# Abstract state machines and computationally complete query languages*

Andreas Blass†
Mathematics Department
University of Michigan
Ann Arbor, MI 48109-1109
USA

Yuri Gurevich
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
research.microsoft.com/~gurevich/

Jan Van den Bussche
University of Limburg (LUC)
B-3590 Diepenbeek
Belgium.

**Abstract**

Abstract state machines (ASMs) form a relatively new computation model holding the promise that they can simulate any computational system in lockstep. In particular, an instance of the ASM model has recently been introduced for computing queries to relational databases. This model, to which we refer as the BGS model, provides a powerful query language in which all computable queries can be expressed. In this paper, we show that when one is only interested in polynomial-time computations, BGS is strictly more powerful than both QL and $while_{new}$, two well-known computationally complete query languages. We then show that when a language such as $while_{new}$ is extended with a duplicate elimination mechanism, polynomial-time simulations between the language and BGS become possible.

**Keywords:** ASM, choiceless, polynomial time, database query, complete query language, QL, while

*Preprint of an article in *Information and Computation*, 174(1):20–36, 2002.
†Partially supported by a grant from Microsoft Research.

# 1 Introduction

Abstract state machines (ASMs) were introduced as a new computation model, accompanied by the "ASM thesis" stating that any algorithm, or more broadly, any computational system, at any level of abstraction, can be simulated in lockstep by an ASM [16, 12, 13, 14]. Recently, Blass, Gurevich, and Shelah (BGS) introduced an instance of the ASM model for expressing queries to relational databases [7].

Roughly, a BGS program is a complex rule, changing the values of certain dynamic functions at various arguments during the run of the program. Rules are built up from elementary updates by conditionals and parallel composition. The program is iterated until a halting condition is reached. A powerful sublanguage of terms provides set-theoretic operations on arbitrarily nested sets over the input data elements. Once "activated," these sets are incorporated in the run of the program, and can become arguments and values of dynamic functions. While any computable query can be expressed in BGS, the actual motivation of BGS to introduce their model was to study the complexity class denoted by $\widetilde{\text{C}}$PTIME, corresponding to BGS programs under a polynomial time restriction.

Computationally complete query languages have been known in database theory for some years now [1], and complexity classes similar to $\widetilde{\text{C}}$PTIME, denoted by GEN-PTIME and GEN-PSPACE, were introduced by Abiteboul and Vianu [6]. These classes can be defined in terms of the language $while_{new}$.[1] This language is the extension of first-order logic with the following features: (1) assignment to relation variables; (2) sequential composition; (3) while-loops; and (4) the introduction of new data elements in terms of tuples of existing ones. All computable queries can be expressed in $while_{new}$. The complexity classes GEN-PSPACE and GEN-PTIME are obtained by putting polynomial space and time restrictions on $while_{new}$ programs. Abiteboul and Vianu illustrated the effect of such restrictions by showing that under a polynomial space restriction, $while_{new}$ programs can no longer check the parity of the cardinality of a set.

The advent of the BGS model thus raises the natural question: how does $\widetilde{\text{C}}$PTIME compare to GEN-PTIME? We will show that $\widetilde{\text{C}}$PTIME is strictly stronger than GEN-PTIME, in the sense that there are classes of structures that can be separated in $\widetilde{\text{C}}$PTIME but not in GEN-PSPACE (and hence neither in GEN-PTIME).[2] We also identify the reason for this inequality:

[1] Abiteboul and Vianu used the name $while^{invent}$ in their paper [6], but use the name $while_{new}$ in their book with Hull [1], so we use the latter name.

[2] A program *separates* two classes $K_0$ and $K_1$ if it outputs 'false' on all structures in

$while_{new}$ only has *tuple-based invention:* new data elements can only be introduced in terms of tuples of existing ones. By repeated application of tuple-based invention one can construct arbitrary lists. BGS, on the other hand, allowing the construction of arbitrary sets, also has a form of *set-based invention.* In the absence of an order on the data elements, it is impossible to simulate sets (which are unordered) using lists (which are ordered) without introducing a lot of duplication.

Our result should be correctly compared to what is known from the theory of object-creating query languages. It is already known [18] that set-based invention cannot be expressed in $while_{new}$. However, this is a statement about object-creating queries where invention is not merely a tool to give more power to query languages, but where we really want to see the new data elements in the result of the query. When only considering standard domain-preserving, or even just boolean queries, set-based invention seemed less relevant because for such queries $while_{new}$ is already complete. Our results show that set-based invention is still relevant, but we have to take complexity into account to see it.

When $while_{new}$ is extended with set-based invention, we show that the language obtained, denoted by $while^{sets}_{new}$, becomes polynomial-time equivalent with BGS (in a sense that will be made precise). Our work is thus related to the update language detTL for relational databases, introduced by Abiteboul and Vianu [3, 5]. Some of the spirit of the ASM model (of which BGS is an instance) is clearly present in detTL, and the equivalence between detTL and $while_{new}$ seems to go without saying.[3] New to our result are the programming with sets and the added focus on polynomial time.

We conclude this introduction by mentioning some other related work. The very first computationally complete query language was QL, introduced by Chandra and Harel [8]. Because QL can be simulated in $while_{new}$ with only a polynomial time overhead [6, 19], our negative result concerning $while_{new}$ applies as well to QL. We also should note that the well-known object-creating query language IQL, introduced by Abiteboul and Kanellakis [2], was set in a complex-object data model with set values, where the distinction between tuples and sets is blurred as one can always have a tuple with a set as a component. Indeed, IQL is polynomial-time equivalent to $while^{sets}_{new}$ [19] and thus also to BGS. Finally, we point out that interest in object creation in query languages has recently resurged in the context of Web

$K_0$ and 'true' on all structures in $K_1$.

[3] To witness, in their book with Hull [1], Abiteboul and Vianu refer to their paper on detTL [5] as the original source for the language $while_{new}$, although no language in the style of $while_{new}$ is discussed in that paper.

databases [11]. Current proposals in this field introduce new data elements by constructing terms, and thus essentially employ tuple-based invention.

## 2 Preliminaries

A relational database scheme is modeled by a finite relational vocabulary in the sense of mathematical logic [10]. So, a scheme is a finite set of relation names with associated arities. A relational database over a scheme $\Upsilon$ is modeled by a finite structure $B$ over $\Upsilon$, i.e., a finite domain $D$ and, for each relation name $R \in \Upsilon$, a relation $R^B \subseteq D^r$, where $r$ is the arity of $R$. The reader is assumed to be familiar with the syntax of first-order logic formulas over $\Upsilon$, and the notion of truth of a formula $\varphi$ in a structure $B$.

We next briefly describe the languages $while_{new}$, $while^{sets}_{new}$, and BGS. For full details we refer to the literature [1, 7, 19].

### 2.1 The language $while_{new}$

An *FO statement* is any expression of the form

$$X := \{(x_1, \ldots, x_j) \mid \varphi\}$$

where $X$ is a $j$-ary relation name, and $\varphi(x_1, \ldots, x_j)$ is a first-order formula. A *tuple-new statement* is any expression of the form

$$Y := \textbf{tup-new}\{(x_1, \ldots, x_j) \mid \varphi\}$$

where $Y$ is a relation name of arity $j+1$, and $\varphi$ is as before.

*Programs* in the language $while_{new}$ are now defined as follows: FO statements and tuple-new statements are programs; if $\Pi_1$ and $\Pi_2$ are programs, then so is their composition $\Pi_1;\Pi_2$; and if $\Pi$ is a program and $\varphi$ is a first-order sentence, then the while-loop **while** $\varphi$ **do** $\Pi$ **od** is a program.

Let $\Pi$ be a program, let $\Upsilon$ be the vocabulary consisting of all the relation names mentioned in $\Pi$, and let $A$ be a finite $\Upsilon$-structure. The *result of applying* $\Pi$ *to* $A$, denoted by $\Pi(A)$, is the $\Upsilon$-structure defined as follows:

- If $\Pi$ is the FO statement $X := \{(x_1, \ldots, x_j) \mid \varphi\}$, then $\Pi(A)$ equals $A$ except for the interpretation of $X$, which is replaced by

$$\{(a_1, \ldots, a_j) \in A^j \mid A \models \varphi(a_1, \ldots, a_j)\}. \qquad (*)$$

- If $\Pi$ is the tuple-new statement $Y := \textbf{tup-new}\{(x_1, \ldots, x_j) \mid \varphi\}$, then $\Pi(A)$ equals $A$ in the interpretation of every relation name other than $Y$. The domain of $\Pi(A)$ is that of $A$, extended with as many new elements as there are tuples in the above set $(*)$. Let $\iota$ be an arbitrary bijection between the set $(*)$ and these new elements. Then the interpretation of $Y$ in $\Pi(A)$ is defined as

$$\{(\bar{a}, \iota(\bar{a})) \mid A \models \varphi(\bar{a})\}.$$

- If $\Pi$ is of the form $\Pi_1; \Pi_2$ then $\Pi(A)$ equals $\Pi_2(\Pi_1(A))$.

- If $\Pi$ is of the form **while** $\varphi$ **do** $\bar{\Pi}$ **od**, then $\Pi(A)$ equals $\bar{\Pi}^n(A)$, where $n$ is the smallest natural number such that $\bar{\Pi}^n(A) \not\models \varphi$. If such a number does not exist, then $\Pi(A)$ is undefined (the while-loop does not terminate).

By the semantics of tuple-new statements (second item), $\Pi(A)$ is clearly defined up to $A$-isomorphism only (isomorphisms that leave $A$ pointwise fixed). This is OK, because the particular choice of the newly invented domain elements really does not matter to us. When doing a complexity analysis, we will assume that the domain of $A$ is an initial segment of the natural numbers, and that a tuple-new statement simply extends this initial segment.

When $\Upsilon_0$ is a subset of $\Upsilon$, and $A$ is an $\Upsilon_0$-structure, we can view $A$ also as an $\Upsilon$-structure by setting $A(X)$ empty for every relation name $X$ in $\Upsilon$ not in $\Upsilon_0$. In this way we can also talk about $\Pi(A)$. This convention formalizes the intuition of initializing relation names not part of the vocabulary of the input structure to the empty set. These relation names are used by the program as variables to do its computation and to contain its final output.

## 2.2 The language *while*

The sublanguage obtained from $while_{new}$ by disallowing tuple-new statements is called *while* and has been extensively studied [1, 9]. In finite model theory, the language *while* is better known under the equivalent form of first-order logic extended with the partial fixpoint operator [4].

## 2.3 The language $while^{sets}_{new}$

A *set-new statement* is an expression of the form

$$Y := \textbf{set-new}\{(x, y) \mid \varphi\},$$

where $Y$ is a binary relation name, and $\varphi(x, y)$ is a first-order formula.

The result $\Pi(A)$ of applying this set-new statement $\Pi$ to a structure $A$, equals $A$ in the interpretations of every relation name other than $Y$. In order to define the domain of $\Pi(A)$ and its interpretation of $Y$, consider the binary relation

$$S = \{(a, b) \in A^2 \mid A \models \varphi(a, b)\}.$$

We can view this relation as a set-valued function in the canonical way: for any $a$ in the first column of $S$, $S(a) := \{b \mid (a, b) \in S\}$.[4] Now the domain of $\Pi(A)$ is that of $A$, extended with as many new elements as there are *different* sets in the range of $S$. Let $\iota$ be an arbitrary bijection between the range of $S$ and these new elements. Then the interpretation of $Y$ in $\Pi(A)$ is defined as

$$\{(a, \iota(S(a))) \mid \exists b : S(a, b)\}.$$

For example, the result of applying

$$Y := \textbf{set-new}\{(x, y) \mid E(x, y)\}$$

to the structure with domain $\{1, 2, 3\}$ where $E$ equals

$$\{(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3)\},$$

is the structure with domain $\{1, 2, 3, 4, 5\}$ where $Y$ equals

$$\{(1, 4), (2, 4), (3, 5)\}.$$

By adding set-new statements to the language $while_{new}$, we obtain the language $while_{new}^{sets}$.

## 2.4 The BGS model

BGS takes a functional point of view: computing means updating the values of certain user-defined, named, "dynamic" functions at various arguments. Arguments and values can be elements of the domain $D$ of the input structure, as well as hereditarily finite sets built over $D$ during the execution of the program. Formally, the set $\mathrm{HF}(D)$ of *hereditarily finite sets over* $D$ is the smallest set such that if $x_1, \ldots, x_n$ are in $D \cup \mathrm{HF}(D)$, then $\{x_1, \ldots, x_n\}$ is in $\mathrm{HF}(D)$. Every dynamic function name has an associated arity $r$, and thus has, at any stage of the computation, an interpretation (which can be updated in later stages) as a function from $(D \cup \mathrm{HF}(D))^r$ to $D \cup \mathrm{HF}(D)$.

[4] In SQL terminology this corresponds to grouping by the first column.

The *extent* of such a function $f$ is the set $\{(\bar{x}, f(\bar{x})) \mid \bar{x} \in (D \cup \mathrm{HF}(D))^r$ and $f(\bar{x}) \neq \emptyset\}$. At any stage of the computation, the extent of the interpretation of any dynamic function will be finite.

A number of *static* functions, which cannot be updated, are predefined: The relations of the input structure are given as boolean functions. The usual logical constants[5] and functions (true, false, and, or, not, equality) are provided. Finally, some set-theoretic constants and functions are provided: the empty set; the input domain; set membership; set union; singleton extraction, and pairing. The input domain is called 'Atoms'. Union is unary, working on a set of sets.

*Terms* can now be built up from variables, constants, functions, and the set constructor $\{t : v \in r : g\}$, where $v$ is a variable that does not occur free in term $r$ but can occur free in term $t$ and boolean term $g$. Variable $v$ becomes bound by the set constructor. The semantics is the obvious one of $\{t : v \in r \text{ and } g\}$.

Finally, *rules* express transitions between states by updating the dynamic functions. *Elementary update rules* are of the form $f(t_1, \ldots, t_j) := t_0$, where $f$ is a dynamic function name (of arity $j$) and $t_1$, ..., $t_j$ are terms. The semantics is obvious. From elementary update rules more complex rules can be built by conditionals and parallel composition. More specifically:

- If $g$ is a boolean term and $R_1$ and $R_2$ are rules, then so is **if** $g$ **then** $R_1$ **else** $R_2$ **endif**, again with the obvious semantics.
- If $v$ is a variable, $r$ is a term in which $v$ does not occur free, and $R_0$ is a rule in which $v$ can occur free, then **forall** $v \in r$ **do** $R_0$ **enddo** is a rule in which $v$ becomes bound. The semantics is to perform $R_0$ in parallel for all $v \in r$, except if this yields conflicting updates in which case we do nothing.

A *BGS program* now is simply a rule without free variables. A program $\Pi$ is started in the initial state, where all dynamic functions have the empty extent, and all static functions are initialized by the input structure $I$. In a run of the program, successive states are computed, until the dynamic boolean constant 'Halt' (which is present in all programs) becomes true. The final state is then the result $\Pi(I)$. As with $while_{new}$ programs, a BGS program may not terminate on some inputs.

[5] As usual, constants are viewed as zero-ary functions.

$Path := \textbf{tup-new}\{(x, y) \mid x = y\}$;
$Ref := \{(p, x) \mid \exists y\, Path(x, y, p)\}$;
$Frontier := \{(x, y, p, z) \mid Path(x, y, p) \wedge E(x, z) \wedge z \neq x\}$;
**while** $Frontier \neq \emptyset$ **do**
  $X := \textbf{tup-new}\{(x, y, p, z) \mid Frontier(x, y, p, z)\}$;
  $Path := \{(x, y, q) \mid Path(x, y, q) \vee \exists p \exists z\, X(x, y, p, z, q)\}$;
  $Ref := \{(q, z) \mid Ref(q, z) \vee \exists x \exists y \exists p\, X(x, y, p, z, q)\}$;
  $Child := \{(p, q) \mid Child(p, q) \vee \exists x \exists y \exists z\, X(x, y, p, z, q)\}$;
  $Frontier := \{(x, y, q, z') \mid \exists p \exists z (X(x, y, p, z, q) \wedge z \neq y \wedge E(z, z'))\}$
**od**;
$Path := \{(x, y, p) \mid \exists p' (Path(x, y, p') \wedge Ref(p', y))\}$.

Figure 1: A $while_{new}$ program computing all-pairs shortest paths.

## 2.5 Examples

An example of a $while_{new}$ program is shown in Figure 1, and an example of a BGS program is shown in Figure 2. Both example programs work on directed graphs, modeled as structures whose domain is the set of nodes and which have a binary relation $E$ holding the edges. Both programs compute, for all pairs of nodes $(x, y)$, all shortest paths from $x$ to $y$. They do not follow exactly the same algorithm; the $while_{new}$ program does a single-source single-target search in parallel for all source-target pairs $(x, y)$, while the BGS program does a single-source all-targets search in parallel for all sources $x$.

In the $while_{new}$ program, a path $x_1 \ldots x_n$ is represented by invented values $p_1, \ldots, p_n$ such that the following relations, defined by the program, hold: $Path(x_1, x_n, p_i)$ for $i = 1, \ldots, n$; $Ref(p_i, x_i)$ for $i = 1, \ldots, n$; and $Child(p_i, p_{i+1})$ for $i = 1, \ldots, n-1$. The relations *Frontier* and $X$ used in the program are auxiliary variables.

In the BGS program, a path $x_1 \ldots x_n$ is represented by a pair $(x_1 \ldots x_{n-1}, x_n)$, where the $x_1 \ldots x_{n-1}$ is again represented by a pair, recursively.[6] The base case $n = 1$ is represented by a singleton $\{x_1\}$. The program updates a dynamic binary function *Paths* such that $Paths(x, y)$ equals the set of shortest paths from $x$ to $y$. Other dynamic functions and constants used by the program to aid the computation are *Mode*, *Reached*, *Frontier*, and *Old_Frontier*. The comma between rules denotes parallel composition, and is a shorthand for a trivial `forall do` construct. The natural numbers 0, 1, and 2 assigned to *Mode* are in $\mathrm{HF}(D)$ by their definition as von Neumann

[6] Recall that ordered pairs $(x, y)$ are by definition in $\mathrm{HF}(D)$, as $\{\{x\}, \{x, y\}\}$ [15].

```
if Mode = 0 then
  forall x ∈ Atoms do
    Reached(x) := {x},
    Paths(x, x) := {{x}},
    Frontier(x) := {x}
  enddo,
  Mode := 1
endif,
if Mode = 1 then
  forall x ∈ Atoms do
    Old_Frontier(x) := Frontier(x),
    Frontier(x) := {y : y ∈ Atoms : y ∉ Reached(x)
                                    and {z : z ∈ Frontier(x) : E(z, y)} ≠ ∅}
  enddo,
  Mode := 2
endif,
if Mode = 2 then
  forall x ∈ Atoms do
    forall y ∈ Frontier(x) do
      Paths(x, y) := {(p, y) :
                      p ∈ ⋃{Paths(x, z) : z ∈ Old_Frontier(x) : E(z, y)} : true}
    enddo,
    Reached(x) := Reached(x) ∪ Frontier(x)
  enddo,
  Halt := ⋃{Frontier(x) : x ∈ Atoms : true} = ∅,
  Mode := 1
endif.
```

Figure 2: A BGS program computing all-pairs shortest paths.

numerals: 0 is the empty set, and $n > 0$ is $\{0, \ldots, n-1\}$, recursively [15]. The numbers 0 and 1 also play the role of the booleans false and true.

## 3 BGS and $while_{new}$ under polynomial time

In this section, we define what it means for two classes of structures over the same vocabulary to be separable in polynomial time by BGS programs, or by $while_{new}$ programs. We then prove that there exists a pair that is separable in polynomial time by a BGS program, but not by any $while_{new}$ program.

During the run of a BGS program on a structure with domain $D$, a certain number of sets in $\mathrm{HF}(D)$ are *activated*, meaning that at some point they appear in the extent of some dynamic function. Elements of active sets are also considered to be active, and this holds recursively. Similarly, during the run of a $while_{new}$ program on a structure, a certain number of new elements are invented. Activated sets and invented elements yield measures of space usage by BGS and $while_{new}$ programs, which are quite rough, but sufficient for our purposes. Equally rough measures of time spent by BGS and $while_{new}$ programs can be defined as follows: the time spent by a BGS program on a structure is the number of times the program is iterated until the halting condition is reached; the time spent by a $while_{new}$ program on a structure is the number of times an FO or tuple-new statement is executed during the run of the program.

In the following two paragraphs fix two disjoint classes $K_0$ and $K_1$ of structures over a common vocabulary.

Let $\Pi$ be a BGS program using a boolean dynamic constant Output for output. We say that $\Pi$ *separates* $K_0$ *from* $K_1$ if for any structure $A \in K_0 \cup K_1$, the value of Output in $\Pi(A)$ is false if $A \in K_0$, and is true if $A \in K_1$. We say that $\Pi$ *separates* $K_0$ *from* $K_1$ *in polynomial time* if moreover, there exist two polynomials $p(n)$ and $q(n)$ such that for any $A \in K_0 \cup K_1$, $\Pi$ runs on $A$ for at most $p(n)$ time, and activates at most $q(n)$ sets, where $n$ is the cardinality of the domain of $A$.

Similarly, let $\Pi$ be a $while_{new}$ program having some relation variable *Output*. We say that $\Pi$ *separates* $K_0$ *from* $K_1$ if $\Pi(A)$ is defined for any structure $A \in K_0 \cup K_1$, and relation *Output* in $\Pi(A)$ is empty if $A \in K_0$, and is not empty if $A \in K_1$. We say that $\Pi$ *separates* $K_0$ *from* $K_1$ *in polynomial time* if moreover, there exist two polynomials $p(n)$ and $q(n)$ such that for any $A \in K_0 \cup K_1$, $\Pi$ runs on $A$ for at most $p(n)$ time, and invents at most $q(n)$ elements, where $n$ is the cardinality of the domain of $A$.

Since we do not care what the programs do on structures outside $K_0$ and $K_1$, the above notion of separation is quite liberal. Still, we will be able to obtain a negative result regarding the separating power of $while_{new}$ in polynomial time. Also, in our definition, it is important to polynomially restrict the space used as well as the time, because in BGS or $while_{new}$ it is possible to use an exponential amount of space even in an only linear amount of time.

We now prove:

**Theorem 1.** *There exist pairs of classes of structures that can be separated in polynomial time by a BGS program, but not by a* $while_{new}$ *program.*

Consider the vocabulary consisting of a single relation name $P$, which is unary. For any natural number $n$, define a structure $I_n$ over this vocabulary as follows. The domain of $I_n$ consists of $2^n$ elements. Exactly $n$ of these satisfy the predicate $P$. The pair now for which we are going to prove the theorem was already considered by Blass, Gurevich and Shelah [7] and is the following: $K_0 = \{I_n \mid n \text{ even}\}$, and $K_1 = \{I_n \mid n \text{ odd}\}$. We can easily separate $K_0$ from $K_1$ by a BGS program in polynomial time: the program generates all subsets of $P$ with even cardinality (which is in polynomial time because the cardinality of the input domain is $2^n$), and then checks whether $P$ itself was generated.

We will actually show that $K_0$ cannot be separated from $K_1$ by any $while_{new}$ program that can invent only a polynomial number of elements; the time spent by the program will be irrelevant.

The following interpretation of tuple-new statements as list constructions will provide insight. *Lists* over some domain $D$ are inductively defined as follows:

- The *empty list* () is a list;
- If $l_1, \ldots, l_j$ are elements of $D$ or lists, then $(l_1, \ldots, l_j)$ is also a list.

Adopting BGS terminology, we will frequently refer to domain elements as "atoms," to distinguish them from lists built over the domain.

Now recall the semantics of a tuple-new statement

$$Y := \textbf{tup-new}\{(x_1, \ldots, x_j) \mid \varphi(x_1, \ldots, x_j)\}$$

on a structure $A$, which assigns to relation name $Y$ the relation

$$\{(a_1, \ldots, a_j, \iota(a_1, \ldots, a_j)) \mid A \models \varphi(a_1, \ldots, a_j)\}$$

for some bijection $\iota$ from the tuples over $A$ satisfying $\varphi$ to new elements. We fix this bijection $\iota$ uniformly as follows. Assume it is the $m$th time we are performing a tuple-new statement in the execution of the program. Then $\iota(a_1, \ldots, a_j)$ is defined to be the following list:

$$((a_1, \ldots, a_j), \lambda_m),$$

where $\lambda_1$ is the empty list, and $\lambda_m$ with $m > 1$ is the list $(\lambda_{m-1})$. The reason for pairing with $\lambda_m$ is to ensure that, if a same tuple $\bar{a}$ participates in two different executions of an invention statement, the second time a different element will be invented for it than the first time.

We thus see that element invention can be viewed as list construction. But this implies that element invention can be discarded altogether, if the lists to be constructed are already present in the structure. More precisely, we can expand an input structure $I$ with a (possibly infinite) collection $\Lambda$ of lists over its atoms as follows. The lists, and all the lists occurring in them, are added to the domain of $I$. To represent their internal structure, we provide a unary relation *Empty* and two binary relations *Head* and *Tail*. Relation *Empty* has just one element, the empty list. Relation *Head* consists of all pairs $(x, y)$ where $x = (x_1, \ldots, x_j)$ is a nonempty list in $\Lambda$ and $y$ equals $x_1$. Relation *Tail* consists of all pairs $(x, y)$ where $x$ is as above and $y$ equals $(x_2, \ldots, x_j)$ (if $j = 1$ then $y$ equals the empty list).

A structure $I$ expanded with a collection $\Lambda$ of lists in the way just described is denoted by $(I, \Lambda)$. We have the following simulation of $while_{new}$ programs by *while* programs:

**Lemma 1.** *For every $while_{new}$ program $\Pi$ over a vocabulary $\Upsilon$ there exists a while program $\Pi'$ over the expanded vocabulary $\Upsilon \cup \{Empty, Head, Tail\}$ with the following property. Let $I$ be any input structure such that $\Pi(I)$ is defined, and let $\Lambda$ be any collection of lists over the atoms of $I$ that includes all lists invented during the execution of $\Pi$ on $I$. Then $\Pi'(I, \Lambda)$ is defined, and equals $\Pi(I)$ on every relation name of $\Upsilon$.*

*Proof.* The desired program $\Pi'$ is identical to $\Pi$ apart from the fact that every tuple-new statement is replaced by two FO statements that simulate it, using the lists that are already present. For example, a statement $Y :=$ **tup-new**$\{(x, y) \mid \varphi\}$ is simulated by the following two FO statements:

$Y := \{(x, y, z) \mid \varphi(x, y) \wedge \exists l_1 \exists c \exists l_2 \exists e(Head(z, l_1) \wedge Tail(z, c) \wedge Counter(c) \wedge Head(l_1, x) \wedge Tail(l_1, l_2) \wedge Head(l_2, y) \wedge Tail(l_2, e) \wedge Empty(e))\}$;

$Counter := \{(c') \mid \exists c \exists e(Counter(c) \wedge Head(c', c) \wedge Tail(c', e) \wedge Empty(e))\}$.

In the beginning of $\Pi'$ relation *Counter* is initialized to *Empty*. □

We note one more lemma regarding $while_{new}$ programs (still under the list construction view of element invention), which is straightforwardly verified:

**Lemma 2.** *If $\Pi$ is an arbitrary $while_{new}$ program, $I$ is an arbitrary input structure, $\alpha$ is an arbitrary automorphism of $I$, and $x$ is an element in $\Pi(I)$, then also $\alpha(x)$ is an element in $\Pi(I)$.*

Note that $\alpha$ is defined on the atoms of $I$ and that it is applied to tuples and lists over these atoms in the canonical way.

We are now ready for the following:

*Proof of Theorem 1.* Recall the classes $K_0 = \{I_n \mid n \text{ even}\}$ and $K_1 = \{I_n \mid n \text{ odd}\}$ introduced after the statement of the theorem. Suppose, for the sake of arriving at a contradiction, that $\Pi$ is a $while_{new}$-program separating these classes and that $p$ is a polynomial such that $\Pi$ invents at most $p(2^n)$ elements when run on any input structure $I_n$ (recall that the cardinality of the domain of $I_n$ is $2^n$). If we take the number $d$ to be one more than the degree of $p$, the program will invent at most $2^{dn}$ elements when run on $I_n$, for sufficiently large $n$.

When considering a structure $I_n$, let us refer to the elements in $P$ as "colored." Note that any permutation of the colored domain elements, as well as any permutation of the uncolored ones, is an automorphism of $I_n$. Consider an invented list $x$ in $\Pi(I_n)$. Let $c$ ($u$) be the number of different colored (uncolored) atoms occurring in $x$. The number of different images of $x$ under automorphisms of $I_n$ equals

$$\frac{n!}{(n-c)!} \cdot \frac{(2^n-n)!}{(2^n-n-u)!}.$$

Denoting $c+u$ by $t$, the above number is, for sufficiently large $n$, larger than $n!/(n-\min\{t,n\})!$. Still the total number of invented elements cannot be more than $2^{dn}$. Hence, with $\mu(n)$ defined to be the largest natural number in $\{1,\ldots,n\}$ such that

$$\frac{n!}{(n-\mu(n))!} \leq 2^{dn},$$

we can conclude by Lemma 2 that *in any invented list, the number of different atoms occurring in it is at most $\mu(n)$.*

Let $\Pi'$ be the *while* program simulating $\Pi$, as given by Lemma 1. So, for any $n$, the output of $\Pi$ on $I_n$ is the same as the output of $\Pi'$ on the expansion of $I_n$ with all lists over its atoms in which at most $\mu(n)$ different

atoms occur; denote this expansion by $I_n^*$. In particular, since $\Pi$ separates $K_0$ from $K_1$, the output of $\Pi'$ on $I_n^*$ is different from that on $I_{n+1}^*$. It is well known that every *while* program can be equivalently expressed by a formula in the infinitary logic $L_{\infty\omega}^{\omega}$ [17, 9]. Let $k$ be the number of variables needed to express $\Pi'$ in $L_{\infty\omega}^{k}$. If we can now show that for certain $n$, $I_n^*$ and $I_{n+1}^*$ are indistinguishable in $L_{\infty\omega}^{k}$, we have arrived at the desired contradiction and completed the proof.

Take $n$ sufficiently large so that $k\mu(n+1) \leq n$.[7] We will actually show that $I_{n+1}^*$ is indistinguishable with $k$ variables from $J$, where $J$ is the expansion of $I_n$ with all lists over its atoms in which at most $\mu(n+1)$ different atoms occur (rather than $\mu(n)$). This is OK, because by Lemma 1, $\Pi'$ has the same output on $I_n^*$ and $J$. We show a winning strategy for the duplicator in the well-known $k$-pebble game [17, 9] on $I_{n+1}^*$ and $J$. We abbreviate $\mu(n+1)$ to $\mu$.

The duplicator maintains, as part of his strategy, a partial bijection $f$ from atoms of $J$ to atoms of $I_{n+1}^*$. Initially, $f$ is empty, and at any time the domain of $f$ consists of at most $k\mu$ atoms. This number comes from the fact that there are $k$ pebbles, and each pebble can be placed either on an atom, or on a list in which at most $\mu$ different atoms occur. Assume the spoiler places pebble number $i$ on an element $x$ in $J$, thereby effectively choosing up to $\mu$ atoms from $J$ at the same time. Let $x'$ be the element (if any) on which pebble $i$ was placed previously at $J$. The duplicator begins by removing from $f$ all pairs involving atoms from $J$ that appear in $x'$ but not in any other currently pebbled element of $J$. He then updates $f$ by matching the newly chosen atoms, i.e., those appearing in $x$ but not yet in $f$, with arbitrary atoms from $I_{n+1}^*$, taking care only that colored atoms are matched with colored ones (and uncolored with uncolored). The inequality $k\mu \leq n$ guarantees that this is possible. Finally the duplicator responds to the spoiler's move by placing pebble number $i$ at $I_{n+1}^*$ on element $f(x)$. If the spoiler's move was at structure $I_{n+1}^*$ rather than at $J$, everything is symmetric. The partial isomorphism that the duplicator thus preserves by this strategy is simply $f$ itself, canonically applied to the pebbled elements. □

Because of the equivalence between $while_{new}$ and the generic machine model of Abiteboul and Vianu [6], Theorem 1 implies that generic machines are strictly weaker than BGS in the context of polynomial time computation. This result corrects a tentative claim ('the simulation in the reverse direction

[7] It is easy to see that it is indeed impossible for $\mu(n+1)$ to be greater than $n/k$ for $n$ sufficiently large.

can, it seems, be carried out using the "form and matter" considerations in Section 9') near the end of Section 1 of the BGS paper [7]. The form and matter considerations mentioned there involve tuples rather than sets as "matter" and therefore run into the same duplication problem as $while_{new}$.

# 4 Polynomial time equivalence of BGS and $while^{sets}_{new}$

In this section, we formally define notions of polynomial-time simulation of BGS programs by $while^{sets}_{new}$ programs, and vice versa, and show that such simulations exist.

## 4.1 Simulating $while^{sets}_{new}$ in BGS

To simulate $while^{sets}_{new}$ in BGS, we need some way to represent elements that are invented by a $while^{sets}_{new}$ program by hereditarily finite sets that can be constructed by a BGS program. For elements invented by a **tup-new** statement, we already did this in the previous section, where we described a list-construction semantics for **tup-new**.[8] So it remains to describe a set-construction semantics for **set-new**.

To this end, recall the semantics of a set-new statement $Y :=$ **set-new** $S$ on a structure $A$ (where $S$ is a binary relation on $A$ defined by some first-order formula), which assigns to relation name $Y$ the relation $\{(a, \iota(\varphi(a))) \mid \exists b : S(a,b)\}$ for some bijection $\iota$ from the range of $S$ (viewed as a set-valued function) to new elements. We fix this bijection $\iota$ uniformly as follows. Assume it is the $m$th time we are performing a tuple-new or set-new statement in the execution of the program. Then $\iota(S(a))$ is defined to be the pair

$$(S(a), \lambda_m),$$

where $\lambda_m$ is as defined in the previous section.

We now say that a BGS program $\Pi'$ *simulates* a $while^{sets}_{new}$ program $\Pi$ if for every input structure $I$, if $\Pi(I)$ is defined then so is $\Pi'(I)$, and for every relation variable $X$ of $\Pi$, say of arity $r$, there is an $r$-ary boolean dynamic function $\widehat{X}$ of $\Pi'$, such that the tuples in $X$ in $\Pi(I)$ are exactly the tuples at which $\widehat{X}$ is true in $\Pi'(I)$. Moreover, we say that the simulation is *linear-step, polynomial-space* if there exists a constant $c$ and a polynomial $p$ such that for every input structure $I$ where $\Pi(I)$ is defined, the following holds. Let the time for which $\Pi$ runs on $I$ be $t$, and let the number of invented elements

[8] Lists are special kinds of sets: a list of length $n$ is a mapping from $\{1, \ldots, n\}$ to the set of members of the list, and a mapping is a set of ordered pairs.

during the run be $s$. Then $\Pi'$ runs on $I$ for at most $ct$ time, activating at most $p(n+s)$ sets, where $n$ is the cardinality of the domain of $I$.

Here, in close analogy to what we defined for $while_{new}$ programs at the beginning of Section 3, we define the time spent by a $while^{sets}_{new}$ program on a structure as the number of times an FO, tuple-new, or set-new statement is executed during the run of the program.

Note that, while we allow a polynomial overhead in space usage, we allow only a linear overhead in the running time of the simulation. A weaker notion of polynomial time simulation could be defined, allowing a polynomial overhead also for running time, but we will not need to consider this weaker notion as we will be able to obtain positive results for our stronger notion.

We show:

**Theorem 2.** *Every $while^{sets}_{new}$ program can be linear-step, polynomial-space simulated by a BGS program.*

*Proof.* Let $\Pi$ be a $while^{sets}_{new}$ program. An *instruction* in $\Pi$ is any FO, tuple-new, or set-new statement occurring in $\Pi$, or any expression of the form **while** $\varphi$ **do** occurring in $\Pi$. The latter kind of instruction is called a *test instruction*. We number the instructions, so that no instruction gets the number 0. (Recall that we can use natural numbers, represented as von Neumann numeral, in BGS programs.) The first and last instructions of $\Pi$ are defined in the obvious way; in particular, if the first (last) statement of $\Pi$ is a while-loop, then the test instruction of that loop is considered to be the first (last) instruction. Let *start* be the first instruction, and let *last* be the last instruction.[9]

Also, every instruction $i \neq last$ that is not a test instruction has a natural "next" instruction $next(i)$; in particular, if $i$ is the last instruction of the body of a while-loop then $next(i)$ is the test instruction of that loop. If $i = last$, we still define $next(i)$, as some number *finish* which is not the number of any instruction. Every test instruction $i \neq last$ has two natural next instructions, $next^+(i)$ and $next^-(i)$: the first when the test succeeds, the second when the test fails. Note that even if a test instruction $i$ equals *last*, it still has a $next^+(i)$. We define $next^-(last)$ to be again *finish*.

We will describe, for each instruction $i$ of $\Pi$, a BGS rule $\rho(i)$. The desired BGS program $\Pi'$ then simply is the parallel composition of all these rules, together with the following initialization and finish rules:

**if** $Mode = 0$ **then**

[9] If the whole program is one while-loop, the test instruction of that loop is at the same time the first and the last instruction.

*Instruction* := *start*,
*Adom* := Atoms,
*Mode* := 1
**endif**,
**if** *Instruction* = *finish* **then**
Halt := 1
**endif**.

The dynamic constant *Instruction* will be used to keep track of which instruction to simulate, and the dynamic constant *Adom* will be used to hold the current domain of elements in course of the computation of $\Pi$ (recall that the domain can expand by the execution of tuple-new and set-new statements).

First, suppose instruction $i$ is an FO statement $X := \{(x_1, \ldots, x_j) \mid \varphi\}$. The first-order formula $\varphi(x_1, \ldots, x_j)$ is to be evaluated by letting the quantifiers in it range over the current domain, stored in *Adom*. It is already known [7] that any such a first-order condition can be expressed by a boolean BGS term, which, when evaluated, will activate a number of sets that is bounded by a polynomial in the cardinality of *Adom*. Denoting the BGS term for formula $\varphi$ simply by $\varphi$ itself, the rule $\rho(i)$ is now as follows:

**if** *Instruction* = $i$ **then**
**forall** $x_1 \in$ *Adom* **do**
$\ddots$
**forall** $x_j \in$ *Adom* **do**
$\widehat{X}(x_1, \ldots, x_j) := \varphi$
**enddo**
$\iddots$
**enddo**,
*Instruction* := *next*($i$)
**endif**.

Next, suppose instruction $i$ is a tuple-new statement $Y :=$ **tup-new**$\{(x_1, \ldots, x_j) \mid \varphi\}$. In addition to assigning to function $\widehat{Y}$, we now also have to expand *Adom* with the invented elements, i.e., all elements $((a_1, \ldots, a_j), \lambda_m)$ where $(a_1, \ldots, a_j)$ satisfies $\varphi(x_1, \ldots, x_j)$. We keep track of the current value of $\lambda_m$ using a dynamic constant *Lambda*. To compute the invented elements by a BGS term, we distinguish two cases:

- If $j = 0$, define the term $t_1$ as follows:

$$t_1 := \{(\emptyset, \mathit{Lambda}) : v \in \{1\} : \varphi\},$$

where $v$ is just a dummy variable.

- If $j \geq 1$, define the terms $t_i$ for $i = 1, \ldots, j$ by downward induction as follows:

$$t_j := \{((x_1, \ldots, x_j), \mathit{Lambda}) : x_j \in \mathit{Adom} : \varphi(x_1, \ldots, x_j)\}$$

and for $i < j$,

$$t_i := \bigcup \{t_{i+1} : x_i \in \mathit{Adom} : \text{true}\}.$$

Now the rule $\rho(i)$ is defined as follows:

**if** *Instruction* $= i$ **then**
  **forall** $x_1 \in \mathit{Adom} \cup t_1$ **do**
    $\ddots$
      **forall** $x_j \in \mathit{Adom} \cup t_1$ **do**
        **forall** $y \in \mathit{Adom} \cup t_1$ **do**
          $\widehat{Y}(x_1, \ldots, x_j, y) := \varphi$ and $y = ((x_1, \ldots, x_j), \mathit{Lambda})$
        **enddo**
      **enddo**
    $\iddots$
  **enddo**,
  $\mathit{Adom} := \mathit{Adom} \cup t_1$,
  $\mathit{Lambda} := \{\mathit{Lambda}\}$,
  *Instruction* $:= \mathit{next}(i)$
**endif**.

Next, suppose instruction $i$ is a set-new statement $Y :=$ **set-new**$\{(x, y) \mid \varphi\}$. Define the following auxiliary terms:

$$t(x) := \{y : y \in \mathit{Adom} : \varphi(x, y)\};$$

$$t := \{(t(x), \mathit{Lambda}) : x \in \mathit{Adom} : t(x) \neq \emptyset\}.$$

The rule $\rho(i)$ is as follows:

**if** *Instruction* $= i$ **then**
  **forall** $x \in \mathit{Adom} \cup t$ **do**
    **forall** $z \in \mathit{Adom} \cup t$ **do**
      $\widehat{Y}(x, z) := t(x) \neq \emptyset$ and $z = (t(x), \mathit{Lambda})$
    **enddo**
  **enddo**,

$Adom := Adom \cup t$,
$Lambda := \{Lambda\}$,
$Instruction := next(i)$
**endif**.

Finally, suppose instruction $i$ is a test instruction **while** $\varphi$ **do**. Then the rule $\rho(i)$ is the following:

**if** $Instruction = i$ **then**
  **if** $\varphi$ **then**
    $Instruction := next^{+}(i)$
  **else**
    $Instruction := next^{-}(i)$
  **endif**
**endif**.

We hope that our description of this BGS program $\Pi'$ has made it evident that $\Pi'$ correctly simulates $\Pi$.

Moreover, if the total number of instructions executed in the run of $\Pi$ on a structure $A$ equals $t$, then $\Pi'$ will iterate $t + 2$ times on $A$ (the extra two iterations are to initialize and to finish). Now recall that we actually defined the time spent by $\Pi$ on $A$ as the total number of times an FO, tuple-new, or set-new statement is executed; in other words, we ignored the test instructions. However, since we can execute only a constant number of test instructions without encountering a non-test instruction, ignoring the test instructions has at most the effect of dividing by a constant.

Regarding space usage, each iteration of $\Pi'$ activates a number of sets that is bounded by a polynomial in the cardinality of $Adom$. This cardinality is at most $n + s$, where $n$ is the cardinality of the domain of $A$, and $s$ is the number of elements invented while running $\Pi$ on $A$.

Hence, the simulation is linear-step, polynomial-space, and the proof is complete. □

## 4.2 Simulating BGS in $while^{sets}_{new}$

To simulate BGS in $while^{sets}_{new}$, we need some way to represent hereditarily finite sets by invented elements. To this end, we observe that for any finite domain $D$, the structure $(D \cup \mathrm{HF}(D), \in)$ is an (infinite) directed acyclic graph. At any stage in the run of a BGS program on a structure with domain $D$, the active sets, together with the elements of $D$, generate a finite subgraph of this graph. The simulating $while^{sets}_{new}$ program will maintain a copy of that subgraph, where the active sets are represented by invented

elements, and the elements of $D$ are represented by themselves. The membership relation $\in$ will be stored in a relation variable *Epsilon*.

We now say that a $while^{sets}_{new}$ program $\Pi'$ *simulates* a BGS program $\Pi$ if for every input structure $I$, if $\Pi(I)$ is defined then so is $\Pi'(I)$, and for every dynamic function name $f$ of $\Pi$, say of arity $r$, there is an $(r+1)$-ary relation variable $\widehat{f}$ of $\Pi'$, such that $\widehat{f}$ in $\Pi'(I)$ equals exactly the extent of $f$ in $\Pi(I)$, under a representation of the active hereditarily finite sets by invented elements as given in relation *Epsilon* in $\Pi'(I)$. Moreover, we say that the simulation is *linear-step, polynomial-space* if there exist a constant $c$ and a polynomial $p$ such that for every input structure $I$ where $\Pi(I)$ is defined, the following holds. Let the time for which $\Pi$ runs on $I$ be $t$, and let the number of sets activated during the run be $s$. Then $\Pi'$ runs on $I$ for at most $ct$ time, inventing at most $p(s)$ elements.[10]

We show:

**Theorem 3.** *Every BGS program can be linear-step, polynomial-space simulated by a* $while^{sets}_{new}$ *program.*

*Proof.* Let $\Pi$ be a BGS program. The simulating $while^{sets}_{new}$ program $\Pi'$ begins with the following initialization part that invents representatives for the sets 0, 1, and Atoms:

$InputDomain := \{(x) \mid \mathbf{true}\}$;
$Zero :=$ **tup-new**$\{() \mid \mathbf{true}\}$;
$One :=$ **tup-new**$\{() \mid \mathbf{true}\}$;
$Epsilon := \{(z, o) \mid Zero(z) \wedge One(o)\}$;
$Atoms :=$ **tup-new**$\{() \mid \mathbf{true}\}$;
$Epsilon := Epsilon \cup \{(i, a) \mid InputDomain(i) \wedge Atoms(a)\}$.

An important part of the program $\Pi'$ is a long sequence of FO, tuple-new, and set-new statements evaluating all occurrences of terms in $\Pi$. We next describe how this can be done. Every occurrence of a term $t$ takes place in a *context*, consisting of a sequence $v_1 \ldots v_k$ of variables which are not bound in or by $t$, but which are bound by a set-constructor term $\{\cdots : v_i \in r_i : \ldots\}$ encompassing $t$, or by a rule **forall** $v_i \in r_i$ **do** $\ldots$ encompassing $t$. We may assume that $\Pi$ does not reuse variables, i.e., that every variable is bound

[10] The reader will have noticed that, while here we require that $\Pi'$ invents at most $p(s)$ elements, in the notion of polynomial-space simulation of $while^{sets}_{new}$ programs by BGS programs as defined in the previous subsection, we allowed the simulating BGS program to activate $p(n+s)$ sets. The reason for this is that, even if a $while^{sets}_{new}$ program $\Pi$ does not invent any new elements (i.e., $s = 0$), a simulating BGS program still needs to activate some sets just to evaluate the first-order formulas used in $\Pi$.

only once. Moreover, we order the context variables $v_1, \ldots, v_k$ top down, so that the context of the occurrence of the term $r_i$ describing the range of $v_i$ is $v_1 \ldots v_{i-1}$. We will compute a $(k+1)$-ary relation $T$ holding all tuples $(a_1, \ldots, a_k, t(a_1, \ldots, a_k))$, where $a_i$ is in the range of $v_i$. We must distinguish between occurrences of the same syntactic term in different contexts, but different occurrences of the same syntactic term in the same context can be treated identically.

We proceed by induction. We may assume that relations $R_1, \ldots, R_k$ for the terms $r_1, \ldots, r_k$ have already been computed. To set the ranges of the context variables, the following formula will be used extensively:

$$\mathit{Ranges}(x_1, \ldots, x_k) := \bigwedge_{i=1}^{k} R_i(x_1, \ldots, x_{i-1}, x_i).$$

It will be convenient to abbreviate $(x_1, \ldots, x_k)$ by $\bar{x}$.

If $t$ is a variable $v_i$, then we write

$$T := \{(\bar{x}, x_i) \mid \mathit{Ranges}(\bar{x})\}.$$

If $t$ is a function term $f(t_1, \ldots, t_j)$, we may assume that relations $T_1, \ldots, T_j$ for the terms $t_1, \ldots, t_j$ have already been computed. We consider the following cases for the function name $f$:

- $f$ is 'true', so $j = 0$. In this case we write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \mathit{One}(t)\}.$$

- $f$ is 'and', so $j = 2$. In this case we write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 \exists t_2 (T_1(\bar{x}, t_1) \wedge T_2(\bar{x}, t_2) \\ \wedge \textbf{if } \mathit{One}(t_1) \wedge \mathit{One}(t_2) \textbf{ then } \mathit{One}(t) \textbf{ else } \mathit{Zero}(t))\}.$$

  Here and below, '**if** $\alpha$ **then** $\beta$ **else** $\gamma$' is an abbreviation for the formula $(\alpha \to \beta) \wedge (\neg\alpha \to \gamma)$.

- $f$ is 'not', so $j = 1$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 (T_1(\bar{x}, t_1) \\ \wedge \textbf{if } \mathit{Zero}(t_1) \textbf{ then } \mathit{One}(t) \textbf{ else } \mathit{Zero}(t))\}.$$

- $f$ is '=', so $j = 2$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 \exists t_2 (T_1(\bar{x}, t_1) \wedge T_2(\bar{x}, t_2) \\ \wedge \textbf{if } t_1 = t_2 \textbf{ then } \mathit{One}(t) \textbf{ else } \mathit{Zero}(t))\}.$$

- $f$ is an input relation name. In this case we write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 \ldots \exists t_j (\bigwedge_{l=1}^{j} T_l(\bar{x}, t_l) \\ \wedge \textbf{if } f(t_1, \ldots, t_j) \textbf{ then } \mathit{One}(t) \textbf{ else } \mathit{Zero}(t))\}.$$

- $f$ is a dynamic function name. Then we write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 \ldots \exists t_j (\bigwedge_{l=1}^{j} T_l(\bar{x}, t_l) \\ \wedge \textbf{if } \exists t' \, \widehat{f}(t_1, \ldots, t_j, t') \textbf{ then } \widehat{f}(t_1, \ldots, t_j, t) \textbf{ else } \mathit{Zero}(t))\}.$$

  Recall that, by definition of simulation, $\widehat{f}$ only stores the extent of $f$, so if there is no value in $\widehat{f}$ for $(t_1, \ldots, t_j)$ this indeed means that $f(t_1, \ldots, t_j) = \emptyset$.

- $f$ is '$\emptyset$', so $j = 0$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \mathit{Zero}(t)\}.$$

- $f$ is 'Atoms', so $j = 0$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \mathit{Atoms}(t)\}.$$

- $f$ is '$\in$', so $j = 2$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 \exists t_2 (T_1(\bar{x}, t_1) \wedge T_2(\bar{x}, t_2) \\ \wedge \textbf{if } \mathit{Epsilon}(t_1, t_2) \textbf{ then } \mathit{One}(t) \textbf{ else } \mathit{Zero}(t))\}.$$

- $f$ is 'TheUnique', for singleton extraction, so $j = 1$. We write

$$T := \{(\bar{x}, t) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t_1 (T_1(\bar{x}, t_1) \\ \wedge \textbf{if } \exists! y \, \mathit{Epsilon}(y, t_1) \textbf{ then } \mathit{Epsilon}(t, t_1) \textbf{ else } \mathit{Zero}(t))\}.$$

  Here, $\exists! y$ is an expression for "there exists a unique $y$."

- $f$ is '$\bigcup$', so $j = 1$. This is the first of three cases where we have to construct sets, so let us explain in some detail the strategy we will follow. A similar strategy will be followed in the other two cases where we have to construct sets.

  1. Using **tup-new**, we invent for every value of $t_1$ a new element to represent $\bigcup t_1$, in an auxiliary relation $X_t$.
  2. We associate the invented representatives to their members in an auxiliary binary relation $E$.
  3. Using **set-new** and an auxiliary relation *Unique*, we make sure that sets have unique representatives. (The value of $t_1$ could be the same for different values of the context variables. Moreover, even for different values $V_1$ and $V_2$ of $t_1$, we could have $\bigcup V_1 = \bigcup V_2$.)
  4. Elements representing a set that was already active are replaced by the already existing representative for that set (a particular example is the empty set), using an auxiliary relation *Replace*.
  5. The representatives of newly activated sets are incorporated in the *Epsilon* relation.
  6. Finally, relation $T$ is set.

  Concretely in $while^{sets}_{new}$, we write:

$$
\begin{aligned}
&X_t := \textbf{tup-new}\{(\bar{x}) \mid \mathit{Ranges}(\bar{x})\};\\
&E := \{(y,z) \mid \exists \bar{x} \exists t_1 \exists s (X_t(\bar{x},z) \wedge T_1(\bar{x},t_1)\\
&\qquad\qquad \wedge \mathit{Epsilon}(s,t_1) \wedge \mathit{Epsilon}(y,s))\};\\
&\mathit{Unique} := \textbf{set-new}\{(z,y) \mid E(y,z)\};\\
&\mathit{Replace} := \{(z,z') \mid (\neg \exists y\, E(y,z) \wedge \mathit{Zero}(z'))\\
&\qquad\qquad \vee (\exists y\, \mathit{Epsilon}(y,z') \wedge \forall y(\mathit{Epsilon}(y,z') \leftrightarrow E(y,z)))\};\\
&\mathit{Epsilon} := \mathit{Epsilon}\\
&\qquad \cup \{(y,u) \mid \exists z (E(y,z) \wedge \neg \exists z'\, \mathit{Replace}(z,z') \wedge \mathit{Unique}(z,u))\};\\
&T := \{(\bar{x},t) \mid \exists z (X_t(\bar{x},z)\\
&\qquad \wedge \textbf{if } \exists z'\, \mathit{Replace}(z,z') \textbf{ then } \mathit{Replace}(z,t) \textbf{ else } \mathit{Unique}(z,t))\}.
\end{aligned}
$$

- $f$ is 'Pair', for pairing, so $j = 2$. The only thing we have to change from the previous case is the assignment to relation $E$. We now write

$$E := \{(y,z) \mid \exists \bar{x} (X_t(\bar{x},z) \wedge (T_1(\bar{x},y) \vee T_2(\bar{x},y)))\}.$$

Finally, if $t$ is a term of the form $\{p : v \in r : q\}$, we may assume that relations $P$, $R$, and $Q$ for the terms $p$, $r$, and $q$ have already been computed. Note that, while the context for $t$ is $v_1 \dots v_n$, the context for $p$ and $q$ is $v_1 \dots v_n v$; the context for $r$ is still $v_1 \dots v_n$. Again, the only thing we have to change from the case where $t$ was of the form $\bigcup t_1$, is the assignment to $E$. We now write

$$E := \{(y,z) \mid \exists \bar{x} \exists v \exists r \exists q (X_t(\bar{x},z) \wedge R(\bar{x},r) \\ \wedge \mathit{Epsilon}(v,r) \wedge Q(\bar{x},v,q) \wedge P(\bar{x},v,y))\}.$$

Having finished the treatment of terms, we next show how to determine the actions of occurrences of rules occurring in $\Pi$. Like terms, every occurrence of a rule $R$ takes place in a context, consisting of a sequence $v_1 \dots v_k$ of variables which are bound by **forall** $v_i \in r_i$ **do** ... rules encompassing $R$. As with terms, we do distinguish between occurrences of the same syntactic rule in different contexts, but need not distinguish between different occurrences in the same context. We will compute, for every dynamic function name $f$ (of arity $j$), a $(k+j+1)$-ary relation $\mathit{Updates}_f^R$ holding all tuples $(a_1, \dots, a_j, b)$ for which $R$ specifies an update of $f(a_1, \dots, a_j)$ to $b$.

Again, we proceed by induction, and as in term evaluation, we use the formula $\mathit{Ranges}(x_1, \dots, x_k)$ setting the ranges of the context variables.

If $R$ is an elementary update of the form $f(t_1, \dots, t_j) := t_0$, we write

$$\mathit{Updates}_f^R := \{(\bar{x}, t_1, \dots, t_j, t_0) \mid \mathit{Ranges}(\bar{x}) \wedge \bigwedge_{l=0}^{j} T_l(\bar{x}, t_l)\}.$$

For every dynamic function name $g$ different from $f$, we put $\mathit{Updates}_g^R := \emptyset$. The relation variables $\mathit{Updates}_g^R$ for all dynamic function names $g \neq f$ are initialized empty.

If $R$ is of the form **if** $q$ **then** $R_1$ **else** $R_2$ **endif**, we may assume that relations $\mathit{Updates}_f^{R_1}$ and $\mathit{Updates}_f^{R_2}$, for all $f$, have already been computed. We then write, for each $f$ (of arity $j$),

$$\mathit{Updates}_f^R := \{(\bar{x}, y_1, \dots, y_j, y_0) \mid \mathit{Ranges}(\bar{x}) \wedge \exists q (Q(\bar{x},q) \\ \wedge \textbf{if } \mathit{One}(q) \textbf{ then } \mathit{Updates}_f^{R_1}(\bar{x}, y_1, \dots, y_j, y_0) \\ \textbf{else } \mathit{Updates}_f^{R_2}(\bar{x}, y_1, \dots, y_j, y_0))\}.$$

Finally, if $R$ is of the form **forall** $v \in t$ **do** $R_0$, we may assume that relations $\mathit{Updates}_f^{R_0}$, for all $f$, have already been computed. Note that the

context for $R_0$ is $v_1 \ldots v_k v$. We then write, for each $f$ (of arity $j$),

$$\mathit{Updates}_f^R := \{(\bar{x}, y_1, \ldots, y_j, y_0) \mid \mathit{Ranges}(\bar{x}) \wedge \exists t \exists v (T(\bar{x}, t) \wedge \mathit{Epsilon}(v, t) \wedge \mathit{Updates}_f^{R_0}(\bar{x}, v, y_1, \ldots, y_j, y_0))\}.$$

Since the BGS program $\Pi$ to be simulated is nothing but a rule without free variables, we can next show how to simulate one application of $\Pi$. All this amounts to is to perform the actual updates as specified in the relations $\mathit{Updates}_f^\Pi$, on condition that none of these relations contains a conflict, i.e., two tuples $(a_1, \ldots, a_j, b)$ and $(a_1, \ldots, a_j, b')$ with $b \neq b'$.

Concretely, for every dynamic function name $f$ (of arity $j$), define the following sentence:

$$\mathit{Conflict}_f := \exists a_1 \ldots \exists a_j \exists b \exists b' (\mathit{Updates}_f^\Pi(a_1, \ldots, a_j, b) \wedge \mathit{Updates}_f^\Pi(a_1, \ldots, a_j, b') \wedge b \neq b').$$

Then define the sentence *Conflict* as the disjunction $\bigvee_f \mathit{Conflict}_f$. We now write, for every dynamic function name $f$ (of arity $j$):

$\widehat{f} := \{(a_1, \ldots, a_j, b) \mid$
**if** *Conflict* **then** $\widehat{f}(a_1, \ldots, a_j, b)$ **else**
**if** $\neg \exists b'\, \mathit{Updates}_f^\Pi(a_1, \ldots, a_j, b')$ **then** $\widehat{f}(a_1, \ldots, a_j, b)$ **else**
**if** $\neg \exists z (\mathit{Updates}_f^\Pi(a_1, \ldots, a_j, z) \wedge \mathit{Zero}(z))$ **then** $\mathit{Updates}_f^\Pi(a_1, \ldots, a_j, b)$
**else false**$\}$.

Note that we avoid storing the empty set as a function value in $\widehat{f}$, because $\widehat{f}$ is supposed to store only the extent of $f$.

We finally have all the necessary ingredients to describe the desired program $\Pi'$:

⟨initialize⟩;
**while** $\neg \exists o(\widehat{\mathrm{Halt}}(o) \wedge \mathit{One}(o))$ **do**
⟨evaluate all terms⟩;
⟨apply $\Pi$ once⟩
**od**

The correctness of the simulation of $\Pi$ by $\Pi'$ should be evident. The time spent by $\Pi'$ is clearly linear in the time spent by $\Pi$. Regarding space usage, the invented elements occurring in the *Epsilon* relation during the execution of $\Pi'$ are in one-to-one correspondence with the active sets during the execution of $\Pi$. Moreover, the number of new elements invented at each

iteration of the while-loop is bounded by a polynomial in the number of these active elements. Hence, the simulation is linear-step, polynomial-space, and the proof is complete. □

## Acknowledgment

We thank Marc Spielmann for proofreading this paper.

## References

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

[2] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998.

[3] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings 7th ACM Symposium on Principles of Database Systems*, pages 240–250, 1988.

[4] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proceedings Fourth Annual Symposium on Logic in Computer Science*, pages 71–79. IEEE Computer Society Press, 1989.

[5] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[6] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on the Theory of Computing*, pages 209–219, 1991.

[7] A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.

[8] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

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

[10] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, 1984.

[11] M. Fernández, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a Web-site management system. *SIGMOD Record*, 27(2):414–425, 1998. Proceedings ACM SIGMOD International Conference on Management of Data.

[12] Y. Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, 1991.

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

[14] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, EECS Department, 1997.

[15] P. Halmos. *Naive Set Theory.* Van Nostrand Reinhold, 1960.

[16] J. Huggins, editor. *Abstract State Machines Web pages.* ⟨www.eecs.umich.edu/gasm⟩.

[17] Ph.G. Kolaitis and M.Y. Vardi. Infinitary logics and 0-1 laws. *Information and Computation*, 98(2):258–294, 1992.

[18] J. Van den Bussche and J. Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120:220–236, 1995.

[19] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.