}$ denote the $(\sigma \cup \{\in, \varnothing\})$-reduct of $\mathcal{A}^+$ (see Definition 8) and $\bar{\mathcal{A}}_k$ the restriction of $\bar{\mathcal{A}}$ to all k-symmetric objects.

Suppose that σ contains only unary relation symbols, say, $P_1, \ldots, P_c$. A finite σ-structure $\mathcal{A}$ is called a *colored set*, if the colors $P_1, \ldots, P_c$ partition the universe of $\mathcal{A}$.

Fact 23 ([BGS97]). *Fix some positive integers c, k, m. If $\mathcal{A}$ and $\mathcal{B}$ are colored sets, in each of which all the colors $P_1, \ldots, P_c$ are sufficiently large, then $\bar{\mathcal{A}}_k$ and $\bar{\mathcal{B}}_k$ are $L^m_{\infty,\omega}$-equivalent.*

Corollary 24. SMALL SUBSET PARITY $\notin$ $\tilde{\mathrm{C}}$LOGSPACE.

Proof. (Sketch.) Towards a contradiction assume that there is a sentence φ in (FO+BS+DTC) over vocabulary $\{S\}$ which defines SMALL SUBSET PARITY. That is, for every finite structure $\mathcal{A} = (A, S)$ satisfying $|S|! \leq |A|$, $\mathcal{A}^+ \models \varphi$ iff $|S|$ is even. From φ we can extract a size bound s, such that the above relation still holds if we restrict $\mathcal{A}^+$ to any $U \supseteq \mathrm{HF}_s(A)$. φ can be translated into an equivalent $\mathrm{L}^m_{\infty,\omega}$-sentence φ' over $\{S, \in \varnothing\}$ (for some m) by unfolding DTC-subformulae. If $|S|! \leq |A|$ and $U \supseteq \mathrm{HF}_s(A)$, then $(U, S, \in, \varnothing) \models \varphi'$ iff $|S|$ is even. Now choose a positive instance $\mathcal{A}$ and a negative instance $\mathcal{B}$ of SMALL SUBSET PARITY, in each of which the colors S and S^C (the complement of S) are so large, such that $\bar{\mathcal{A}}_s$ and $\bar{\mathcal{B}}_s$ are $\mathrm{L}^m_{\infty,\omega}$-equivalent. Since for every object $Y \in \mathrm{HF}_s(A)$

the trivial support A_Y has cardinality at most s, every such Y is s-symmetric. Hence, $\mathrm{HF}_s(A)$ is a subset of the set of all s-symmetric objects and $\bar{\mathcal{A}}_s \models \varphi'$. A similar argument shows $\bar{\mathcal{B}}_s \not\models \varphi'$. But this contradicts $\varphi' \in \mathrm{L}^m_{\infty,\omega}$. $\square$

Acknowledgements. We are grateful to Andreas Blass, Yuri Gurevich and Eric Rosen for numerous discussions and suggestions. In particular Yuri suggested the development of a logspace reduction theory among structures and proposed nullary ASM-programs as an initial model. Yuri's persistent propaganda for the ASM-model kept our interest alive.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addision-Wesley, 1995.

[BGS97] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. Technical Report CSE-TR-338-97, University of Michigan, May 1997.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of the EATCS*, 64:105–127, February 1998.

[CH82] A. Chandra and D. Harel. Structure and Compexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[EF95] H. D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, 1995.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation – P-Completeness Theory.* Oxford University Press, 1995.

[GM95] E. Grädel and G. McColm. On the Power of Deterministic Transitive Closures. *Information and Computation*, 119:129–135, 1995. See also: Deterministic versus Nondeterministic Transitive Closure Logic, in *Proceedings of 7th IEEE Symposium on Logic in Computer Science (LICS '92)*, 58–63.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving Algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, 1991. a slightly revised version in G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292, World Scientific, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan, May 1997.

[Hod93] W. Hodges. *Model Theory.* Cambridge University Press, 1993.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[Imm92] N. Immerman. Personal communication, 1992.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.