Choiceless Polynomial Time Computation and the Zero-One Law

Andreas Blass[1]* and Yuri Gurevich[2]

[1] Mathematics Dept., University of Michigan, Ann Arbor, MI 48109–1109, USA
`ablass@umich.edu`
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
`gurevich@microsoft.com`

1 Introduction

This paper is a sequel to [2], a commentary on [7], and an abridged version of a planned paper that will contain complete proofs of all the results presented here.

The BGS model of computation was defined in [2] with the intention of modeling computation with arbitrary finite relational structures as inputs, with essentially arbitrary data structures, with parallelism, but without arbitrary choices. In the absence of any resource bounds, the lack of arbitrary choices makes no difference, because an algorithm could take advantage of parallelism to produce all possible linear orderings of its input and then use each of these orderings to make whatever choices are needed. But if we require the total computation time (summed over all parallel subprocesses) to be polynomially bounded, then there isn't time to construct all the linear orderings, and so the inability to make arbitrary choices really matters.

In fact, it was shown that choiceless polynomial time $\tilde{\mathrm{C}}\mathrm{PTime}$, the complexity class defined by BGS programs subject to a polynomial time bound, does not contain the parity problem: Given a set, determine whether its cardinality is even. Several similar results were proved, all depending on symmetry considerations, i.e., on automorphisms of the input structure.

Subsequently, Shelah [7] proved a zero-one law for $\tilde{\mathrm{C}}\mathrm{PTime}$ properties of graphs. We shall state this law and discuss its proof later in this paper. For now, let us just mention a crucial difference from the earlier results in [2]: Almost all finite graphs have no non-trivial automorphisms, so symmetry considerations cannot be applied to them. Shelah's proof therefore depends on a more subtle concept of partial symmetry, which we explain in Section 8 below.

Finding the proof in (an early version of) [7] difficult to follow, we worked out a presentation of the argument for the main case, which we hope will be helpful for others interested in Shelah's ideas. We also added some related results, indicating the need for certain aspects of the proof and clarifying some of the concepts involved in it. Unfortunately, this material is not yet fully written up.

* Preparation of this paper was partially supported by a grant from Microsoft Corporation.

The part already written, however, exceeds the space available to us in the present volume. We therefore present here an abridged version of that paper and promise to make the complete version available soon.

For simplicity, we shall often deal only with input structures that are undirected, loopless graphs, i.e., sets equipped with a symmetric, irreflexive binary relation of adjacency. We also restrict our attention to the uniform probability model. That is, we define the probability of a property (assumed isomorphism-invariant) of n-vertex graphs by considering all graphs with vertex set $\{1, 2, \dots, n\}$ to be equally probable. The *asymptotic probability* of a property of graphs is defined as the limit, as $n \to \infty$, of its probability among n-vertex graphs. In general, a zero-one law says that properties have asymptotic probability 0 or 1, but, as we shall see, some care is needed in formulating the zero-one law for $\tilde{\mathrm{C}}$PTime.

All the results discussed in this paper can be routinely extended to other contexts, such as directed graphs, or sets with several relations, including relations of more than two arguments. It is also routine to replace the uniform probability measure by one where all potential edges have probability p, a constant other than $\frac{1}{2}$. We do not discuss these generalizations further, because they complicate the notation without contributing any new ideas.

2 The Zero-One Law

We start with a very brief description of the BGS model of computation, just adequate to formulate the zero-one law. In Section 3, we shall give more details about the model, in preparation for a description of its proof.

The BGS model, introduced in [2], is a version of the abstract state machine (ASM) paradigm [6]. The input to a computation is a finite relational structure I. A state of the computation is a structure whose domain is $\mathrm{HF}(I)$, which consists of the domain of I together with all hereditarily finite sets over it; the structure has the relations of I, some set-theoretical apparatus (for example the membership relation $\in$), and some dynamic functions. The computation proceeds in stages, always modifying the dynamic functions in accordance with the program of the computation. The dynamic functions are initially constant with value $\emptyset$ and they change at only finitely many arguments at each step. So, although $\mathrm{HF}(I)$ is infinite, only a finite part of it is involved in the computation at any stage. The computation ends when and if a specific dynamic 0-ary function Halt acquires the value `true`, and the result of the computation is then the value of another dynamic 0-ary function Output.

This model was used to define choiceless polynomial time $\tilde{\mathrm{C}}$PTime by requiring a computation to take only polynomially many (relative to the size of the input structure I) steps and to have only polynomially many active elements. (Roughly speaking, an element of $\mathrm{HF}(I)$ is active if it participates in the updating of some dynamic function at some stage.) Also, Output was restricted to have Boolean values, so the result of a computation could only be true, or false, or undecided. (The "undecided" situation arises if the computation exhausts the allowed number of steps or the allowed number of active elements without Halt becoming true.) We shall use the name *polynomial time BGS program* to refer to a BGS program, with Boolean Output, together with polynomial bounds on the number of steps and the number of active elements.

Two classes $\mathcal{K}_0$ and $\mathcal{K}_1$ of graphs are $\tilde{\text{C}}$PTime-*separable* if there is a polynomial time BGS program Π such that, for all input structures from $\mathcal{K}_0$ (resp. $\mathcal{K}_1$), Π halts with output `false` (resp. `true`) without exceeding the polynomial bounds. It doesn't matter what Π does when the input is in neither $\mathcal{K}_0$ nor $\mathcal{K}_1$.

Theorem 1 (Shelah's Zero-One Law) *If $\mathcal{K}_0$ and $\mathcal{K}_1$ are $\tilde{C}$PTime-separable classes of undirected graphs, then at least one of $\mathcal{K}_0$ and $\mathcal{K}_1$ has asymptotic probability zero.*

An equivalent formulation of this is that, for any given polynomial time BGS program, either almost all graphs produce output true or undecided or else almost all graphs produce output false or undecided. It is tempting to assert the stronger claim that either almost all graphs produce true, or almost all produce false, or almost all produce undecided. Unfortunately, this stronger claim is false; a counterexample will be given after we review the definition of BGS programs in Section 3.

The theorem was, however, strengthened considerably in another direction in [7]. It turns out that the number of steps in a halting computation is almost independent of the input.

Theorem 2 *Let a BGS program Π with Boolean output and a polynomial bound for the number of active elements be given. There exist a number m, an output value v, and a class $\mathcal{C}$ of undirected graphs, such that $\mathcal{C}$ has asymptotic probability one and such that, for each input $I \in \mathcal{C}$, one of the two following alternatives holds. Either Π on input I halts after exactly m steps with output value v and without exceeding the given bound on active elements, or Π on input I exceeds the bound on active elements by step m.*

Notice that this theorem does not assume a polynomial bound on the number of steps. It is part of the conclusion that the number of steps is not only polynomially bounded but constant as long as the input is in $\mathcal{C}$ and the number of active elements obeys its bound.

Intuitively, bounding the number of active elements, without bounding the number of computation steps, amounts to a restriction on space, rather than time. Thus, Theorem 2 can be viewed as a zero-one law for choiceless polynomial space computation.

The class $\mathcal{C}$ in the theorem actually has a fairly simple description; it consists of the graphs that have at least n_1 nodes and satisfy the strong extension axioms to be defined in Section 7 below for up to n_2 variables. The parameters n_1 and n_2 in this definition can be easily computed when the program Π and the polynomial bound on the number of active elements are specified.

3 BGS Programs

In this section, we review the syntax and semantics of BGS programs, as well as the concept of active elements. These are the ingredients used in defining $\tilde{\text{C}}$PTime in [2].

We identify the truth values `false` and `true` with the sets $0 = \emptyset$ and $1 = \{0\}$, respectively. Thus, relations can be regarded as functions taking values in $\{0, 1\}$.

Definition 3 Our *function symbols* are

- the logical symbols, namely $=$ and the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, `true`, and `false`,
- the set-theoretic function symbols $\in$, $\emptyset$, Atoms, $\bigcup$, TheUnique, and Pair,
- the input predicate symbol A, and
- finitely many dynamic function symbols.

The intended interpretation of $\bigcup x$, where x is a family of sets and atoms, is the union of the sets in x (ignoring the atoms). If x is a set with exactly one member then TheUnique(x) is that member. Pair(x, y) means $\{x, y\}$. The input predicate A denotes the adjacency relation of the input graph. The intended meanings (and arities) of the other symbols should be clear. We adopt the convention that if a function is applied to an inappropriate argument (like $\bigcup$ applied to an atom or A applied to sets) then the value is $\emptyset$.

The function symbols $\in$, A, and the logical symbols are called *predicates* because their intended values are only `true` and `false`.

In addition to function symbols, we use a countably infinite supply of variables and we use certain symbols introduced in the following definitions of terms and rules.

Definition 4 *Terms* and *Boolean terms* are defined recursively as follows.

- Every variable is a term.
- If f is a j-ary function symbol and $t_1, \ldots, t_j$ are terms, then $f(t_1, \ldots, t_j)$ is a term. It is Boolean if f is a predicate.
- If v is a variable, $t(v)$ a term, r a term in which v is not free, and $\varphi(v)$ a Boolean term, then
$$\{t(v) : v \in r : \varphi(v)\}$$
is a term.

The construction $\{t(v) : v \in r : \varphi(v)\}$ binds the variable v.

In connection with $\{t(v) : v \in r : \varphi(v)\}$, we remark that, by exhibiting the variable v in $t(v)$ and $\varphi(v)$, we do not mean to imply that v must actually occur there, nor do we mean that other variables cannot occur there. We are merely indicating the places where v could occur free. The "two-colon" notation $\{t(v) : v \in r : \varphi(v)\}$ is intended to be synonymous with the more familiar "one-colon" notation $\{t(v) : v \in r \wedge \varphi(v)\}$. By separating the $v \in r$ part from $\varphi(v)$, we indicate the computational intention that the set should be built by running through all members of r, testing for each one whether it satisfies φ, and collecting the appropriate values of t. Thus $\{t(v) : v \in r : v \in r'\}$ and $\{t(v) : v \in r' : v \in r\}$ are the same set, but produced in different ways. (Such "implementation details" have no bearing on the results but provide useful intuitive background for some of our definitions.)

Definition 5 *Rules* are defined recursively as follows.

- `Skip` is a rule.
- If f is a dynamic j-ary function symbol and $t_0, t_1, \ldots, t_j$ are terms, then

$$f(t_1, \ldots, t_j) := t_0$$

  is a rule, called an *update rule.*
- If φ is a Boolean term and R_0 and R_1 are rules, then

$$\texttt{if } \varphi \texttt{ then } R_0 \texttt{ else } R_1 \texttt{ endif}$$

  is a rule, called a *conditional rule.*
- If v is a variable, r is a term in which v is not free, and $R(v)$ is a rule, then

$$\texttt{do forall } v \in r,\ R(v) \texttt{ enddo}$$

  is a rule, called a *parallel combination.*

The construct `do forall` $v \in r,\ R(v)$ `enddo` binds the variable v.

Convention 6 When the "else" part is `Skip`, we use `if` φ `then` R to abbreviate `if` φ `then` R `else Skip endif`. We use `do in parallel` R_0, R_1`enddo` as an abbreviation for

```
do forall $v \in$ Pair(true, false)
   if $v$ =true then $R_0$ else $R_1$
   endif
enddo
```

The `do in parallel` construct applied to more than two rules means an iteration of the binary `do in parallel` .

Definition 7 A *program* is a rule with no free variables.

Convention 8 By renaming bound variables if necessary, we assume that no variable occurs both bound and free, and no variable is bound twice, in any term or rule.

Throughout much of this paper, the context of our discussion will include a fixed program Π. In such situations, we adopt the following convention.

Convention 9 When we refer to a term or rule within Π, we mean a specific occurrence of the term or rule in Π.

Since a program Π has no free variables, every variable v occurring in it is bound exactly once, either by a term $\{t : v \in r : \varphi\}$ or by a rule `do forall` $v \in r,\ R$ `enddo`.

Definition 10 If v is bound by $\{t : v \in r : \varphi\}$, then the *scope* of v consists of the exhibited occurrence of v as well as t and φ. If v is bound by `do forall` $v \in r,\ R$ `enddo`, then the *scope* of v consists of its exhibited occurrence and R. In both cases, the *range* of v is r. Notice that the range of v is not in the scope of v.

We shall need to know that the semantics of terms and rules can be defined by first-order formulas using only primitive set-theoretic notions ($\in$ and Atoms), the adjacency relation of the input graph, and the dynamic functions. More precisely, consider any particular state of one of our ASM's. It is a structure H^+ with underlying set $HF(I)$ and with interpretations for all the function symbols listed in Definition 3. Let H be the structure

$$H = \langle HF(I), \in, I, A \rangle$$

that is like H except that among the non-logical symbols only $\in$, Atoms, and A are interpreted. (In the usual terminology of mathematical logic, H is a reduct of H^+.) Let H^D be the intermediate structure in which the dynamic function symbols are also interpreted (by the same functions as in H^+). We shall need to know that all essential aspects of the execution of Π in the state H^+, i.e., the computation leading from H^+ to its sequel, can be defined in the structure H^D. (It will turn out that, for every state H^+ that actually arises during the computation of a BGS machine, the dynamic functions will be definable in H, and so we could use H instead of H^D here. See the proof of Proposition 11 below.)

The longer version of this paper will contain these H^D-definitions in full; here we give only some typical examples. To avoid having to introduce new symbols for a multitude of set-theoretic formulas, we adopt the notational convention that $\lceil\varphi\rceil$ means the set-theoretic formalization of the (informal) statement φ.

We first define $\lceil y \in t\rceil$ and $\lceil y = t\rceil$ for all terms t; here y is a variable not free in t. Here are some typical clauses from the definition.

– If f is a dynamic function symbol, then $\lceil y = f(t_1, \dots, t_j)\rceil$ is

$$\exists z_1 \dots \exists z_j \left(\bigwedge_{i=1}^{j} \lceil z_i = t_i \rceil \wedge y = f(z_1, \dots, z_j) \right).$$

– $\lceil y \in \mathrm{Pair}(t_1, t_2)\rceil$ is $\lceil y = t_1\rceil \vee \lceil y = t_2\rceil$.
– $\lceil y \in \{t(v) : v \in r : \varphi(v)\}\rceil$ is

$$\exists v\, (\lceil v \in r\rceil \wedge \lceil \mathtt{true} = \varphi(v)\rceil \wedge \lceil y = t(v)\rceil).$$

Next, we define in H^D the semantics of rules. For each rule R and each dynamic function symbol f, say of arity j, we first define a preliminary formula, $\lceil R$ wants to set f at $x_1, \dots, x_j$ to $y\rceil$, which ignores possible conflicts between updates. This is a formula with free variables $x_1, \dots, x_j, y$ and any variables $z_1, \dots, z_k$ free in the rule R. It holds in state H^D of elements $a_1, \dots, a_j, b, c_1 \dots, c_k$ if and only if $((f, a_1, \dots, a_j), b)$ is in the update set of rule $R(c_1, \dots, c_k)$ in state H^+, as defined in [6]. (We use the symbol f in our name for the formula $\lceil R$ wants to set f at $x_1, \dots, x_j$ to $y\rceil$, but f need not occur in the formula itself.) Typical clauses in the construction of $\lceil R$ wants to set f at $x_1, \dots, x_j$ to $y\rceil$ include:

– $\lceil f(t_1, \dots, t_j) := t_0$ wants to set f at $x_1, \dots, x_j$ to $y\rceil$ is

$$\bigwedge_{i=1}^{j} \lceil x_i = t_i \rceil \wedge \lceil y = t_0 \rceil.$$

– $\lceil$`do forall` $v \in r,\ R$ `enddo` wants to set f at $x_1, \dots, x_j$ to $y\rceil$ is

$$\exists v\, (\lceil v \in r \rceil \wedge \lceil R \text{ wants to set } f \text{ at } x_1, \dots, x_j \text{ to } y \rceil).$$

A rule may want to set f at $x_1, \dots, x_j$ to several values. We adopt the standard ASM convention that if the program Π contains such a conflict, then all the dynamic functions remain unchanged. The formal definitions are as follows.

– $\lceil \Pi$ clashes$\rceil$ is

$$\begin{aligned}\bigvee_f \exists x_1 \dots \exists x_j \exists y \exists z \quad & \\ & (\Pi \text{ wants to set } f \text{ at } x_1, \dots, x_j \text{ to } y) \wedge \\ & (\Pi \text{ wants to set } f \text{ at } x_1, \dots, x_j \text{ to } z) \wedge y \neq z.\end{aligned}$$

Of course, the arity j depends on f.

– $\lceil \Pi$ sets f at $x_1, \dots, x_j$ to $y\rceil$ is

$$\lceil \Pi \text{ wants to set } f \text{ at } x_1, \dots, x_j \text{ to } y \rceil \wedge \neg \lceil \Pi \text{ clashes} \rceil.$$

Finally, we define the dynamic functions for the sequel state, that is, for the state obtained from H^+ by executing Π once.

For a j-ary dynamic function f, we define

$$\lceil y = f(x_1, \dots, x_j) \text{ in the sequel} \rceil$$

to be

$$\begin{aligned}&\lceil \Pi \text{ sets } f \text{ at } x_1, \dots, x_j \text{ to } y \rceil \vee \\ &(\lceil y = f(x_1, \dots, x_j) \rceil \wedge \neg \exists y' \lceil \Pi \text{ sets } f \text{ at } x_1, \dots, x_j \text{ to } y' \rceil\end{aligned}$$

The preceding definitions provide most of the proof of the following result.

Proposition 11 *For any BGS program Π, there exists a number B with the following property. For each natural number m and each dynamic function symbol f, there is a first-order formula $\lceil y = f(x_1 \dots, x_j)$ at step $m\rceil$ in the vocabulary $\{\in, Atoms, A\}$ such that, for any input structure (I, A), the tuples that satisfy $\lceil y = f(x_1 \dots, x_j)$ at step $m\rceil$ in $H = \langle HF(I), \in, I, A\rangle$ constitute the graph of f in the m^{th} state of the run of Π on (I, A). Furthermore, the number of variables occurring in $\lceil y = f(x_1 \dots, x_j)$ at step $m\rceil$ is at most B.*

It will be important that the bound B depends only on Π, not on m. To avoid possible confusion, we emphasize that variables can be re-used in these formulas; thus the same variable may be bound many times and may occur free as well.

Proof We construct the required formulas by induction on m, starting with $\lceil y = f(x_1 \dots, x_j)$ at step $0\rceil$, which can be taken to be $\lceil y = \emptyset\rceil$ because all dynamic functions are initialized to be constant with value $\emptyset$.

For the induction step from m to $m + 1$, we begin with the formula $\lceil y = f(x_1, \dots, x_j)$ in the sequel$\rceil$ as constructed above. Then, for each dynamic function symbol g, we replace each occurrence of a subformula $\lceil t_0 = g(t_1, \dots, t_k)\rceil$ with $\lceil t_0 = g(t_1, \dots, t_k)$ at step $m\rceil$.

As for the bound B, it can be taken to be the maximum number of variables in any of the formulas $\lceil y = f(x_1, \dots, x_j)$ in the sequel$\rceil$ as f ranges over all the dynamic function symbols. We omit the verification of this; it is a fairly standard application of the idea of re-using variables. $\square$

Remark 12 For the purpose of proving Theorem 2, it is important that the number of variables in the formulas $\lceil y = f(x_1, \dots, x_j)$ at step $m\rceil$ be bounded independently of m, but it is not crucial that the formulas be finite. Formulas of the infinitary language $L_{\infty,\omega}$ would serve as well. An alternate approach to obtaining such formulas is to express $\lceil y = f(x_1, \dots, x_j)$ at step $z\rceil$ (with a variable z for the number of steps, numbers being viewed as von Neumann ordinals) in first-order logic with the least-fixed-point operator, and then to use the known translation from this logic into the infinitary, finite-variable logic $L^{\omega}_{\infty,\omega}$ (see [5]). This is the approach used in [2]. Then, to get the corresponding formulas for specific m in place of z, one only has to check that each natural number m can be defined with a fixed number of variables, independent of m. In fact, each natural number can be defined with just three variables.

We shall need a slight generalization of the notions, defined in [2], of "critical" and "active" elements of a state of a BGS computation. Instead of considering only states, we consider *pebbled states* consisting of a state together with an assignment of values to finitely many variables. (Pebbled states are the contexts in which it makes sense to evaluate a term or rule.) When the relevant variables and their ordering are understood, we think of a pebbled state as a state plus a tuple of elements, $(H, a_1, \dots, a_j)$, where a_i is the value assigned to the i^{th} variable. For brevity, we sometimes use vector notation $\boldsymbol{a}$ for $(a_1, \dots, a_j)$.

Definition 13 The *critical* elements of a pebbled state $(H, \boldsymbol{a})$ are

– all the atoms and the set I of atoms,
– the Boolean values `true` and `false`,
– all values of dynamic functions,
– all components of locations where dynamic functions have values other than $\emptyset$, and
– all components of $\boldsymbol{a}$.

An element is *active* if it is in the transitive closure of some critical element.

For ordinary (unpebbled) states, this definition differs from that in [2] only in that the set I of atoms is critical and therefore also active.

We are now in a position to give the example, promised in Section 2, of a BGS program Π together with polynomial bounds on the number of steps and the number of active elements, such that not all of the following three classes of graphs have asymptotic probability 0 or 1: the graphs on which Π halts with output `true` (within the prescribed bounds on steps and active elements), the analogous class for `false`, and the class of graphs on which Π fails to halt within the prescribed bounds. The required Π can be taken to be

```
do forall $x \in$ Atoms
do forall $y \in$ Atoms
   do in parallel
      if $A(x,y)$ then $f(\texttt{Pair}(x,y)) :=$ true,
      Output $:=$ true,
      Halt $:=$ true
   enddo
enddo enddo
```

This program Π only executes once before halting, so we can take the polynomial bound on the number of steps to be 2 and ignore this bound. The number of active elements is $n+3+e$ where n and e are the numbers of vertices and edges in the input graph. (The active elements are the n atoms, the e two-element sets corresponding to edges, the two boolean values, and the set I of atoms.) In a large random graph, the expected value of e is $n(n-1)/4$, i.e., half the number of possible edges, but small fluctuations about this value are probable. Indeed, the asymptotic probability that $e \le n(n-1)/4$ is 1/2. So, if we impose a bound of $n + 3 + n(n-1)/4$ on the number of active elements, then with asymptotic probability 1/2 our program will halt with output `true`, and with asymptotic probability 1/2 it will fail to halt because it cannot execute its single computation step without activating too many elements.

4 Outline of Proof of Zero-One Law

We already know, from Proposition 11, that whether a BGS program halts at a particular step with a particular output can be defined by a first-order sentence over the structure $H = \langle HF(I), \in, I, A\rangle$, with a number of variables that does not depend on the number of steps. A natural approach to proving Theorem 2 would therefore be to use Ehrenfeucht-Fraïssé games and produce winning strategies for the duplicator, to show that such sentences have the same truth value for almost all input graphs (I, A). Unfortunately, that isn't true; for example, the parity of $|I|$ can be defined by a first-order statement over H.

As in [2], this approach can be modified by defining a subclass S of $HF(I)$ for which the duplicator has the necessary winning strategies, and then showing that the definitions given in Proposition 11 remain correct when interpreted in S instead of $HF(I)$. S consists of those elements of $HF(I)$ that have suitable symmetry properties. In [2], symmetry meant invariance under enough automorphisms of the input structure. But almost all graphs have no non-trivial automorphisms, so a subtler approach to symmetry is needed. Shelah introduces a suitable class of partial automorphisms (for any given program Π and any given polynomial bound on the number of active elements) and shows that it leads to an appropriate notion of symmetry. Here "appropriate" means that the symmetry requirements are restrictive enough to provide winning strategies for the duplicator yet are lenient enough to include all the sets actually involved in a computation of Π, limited by the given bound on active elements.

The hardest part of the proof is the leniency just mentioned: The computation involves only symmetric sets. This will be proved by a double induction, first on the stages of the computation and second, within each stage, on the subterms and subrules of Π. That inner induction proceeds along a rather unusual ordering of the subterms and subrules, which we call the computational ordering.

In this double induction, it is necessary to strengthen the induction hypothesis, to say not only that every set x involved in the computation is symmetric but also that all sets x' obtained from x by applying suitable partial automorphisms are also involved in the computation. The assumed bound on the number of active elements will imply a polynomial bound on the number of involved elements. (Not all involved elements are active, but there is a close conection between the two.) That means that the number of x''s is limited, which in turn implies, via a highly non-trivial combinatorial lemma, that x is symmetric.

The traditional extension axioms, as in [5], are satisfied by almost all graphs and are adequate to produce the duplicator's strategies that we need, but they are not adequate to imply the combinatorial lemma needed in the symmetry proof. For this purpose, we need what we call strong extension axioms, saying that every possible type over a finite set is not only realized but realized by a large number of points.

In the next few sections, we shall assemble the tools for the proof that have been mentioned here. After thus describing the proof in somewhat more detail, we shall add some sections about related issues.

5 Tasks and Their Computational Order

To compute the value of a term or the update set of a rule, one needs a structure (the state of the computation) and values for the variables free in the term or rule. In most of our discussion, there will be a fixed structure under consideration but many choices of values for variables. It will therefore be convenient to push the structures into the background, often neglecting even to mention them, and to work with pairs whose first component is a term or rule and whose second component is an assignment of values to some variables (including all the variables free in the first component). We shall call such pairs "tasks" because we regard them as things to be evaluated in the course of a computation.

The official definition of tasks will differ in two respects from this informal description. First, we shall always have a particular program Π under consideration, and we deal only with terms and rules occurring in Π. Recall in this connection our Convention 9, by which we are really dealing with occurrences of terms and rules. Second, it will be convenient to include in our tasks not only the obviously necessary values for free variables but also values for certain additional variables, defined as follows.

Definition 14 A variable v is *pseudo-free* in a term t or rule R if t or R lies in the scope of v.

Because Π, being a program, has no free variables, all the free variables in a term or rule must also be pseudo-free in it. But there may be additional pseudo-free variables. From a syntactic point of view, v is pseudo-free in t or R if one could introduce v as a free variable in t or R without violating the requirement that Π have no free variables. From a computational point of view, the pseudo-free variables of t or R are those variables to which specific values have been assigned whenever one is about to evaluate t or R in the natural algorithm for executing Π.

Definition 15 A *term-task* (relative to a program Π and a state H) is a pair $(t, \boldsymbol{a})$ where t is a term in Π and $\boldsymbol{a}$ is an assignment of values in H to all the pseudo-free variables of t. *Rule-tasks* are defined similarly except that the first component is a rule in Π. Term-tasks and rule-tasks are collectively called *tasks*.

Although the second component $\boldsymbol{a}$ of a task is officially a function assigning values to the pseudo-free variables of the first component, we sometimes view it as simply a tuple of values. This makes good sense provided we imagine a fixed order for the pseudo-free variables. We also sometimes write $\boldsymbol{a}\restriction$ for the restriction of the function $\boldsymbol{a}$ to an appropriate subset of its domain; it will always be clear from the context what the appropriate subset is.

We write $\mathrm{Val}(t, \boldsymbol{a})$ for the value of the term t in a structure (assumed fixed throughout the discussion) when the free variables are assigned values according to $\boldsymbol{a}$.

Definition 16 The *computational order* $\prec$ is the smallest transitive relation on tasks satisfying the following conditions.

- $(t_i, \boldsymbol{a}) \prec (f(t_1, \dots, t_j), \boldsymbol{a})$ for $1 \le i \le j$.
- $(r, \boldsymbol{a}) \prec (\{s : v \in r : \varphi\}, \boldsymbol{a})$ and, for each $b \in \mathrm{Val}(r, \boldsymbol{a})$, all three of $(v, \boldsymbol{a}, b)$, $(s, \boldsymbol{a}, b)$ and $(\varphi, \boldsymbol{a}, b)$ are $\prec (\{s : v \in r : \varphi\}, \boldsymbol{a})$.
- $(t_i, \boldsymbol{a}) \prec (f(t_1, \dots, t_j) := t_0)$ for $0 \le i \le j$.
- All of $(\varphi, \boldsymbol{a})$, $(R_0, \boldsymbol{a})$, and $(R_1, \boldsymbol{a})$ are $\prec$ $(\texttt{if}\ \varphi\ \texttt{then}\ R_0\ \texttt{else}\ R_1\ \texttt{endif}, \boldsymbol{a})$.
- $(r, \boldsymbol{a}) \prec (\texttt{do forall}\ v \in r,\ R\ \texttt{enddo}, \boldsymbol{a})$ and, for each $b \in \mathrm{Val}(r, \boldsymbol{a})$, both $(v, \boldsymbol{a}, b)$ and $(R, \boldsymbol{a}, b)$ are $\prec (\texttt{do forall}\ v \in r,\ R\ \texttt{enddo}, \boldsymbol{a})$.
- If r is the range of a variable v pseudo-free in t or R, then $(r, \boldsymbol{a}\restriction) \prec (t, \boldsymbol{a})$ or $(r, \boldsymbol{a}\restriction) \prec (R, \boldsymbol{a})$, respectively.

Except for the last clause, the definition assigns as the predecessors of a task simply the subterms or subrules of its first component, equipped with suitable values for the variables. The last clause, however, is quite different. Here the first component r of the lower clause may well be much larger than the first component t or R of the upper clause.

Intuitively, if one task precedes another in this computational ordering, then in the natural calculation involved in the execution of Π the former task would be carried out before the latter. In the range clause, the idea is that, before attempting to evaluate t or R, we would have assigned a value to v, and before that we would have evaluated the range in order to know what the possible values for v are.

This intuition strongly suggests that the computational order should be well-founded. In fact it is, but the proof is not trivial and will only be sketched here. To treat term-tasks and rule-tasks simultaneously, we use $X, Y, \dots$ to stand for either terms t or rules R.

Proposition 17 *There is a rank function ρ, mapping terms and rules to natural numbers, such that if $(X, \boldsymbol{a}) \prec (Y, \boldsymbol{b})$ then $\rho(X) < \rho(Y)$.*

Proof Let V be the number of variables occurring in Π, and let D be the maximum depth of nesting of terms and rules occurring in Π.

For each variable v in Π, let $\tau(v)$ be the number of symbols in the term or rule that binds v. Notice that, if x is bound in the range of y then $\tau(x) < \tau(y)$.

Call a variable *relevant* to a term or rule X if it is either pseudo-free in X or bound in X. In other words, the scope of v either includes or is included in X. Define $\sigma(X)$ to be the sum of $V^{\tau(v)}$ over all variables relevant to X. As one goes downward in the computational order, using any clause in its definition except the range clause, the set of relevant variables either remains the same or shrinks, so the value of σ remains the same or decreases. Furthermore, when σ stays the same, the depth of nesting inside the term or rule decreases. As one goes downward using the range clause, the variable v explicitly mentioned in the clause loses its relevance, but other variables, bound in r, may become relevant. All of the latter have τ values strictly smaller than that of v, and there are fewer than V of them, so σ still decreases.

Therefore, if we define

$$\rho(X) = (D + 1)\sigma(X) + \mathrm{depth}(X)$$

where depth(X) means the depth of nesting of terms and rules in X, then $\rho(X)$ decreases at every downward step in the computational ordering. $\square$

This proposition immediately implies that the computational order is a strict partial order; it has no cycles. Since finite partial orders are well-founded, it is legitimate to do proofs by induction with respect to $\prec$. Such proofs can also be regarded as ordinary mathematical induction on the rank ρ. Furthermore, induction on ρ is somewhat more powerful than induction with respect to $\prec$ because ρ-induction allows us, when proving a statement for some task, to assume the same statement not only for all computationally earlier tasks $(X, \boldsymbol{a})$ but also for all tasks $(X, \boldsymbol{b})$ having the same first components as these. This is because ρ depends only on the first component.

The following lemma is useful technically, and it also serves to clarify the role of the range clause in the definition of the computational ordering. The definition of this ordering $\prec$ ensures that, if one task T precedes another T' then there is a chain

$$T' = T_0 \succ T_1 \succ \cdots \succ T_n = T$$

joining them, in which each two consecutive terms are related as in one of the clauses in Definition 16.

Lemma 18 *If $T' \succ T$ then there is a chain as above, in which the range clause is used, if at all, only at the first step, from T_0 to T_1.*

We omit the proof, which is an induction on the length n of the chain.

Corollary 19 *If a task T precedes a task T' with no pseudo-free variables in its first component (and thus with empty second component), then this can be established without the range clause.*

Proof The chain from T' down to T obtained in the lemma must not use the range clause at all, for the range clause is not applicable at the first step in the absence of pseudo-free variables in T'. $\square$

The important case of the corollary is when (the first component of) T' is the whole program Π.

6 Involved and Active Elements

Throughout this section, we assume that we are dealing with a fixed program Π and a fixed state arising during its execution on some input (I, A). As before, we write H for the structure $\langle HF(I), \in, I, A\rangle$. The state under consideration is an expansion of H, interpreting the dynamic function symbols as well as the other static function symbols. We write H^+ for this state. Notice that, by Proposition 11, the interpretations of the additional function symbols of H^+ are all definable in H. The definitions of the dynamic function symbols depend on the stage of the computation; the others do not.

The following definition is intended to capture the intuitive notion of an element of $HF(I)$ being "looked at" during the execution of a task T. It doesn't quite fulfill this intention, for it includes too many elements, pretending for example that execution of an `if` ... `then` ... `else` ... includes execution of both the consitituent rules regardless of the truth value of the guard. Nevertheless, it serves our purposes because (1) it includes all the elements that are really needed (see Lemma 22 below) and (2) it doesn't include too many additional elements (see Lemma 23 below).

Definition 20 An element $c \in HF(I)$ is *involved* in a task $(X, \boldsymbol{a})$ (with respect to a program Π and a state H^+) if either it is active in $(H^+, \boldsymbol{a})$ or it is the value in H^+ of some term-task $\prec (X, \boldsymbol{a})$.

The next three lemmas give the key properties of this notion of "involved."

Lemma 21 *Any object that is not active in a state H^+ but is active in its sequel with respect to Π must be involved in the task Π with respect to state H^+.*

Lemma 22 *The set-theoretic definitions at the end of Section 3, in particular Proposition 11, remain correct if the quantifiers are interpreted to range only over elements involved in Π rather than over all of $HF(I)$.*

This is proved by inspecting all those definitions, including the ones that we did not explicitly exhibit, seeing which values of the quantified variables are essential, and checking that these values are all involved in Π.

Lemma 23 *For any state H^+ and any task $(X, \boldsymbol{a})$, the number of elements involved in $(X, \boldsymbol{a})$ is bounded by a polynomial function of the number of active elements in $(H^+, \boldsymbol{a})$. The polynomial depends only on the term or rule X.*

This is proved by induction on X. We are interested in the lemma primarily in the case that the task $(X, \boldsymbol{a})$ is the entire program Π (with the empty assignment, as Π has no pseudo-free variables). A $\tilde{\text{C}}$PTime program comes with a polynomial bound on the number of active elements during its run; the lemma allows us to deduce a (possibly larger) polynomial bound on the number of involved elements. This will be crucial in showing that the computation takes place entirely in the world of symmetric sets.

7 Strong Extension Axioms

Let $\tau(x_0, x_1, \ldots, x_k)$ be a quantifier-free formula of the form

$$\Big(\bigwedge_{1\le i<j\le k} x_i \neq x_j\Big) \to \Big(\bigwedge_{1\le i\le k} (x_0 \neq x_i) \wedge \pm(x_0 A x_i)\Big),$$

so that, for a given k, there are exactly 2^k different formulas of that form. The extension axiom $\text{EA}(\tau)$ is the axiom $\forall x_1 \ldots x_k \exists x_0 \tau(x_0, \ldots, x_k)$. A graph G satisfies the *strong extension axiom SEA*(τ), if for all distinct vertices $x_1, \ldots, x_k$, there are at least $\frac{1}{2}n/2^k$ vertices x_0 in G satisfying $\tau(x_0, x_1, \dots, x_k)$. The extension axiom EA_k is the conjunction of all 2^k extension axioms $\mathrm{EA}(\tau)$ with $k+1$ variables; the strong extension axiom SEA_k is the conjunction of all 2^k strong extension axioms $\mathrm{SEA}(\tau)$ for all τ with $k+1$ variables.

Thus, EA_k says that every possible configuration for a vertex x_0, relative to k given vertices $x_1, \dots, x_k$, actually occurs. SEA_k says that every such configuration occurs fairly often.

Why $\frac{1}{2}n/2^k$? In a random graph with n vertices, the probability that an arbitrary vertex a_0, different from $a_1, \dots, a_k$, satisfies $\tau(a_0, a_1, \dots, a_k)$ is $1/2^k$, so the expected number of vertices a_0 satisfying $\tau(a_0, a_1, \dots, a_k)$ is $(n-k)/2^k$. So with high probability, there are at least $\frac{1}{2}n/2^k$ vertices a_0 in G satisfying $\tau(a_0, a_1, \dots, a_k)$. The factor $\frac{1}{2}$ could be replaced with any positive constant $c < 1$; that gives a strong extension axiom SEA_k^c.

Lemma 24 *For each k, the asymptotic probability of SEA_k is 1.*

The proof uses Chernoff's inequality from probability theory.

8 Supports

In this section, we describe the notion of symmetry with respect to partial automorphisms that will, as explained in Section 4, apply to all objects involved in a computation and lead to winning strategies for the duplicator in certain Ehrenfeucht-Fraïssé games. Two numerical parameters will be involved in this notion of symmetry, namely the size of the partial automorphisms and the number of atoms a symmetric object can depend on. The appropriate values for these parameters will depend on the BGS program and the polynomial bound on the number of active elements.

For the purposes of this section, let $q \geq 1$ and $k \geq 4$ be fixed integers. When this material is applied in the proof of Theorem 2, q will be the degree of a polynomial bounding the number of involved elements (obtainable from Π and the bound on active elements, via Lemma 23) and k will be $2B+4$, where B is as in Proposition 11.

We assume that the input graph (I, A) satisfies the extension axioms for up to $3kq$ variables. (We don't need the strong extension axioms yet.)

For brevity we adopt the conventions that

- w, x, y, z (possibly with subscripts, superscripts, or accents) stand for members of $HF(I)$,
- a, b, c (possibly with subscripts, superscripts, or accents) stand for sets of $\leq q$ atoms, and we call such sets *possible supports*, and
- $\alpha, \beta, \gamma, \delta, \zeta$ (possibly with subscripts or superscripts) stand for partial automorphisms of the graph (I, A) whose domains have

The inverse α^{-1} of a motion and the composite $\alpha \circ \beta$ of two motions are defined in the obvious way. In particular, the domain of $\alpha \circ \beta$ is $\beta^{-1}(\mathrm{Dom}(\alpha))$.

The extension axioms imply that, if α is a motion and s is a set of atoms with $|\mathrm{Dom}(\alpha)| + |s| \leq kq$, then α can be extended to a motion whose domain includes s. (In fact, the extension axioms imply considerably more, as they go up to $3kq$ variables, not just the kq needed for the preceding statement.)

We next define, by simultaneous recursion on the rank of x, the three concepts "a supports x," "x is supported," and "$\hat{\alpha}(x)$," where the last of these is defined if and only if $\mathrm{Dom}(\alpha)$ includes some a that supports x.

Definition 25 If x is an atom, then

- a supports x if and only if $x \in a$,
- x is supported (always), and
- $\hat{\alpha}(x) = \alpha(x)$.

If, on the other hand, x is a set, then

- a supports x if and only if every $y \in x$ is supported and, for every y of lower rank than x and every motion α, if α pointwise fixes a and if $\mathrm{Dom}(\alpha)$ includes some set supporting y, then

$$y \in x \iff \hat{\alpha}(y) \in x,$$

- x is supported if and only if some a supports x, and
- if a supports x and $a \subseteq \mathrm{Dom}(\alpha)$, then $\hat{\alpha}(x)$ is the set of all $\hat{\beta}(y)$ where $y \in x$, $\beta \restriction a = \alpha \restriction a$, and $\mathrm{Dom}(\beta)$ includes some support of y.

The definition of $\hat{\alpha}(x)$ when x is a set seems to depend on the choice of a particular support a of x. The first part of the following lemma gets rid of that apparent dependence; the rest of the lemma gives useful technical information about supports and about the application of motions to supported sets.

Lemma 26 *1. $\hat{\alpha}$ is well-defined. Specifically, if a_1 and a_2 both support x and are both included in Dom(α), then $\hat{\alpha}^1(x)$ and $\hat{\alpha}^2(x)$, defined as above using a_1 and a_2 respectively, are equal.*

2. *If a supports x and $a \subseteq$ Dom(α) $\cap$ Dom(β) and $\alpha \restriction a = \beta \restriction a$, then $\hat{\alpha}(x) = \hat{\beta}(x)$.*
3. *If $\hat{\alpha}(x)$ is defined then it has the same rank as x.*
4. *If α is an identity map, then so is $\hat{\alpha}$, i.e., $\hat{\alpha}(x) = x$ whenever $\hat{\alpha}(x)$ is defined.*
5. *$\hat{\alpha}$ is a partial automorphism of the structure $(HF(I), \in, I, A)$. In other words, $\hat{\alpha}(I) = I$ and, whenever x and x' have supports included in Dom(α),*

$$\begin{aligned} x' = x &\iff \hat{\alpha}(x') = \hat{\alpha}(x) \\ x' \in x &\iff \hat{\alpha}(x') \in \hat{\alpha}(x) \\ x'Ax &\iff \hat{\alpha}(x')A\hat{\alpha}(x). \end{aligned}$$

6. *If a supports x and $a \subseteq$ Dom(α) then $\alpha[a]$ supports $\hat{\alpha}(x)$.*
7. *If $\hat{\alpha}(x)$ is defined and $a \subseteq$ Dom(α) and $\alpha[a]$ supports $\hat{\alpha}(x)$, then a supports x.*

8. *$\widehat{\beta\circ\alpha}=\hat\beta\circ\hat\alpha$ in the following sense: If $\widehat{\beta\circ\alpha}(x)$ is defined then so is $\hat\beta(\hat\alpha(x))$ and they are equal.*

All eight parts of the lemma are proved together, by induction on the maximum of the ranks of x and x'. We omit the tedious proof.

Definition 27 S is the collection of supported objects in $HF(I)$. We also write S for the structure $(S, I, \in, A)$.

By the definition of supports, S is a transitive set containing all the atoms; by (5) of Lemma 26, it also contains I. Furthermore, by (5) and (6) of that lemma, each $\hat\alpha$ is a partial automorphism of S. In fact, as the following lemma shows, the $\hat\alpha$'s are much better than just partial automorphisms, because they fit together well. Recall that $L^k_{\infty,\omega}$ is the part of the infinitary first-order language $L_{\infty,\omega}$ (allowing infinite conjunctions and disjunctions) consisting of formulas with at most k variables (free or bound, but the same variable can be re-used). Recall also that $k\geq 4$ is fixed throughout this section.

Lemma 28 *Let φ be a formula of $L^k_{\infty,\omega}$ with $j\leq k$ free variables. Let α be a motion, and let $x_1,\dots,x_j$ be elements of S with supports included in $Dom(\alpha)$. Then*

$$S\models\varphi(x_1,\dots,x_j)\iff S\models\varphi(\hat\alpha(x_1),\dots,\hat\alpha(x_j)).$$

Proof We give a strategy for the duplicator in the Ehrenfeucht-Fraïssé game for $L^k_{\infty,\omega}$. At any stage of the game, let $\boldsymbol{y}$ and $\boldsymbol{z}$ be the positions of the pebbles on the two boards; so initially, $y_i=x_i$ and $z_i=\hat\alpha(x_i)$. The duplicator's strategy is to arrange that there is always a motion β whose $\hat\beta$ sends each y_i to the corresponding z_i. There is such a β initially, namely α, and as long as he maintains such a β the duplicator cannot lose, by (5) of Lemma 26. So we need only check that, if such a β exists and then the spoiler moves, the duplicator can move so that again a (possibly new) β does the job. Without loss of generality, suppose the spoiler moves the first pebble on the left board from its position y_1 to a new $y_1'\in S$. Restrict the old β to the union of supports of the y_i for $i\neq 1$. There are strictly fewer than k of these supports, hence at most $(k-1)q$ points in the domain of the restricted β. So we can extend this motion to a new β having in its domain some support of the new y_1'. The resulting $\hat\beta(y_1')$ is where the duplicator should move the pebble from z_1. The resulting board position and the new β satisfy the specification of the duplicator's strategy (thanks to (2) of Lemma 26). So we have shown that the duplicator can carry out the indicated strategy. $\square$

We shall need variants of these results, dealing with two graphs and the universes of hereditarily finite sets built over them. Specifically, suppose (I_1, A) and (I_2, A) are graphs satisfying the extension axioms for up to $3kq$ variables. (We've simplified notation slightly by using the same name A for the adjacency relations in both graphs I_i.) For $i,j\in\{1,2\}$, we define an *i,j-motion* to be a partial isomorphism of size at most kq from I_i to I_j. Thus 1,1-motions and 2,2-motions are motions in the earlier sense for I_1 and I_2, respectively.

If α is a 1,2-motion, then we define $\hat{\alpha}(x)$ for all $x \in HF(I_1)$ having supports included in $\mathrm{Dom}(\alpha)$. We do this in exact analogy with the earlier definition: If x is an atom then $\hat{\alpha}(x) = \alpha(x)$. If x is a set with a support $a \subseteq \mathrm{Dom}(\alpha)$, then $\hat{\alpha}(x)$ is the set of all $\hat{\beta}(y)$ where $y \in x$, β is a 1,2-motion extending $\alpha \upharpoonright a$, and $\mathrm{Dom}(\beta)$ includes some support of y.

Similarly, we define $\hat{\alpha}(x)$ when α is a 2,1-motion and $x \in HF(I_2)$ has a support $\subseteq \mathrm{Dom}(\alpha)$. These definitions together with the definitions already available from Section 8 for 1,1- and 2,2-motions allow us to refer to $\hat{\alpha}(x) \in HF(I_j)$ whenever α is an i,j-motion and $x \in HF(I_i)$ has a support included in $\mathrm{Dom}(\alpha)$.

We can now repeat the earlier arguments in this slightly more general context. No conceptual changes or additions are needed, only a little bookkeeping to keep track of the four different sorts of motions. We exhibit for future reference the 1,2-analog of Lemma 28. Let S_i be the collection of supported objects in $HF(I_i)$; as before, we also write S_i for the structure $(S_i, \in, I_i, A)$.

Lemma 29 *Let φ be a formula of $L^k_{\infty,\omega}$ with $j \leq k$ free variables. Let α be a 1,2-motion, and let $x_1, \ldots, x_j$ be elements of S_1 with supports included in Dom(α). Then*

$$S_1 \models \varphi(x_1, \ldots, x_j) \iff S_2 \models \varphi(\hat{\alpha}(x_1), \ldots, \hat{\alpha}(x_j)).$$

9 Combinatorics

In this section, we describe (without proof) the main combinatorial lemma needed in the proof of the zero-one law. The parameters q and k are fixed as before, and the graph (I, A) of atoms is now assumed to satisfy the *strong* extension axioms for up to $3kq$ variables.

By a *polymer* we mean a sequence of at most kq atoms. (The reason for the terminology is that, in [2], we used "molecule" for a one-to-one listing of a support; here that would be a sequence of length q. A polymer is essentially the concatenation of up to k molecules. It gives the supports for up to k objects.) The *configuration* of a polymer consists of the following information: which components are equal and which are adjacent in the graph of atoms. If the polymer has length l, then its configuration could be viewed as the combination of an equivalence relation on $\{1, 2, \ldots, l\}$ (telling which components are equal) and an irreflexive symmetric relation on the quotient set (telling which components are adjacent). By the *joint configuration* of two (or more) polymers, we mean the equality and adjacency information about all components of both (or all) of the polymers. It could be viewed as the configuration of the concatenation of the polymers, except for the technicality that the concatenation may be too long to count as a polymer.

We shall be concerned with equivalence relations of the following sort.

Definition 30 A *configuration-determined equivalence* (*cde* for short) is an equivalence relation E with the following two properties.

- Its domain consists of all polymers of one specified configuration.

– Whether two polymers ξ and η of this configuration are related by E depends only on their joint configuration.

When dealing with polymers ξ of a specified configuration (e.g., those in the domain of a cde), we can simplify their presentation by omitting any repetitions of components in ξ. Because of the tight correspondence between the polymers ξ of a known configuration and these compressed versions, we can confine our attention to the compressed versions; that is, we can assume that we deal only with polymers that are one-to-one sequences.

Theorem 31 *Assume that $|Atoms| \geq kq2^{3kq+1}$. Let E be a configuration-determined equivalence with fewer than*

$$\frac{1}{(q+1)!}\left(\frac{|Atoms|}{2^{3kq+1}}\right)^{q+1}$$

equivalence classes. Let l be the common length of the polymers in the domain of E. There exists a set $u \subseteq \{1,2,\ldots,l\}$ of size at most q, and there exists a group G of permutations of u such that, for any ξ and η in the domain of E,

$$\xi E \eta \iff (\exists \sigma \in G)\,(\forall i \in u)\,\xi_i = \eta_{\sigma(i)}.$$

Notice that, although l can be as large as kq, the theorem requires u to be relatively small, of size at most q.

The conclusion of the theorem completely describes E in terms of u and G. It says that the E-equivalence class of ξ consists of those polymers (of the right configuration) obtainable from ξ by

– permuting the components indexed by u, using a permutation from G, and
– changing the other components completely arbitrarily.

In particular, the equivalence class of ξ depends only on the ξ_i for $i \in u$.

The hypothesis of the theorem involves a complicated bound on the number of equivalence classes. Most of the complication disappears if one remembers that q and k are fixed, so the bound is, up to a constant factor, just $|\text{Atoms}|^{q+1}$. When we apply the theorem, the cde's of interest will be such that the equivalence classes correspond to elements involved in the computation, so their number is bounded by a polynomial in $|\text{Atoms}|$ of degree at most q. So the bound will automatically be satisfied once the number of atoms is large enough.

Because of space and time limitations, we omit Shelah's proof of this combinatorial theorem.

10 Putting the Proof Together

Consider a BGS program Π and a polynomial bound on the number of active elements. According to Lemma 23, we obtain a polynomial bound on the number of elements involved in Π at any stage of the computation. Let q be the degree of this polynomial. Also, let $k = 2B + 4$ where B is the bound from Proposition 11 on the number of variables in the set-theoretic formulas describing the computation of Π. Whenever we refer to supports, motions, etc., we take these concepts to refer to the particular q and k just introduced.

In the following theorem, to say that a formula is absolute for S means that the formula's meaning does not change if the quantifiers are interpreted as ranging over the class S of supported objects rather than over all of $HF(I)$.

Theorem 32 *Assume that all active elements in state H^+ are supported. Then the following are true for every term-task $(t, \boldsymbol{a}) \prec \Pi$, provided the input graph (I, A) satisfies the strong extension axioms up to $3kq$ variables and is large enough.*

1. *If $(t, \boldsymbol{a}) \prec (X, \boldsymbol{a}')$ and if α is a motion whose domain includes supports of all the elements of $\boldsymbol{a}$, then $(t, \hat{\alpha}(\boldsymbol{a})) \prec (X, \hat{\alpha}(\boldsymbol{a}'))$.*
2. *All elements of $\boldsymbol{a}$ and of Val$(t, \boldsymbol{a})$ are supported.*
3. *The formula defining $x \in$ Val$(t, \boldsymbol{y})$ is absolute for S when $\boldsymbol{y}$ is instantiated to $\boldsymbol{a}$.*
4. *Val$(t, \boldsymbol{a})$ is supported. Furthermore, given supports for all components of $\boldsymbol{a}$, their union includes a support for Val$(t, \boldsymbol{a})$.*
5. *The formula defining $x =$ Val$(t, \boldsymbol{y})$ is absolute for S when $\boldsymbol{y}$ is instantiated to $\boldsymbol{a}$.*

All five parts of the theorem are proved in a simultaneous induction on $(t, \boldsymbol{a})$ with respect to the rank function ρ from Proposition 17 the computational order. The proof is too long to give here, but we describe a few points in it that explain why much of the work in previous sections is needed.

The computational order, and specifically the range clause in its definition, are crucial in the proof of (1). Consider, for example, a situation where $\boldsymbol{a} = (\boldsymbol{c}, b)$ and $(t, \boldsymbol{c}, b) \prec (Y, \boldsymbol{c})$ where b is the value assigned to a variable v bound by Y and thus pseudo-free in the subterm t. So $b \in \mathrm{Val}(r, \boldsymbol{c})$, where r is the range of v. To show that $(t, \hat{\alpha}(\boldsymbol{c}), \hat{\alpha}(b)) \prec (Y, \hat{\alpha}(\boldsymbol{c}))$, we want to know that $\hat{\alpha}(b) \in \mathrm{Val}(r, \hat{\alpha}(\boldsymbol{c}))$. Fortunately, the range clause makes $(r, \boldsymbol{c})$ a computational predecessor of $(t, \boldsymbol{c}, b)$, so we can apply induction hypotheses (2) and (3) to it. Thus, the fact that $b \in \mathrm{Val}(r, \boldsymbol{c})$ can be expressed as a set-theoretic statement true in S. And this statement will retain its truth value when we apply $\hat{\alpha}$, by Lemma 28.

The combinatorial Theorem 31 is used in the hardest case in the proof of (4), namely where t is $\{s : w \in r : \varphi\}$. Choose supports for all the a_i and let ξ_0 be a polymer in which all those supports are listed. For any other polymer ξ of the same configuration as ξ_0, let α be the motion sending ξ_0 to ξ. Write $t(\xi)$ for $\mathrm{Val}(t, \hat{\alpha}(\boldsymbol{a}))$. In particular, $t(\xi_0)$ is the object $\mathrm{Val}(t, \boldsymbol{a})$ that we hope to prove to be supported.

Define an equivalence relation E on the set of polymers of the same configuration as ξ_0 by

$$\xi E \xi' \iff t(\xi) = t(\xi').$$

One can verify, using the induction hypotheses and Lemma 28, that this E is configuration-determined. The number of equivalence classes of E is the number of different elements of the form $t(\xi) = \mathrm{Val}(t, \hat{\alpha}(\boldsymbol{a})$. It follows from part (1) that the number of such elements is bounded by a constant times $|I|^q$. That's smaller than the bound required in the Dichotomy Theorem, a polynomial of degree $q+1$, provided $|I|$ is large enough. So applying the combinatorial Theorem 31, we find that $t(\xi)$ depends only on the restriction of ξ to a certain set u of size at most q. Then one can show that the range of $\xi_0 \restriction u$ supports $\mathrm{Val}(t, \boldsymbol{a})$.

Corollary 33 *Assume that all active elements in state H^+ are supported. Then so are all objects involved in Π with respect to H^+.*

This is immediate from part (4) of the theorem and the definition of "involved." By Lemma 21, it implies that every active element of the sequel is supported, and so the theorem and its corollary are applicable to the sequel. Proceeding inductively and then invoking Lemma 22, we find that the set-theoretic formulas describing the computation are all absolute for S as long as the polynomial bound on the number of active elements is obeyed. But then, thanks to Lemma 29, the truth values of these formulas and therefore the behavior of the computation are the same for all input graphs that satisfy the necessary strong extension axioms and are sufficiently large. By virtue of Lemma 24, the class of such graphs has asymptotic probability 1, so Theorem 2 is proved.

11 Extension and Strong Extension

The zero-one law for first-order logic is based on the extension axioms: for every first-order sentence φ (of relational vocabulary), there exists k such that EA_k implies φ or EA_k implies $\neg\varphi$. The same holds for fixed-point logic FO+LFP and for the infinitary logic $L^{\omega}_{\infty,\omega}$ [5]. However, extension axioms are too weak to support the zero-one law for $\tilde{\mathrm{C}}$PTime. We give an example of a single polynomial time BGS program that separates structures satisfying arbitrarily many extension axioms. So strong extension axioms are really needed for the $\tilde{\mathrm{C}}$PTime zero-one law.

Example 34 For simplicity, we consider graphs equipped with a unary relation, (I, A, R). We informally describe a BGS program computing the maximal size of a clique included in R.

– In mode Initial, initialize i, p to 0, initialize C to $\{\emptyset\}$, and go to mode Compute. Intuitively, i is a counter, p is the parity of i and C is the collection of all subsets of R of size i.
– In mode Compute, increase i by one, flip p, update C as follows

$$C := \{x \cup \{y\} : x \in C \wedge y \in R\}$$

and go to mode Decide.
– In mode Decide, check if C contains a clique. If yes then go to mode Compute; otherwise output $1-p$ and halt.

Consider running this program on the input having vertex set $\{1, 2, \ldots, n\}$, a random adjacency relation A, and R interpreted as $\{1, 2, \ldots, r\}$ for some $r < n$. It follows from results in [4, Section XI.1] that there are values of r of magnitude roughly $\log n \cdot \log\log n$ for which the clique number of $(R, A \restriction R)$ is very probably even and there are other nearby values of r for which this clique number is very probably odd. Also, because r is so much smaller than n (and the cliques smaller yet) the computation of Π will very probably activate fewer than n sets. So we can impose a polynomial bound slightly larger than $2n$ and be reasonably certain that the computation will halt. Finally, by choosing r and therefore n large enough, one can ensure (again with very high probability) that the graphs under consideration satisfy any specified extension axiom EA_k.

Since the strong extension axioms go beyond the ordinary extension axioms, one might hope that they imply some of the classical properties of almost all graphs, like rigidity and hamiltonicity, that are known [3] not to follow from extension axioms.

This is not the case for rigidity. The non-rigid graphs constructed in [3] — random modulo an imposed symmetry — can be shown to also satisfy strong extension axioms.

For Hamiltonicity, the situation is less clear. We can show, again using the construction from [3], that the axioms SEA_k^c for $c < \frac{1}{2}$ do not imply Hamiltonicity. For $c \geq \frac{1}{2}$, these examples no longer work, but we do not know whether others are available.

12 The Almost Sure Theory is Undecidable

In the case of first-order logic, the almost sure theory (that is the set of almost surely true sentences) is decidable. The same holds if we add the least fixed point operator to first-order logic [1]. But it fails for $\tilde{\mathrm{C}}$PTime.

Proposition 35 *The class of almost surely accepting polynomially bounded programs and the class of almost surely rejecting polynomially bounded programs are recursively inseparable.*

Proof Consider Turing machines with two halting states h_1 and h_2. For $i = 1, 2$, let H_i be the collection of Turing machines that halt in state h_i on the empty input tape. It is well-known that H_1 and H_2 are recursively inseparable. Associate to each Turing machine T a polynomial time BGS program as follows. The program Π ignores its input graph and simulates T on empty input tape (working exclusively with pure sets). Π outputs `true` (resp. `false`) if T halts in state h_1 (resp. h_2). The polynomial bounds on steps and activated elements are both the identity function, i.e., the number of atoms. Then if $T \in H_1$ (resp. $T \in H_2$) our polynomial time BGS program will accept (resp. reject) all sufficiently large inputs. $\square$

References

1. Andreas Blass, Yuri Gurevich, and Dexter Kozen, *A zero-one law for logic with a fixed-point operator*, Information and Control 67 (1985) 70–90.
2. Andreas Blass, Yuri Gurevich, and Saharon Shelah, *Choiceless polynomial time*, Ann. Pure Applied Logic 100 (1999) 141–187.
3. Andreas Blass and Frank Harary, *Properties of almost all graphs and complexes*, J. Graph Theory 3 (1979) 225–240.
4. Béla Bollobás, Random graphs, Academic Press, 1985.
5. Heinz-Dieter Ebbinghaus and Jörg Flum, Finite Model Theory, Springer-Verlag (1995).
6. Yuri Gurevich, *Evolving algebras 1993: Lipari guide*, in Specification and Validation Methods, ed. E. Börger, Oxford University Press (1995) pp. 9–36. See also the *May 1997 draft of the ASM guide*, Tech Report CSE-TR-336-97, EECS Dept., Univ. of Michigan, 1997. Found at `http : //www.eecs.umich.edu/gasm/`.
7. Saharon Shelah, *Choiceless polynomial time logic: inability to express* [paper number 634], these proceedings.