A semi-deterministic approach to object creation and non-determinism in database queries*

Jan Van den Bussche† Dirk Van Gucht‡

Abstract

We introduce and study the concept of semi-determinism. A non-deterministic, generic query is called semi-deterministic if any two possible results of the query to a database are isomorphic. Semi-determinism is a generalization of determinacy, proposed by Abiteboul and Kanellakis in the context of object-creating query languages. The framework of semi-deterministic queries is less restrictive than that of the determinate queries and avoids the problem of copy elimination connected with determinacy. By offering a less restrictive framework, it avoids the problem of copy elimination connected with determinacy. We argue that semi-determinism is also interesting in its own right and show that it is natural and desirable, though hard to achieve in general. Nevertheless, we exhibit two major applications where semi-deterministic computations are possible. First, we show that there is a universal procedure to compute any semi-deterministic query in a semi-deterministic manner. Second, we show that the polynomial-time counting queries can be efficiently expressed semi-deterministically.

*An extended abstract of a preliminary version of this paper was presented at the 11th ACM Symposium on Principles of Database Systems.

†Research Assistant of the NFWO. Address: Dept. Math. & Computer Sci., University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: vdbuss@uia.ac.be.

‡Address: Computer Sci. Dept., Indiana University, Bloomington, IN 47405-4101, USA. E-mail: vgucht@cs.indiana.edu.

1 Introduction

The theory of queries in the context of the conventional relational database model is well understood. For a survey, see [3]. The theory started with the work of Chandra and Harel [10], who formally defined a *query* as a function from databases to databases which is partial recursive and preserves isomorphisms. They also presented a query language called QL, an extension of the relational algebra with unbounded looping and the possibility to store intermediate results in relations with unbounded arity. It was furthermore shown that QL is complete in the sense that it can express exactly all queries.

Abiteboul and Vianu [5] extended the notion of query by allowing non-determinism. They redefined a query as a binary relationship between databases which is recursively enumerable and preserves isomorphisms.[1] If the binary relationship happens to be a function, we call the query *deterministic* and the deterministic queries are exactly those of the previous paragraph. A language called TL [5, 6] was then proved to be complete for all (possibly non-deterministic) queries. TL is equivalent to the extension of the relational algebra with (i) unbounded looping; (ii) an operation to choose arbitrary tuples from a relation; and (iii) an operation to create new objects by tagging each tuple of a relation with a different new object (which can be thought of as an identifier.) It was also shown in [5] that detTL, a deterministic version of TL obtained by disallowing the choice operation and the appearance of new objects in the final result, is complete for the deterministic queries. In particular, this showed that object creation in intermediate relations is an alternative to the intermediate relations with unbounded arity of QL.

Motivated by applications in object-oriented database systems [19], the need arose for queries where new objects *do* appear in the final result. This lead Abiteboul and Kanellakis to the study of IQL [4], a query language roughly comparable in expressive power to detTL but without the prohibition of new objects in the result. In an attempt to capture the queries expressible in IQL, they defined the intuitively appealing class of *determinate* queries.[2] A non-deterministic query is called determinate if any two different results of the query applied to a database are isomorphic through an isomorphism that

[1] The work of Abiteboul and Vianu was not restricted to queries only, but also included updates. In this paper we will focus on the query aspect.

[2] Determinacy was also implicit in Kuper's thesis [21].

is the identity on the domain of the input database. Hence, the isomorphism can only permute the new objects. Determinacy thus intends to isolate the very weak form of non-determinism which is needed to accommodate new objects.

It turned out that IQL is not complete for the determinate queries: an extra operation called *copy elimination* [4] has to be added in order to obtain determinate-completeness. Nevertheless, most likely all determinate queries arising in practice can be already be expressed in IQL (without copy elimination.) So, one could conclude that the class of determinate queries is perhaps not so natural after all. This philosophically unsatisfactory situation left open two natural directions for further research: restrict or extend the class of determinate queries to find more natural classes.

The first direction was explored in [27], where the precise expressive power of IQL was characterized. It was shown that this characterization is obtained when an additional requirement, besides determinacy, is made which simply expresses that the creation of new objects can be interpreted as the deterministic construction of hereditarily finite sets. The feeling expressed above that the class of IQL-expressible determinate queries, coined the *constructive* queries in [27], indeed arises very naturally was thus confirmed.

In the present paper, we explore the other direction. We propose the notion of *semi-determinism*. A non-deterministic query is called semi-deterministic if any two different results of the query to a database are isomorphic, by an isomorphism which maps the input database onto itself (i.e., an automorphism.) Note that determinate queries are very restricted semi-deterministic queries, for which this automorphism is the identity. Thus, semi-determinism is a natural generalization of determinacy, and at the same time a natural restriction on arbitrary non-determinism: the only sources of non-determinism are the symmetries (automorphisms) present in the input database.

The contents of this paper can be summarized as follows. In Section 2, we review the necessary preliminary notions.

In Section 3, we introduce and motivate semi-determinism. Given that semi-determinism is an extension of determinacy, we provide a necessary and sufficient algebraic condition for when a semi-deterministic query is actually determinate. We also present an alternative characterization for semi-determinism which demonstrates its naturalness. Another motivation for semi-determinism which we prove is that a non-deterministic loop program

which makes its choices in a semi-deterministic manner, has the good property that if one of its possible computations halts, then all possible computations will halt, and this after the same number of iterations of the loop. We also observe that semi-deterministic queries which yield a yes-no answer must in fact be deterministic, which is quite desirable.

While semi-determinism has good properties, it is hard to achieve in general. Indeed, we prove that the problem of syntactically verifying whether a non-deterministic program expresses a semi-deterministic query is undecidable. Furthermore, checking a computation for semi-determinism at run-time is shown to be polynomial-time equivalent to checking graph isomorphism.

Nevertheless, in Sections 4 and 5, we exhibit two major applications where semi-deterministic computations are possible: *completeness* and *counting*. In both of these applications, object creation plays a crucial role.

In Section 4, we prove that every semi-deterministic query can be expressed by a program in TL that contains only one single application of the choice operation. This application is semi-deterministic and is a generalization of the copy elimination operation to the semi-deterministic context. This result indicates that the obvious candidate language for semi-deterministic completeness, namely the semi-deterministic TL programs, is indeed semi-deterministic complete. It also suggests that the notion of copy elimination naturally falls in the framework of semi-deterministic queries.

In Section 5, we show that all polynomial-time counting queries can be expressed efficiently in a semi-deterministic manner. Queries involving counting are computationally simple but cannot be expressed in the extension of the relational algebra with looping. Using non-determinism, this situation can be remedied: it is well-known [3] that all polynomial-time computable queries can be expressed efficiently in the extension of the relational algebra with looping and non-deterministic choice. Our result shows that, as far as counting is concerned, the advantages of non-determinism can also be obtained semi-deterministically. To this end, we demonstrate a new, semi-deterministic technique for constructing restricted types of orderings that are sufficient for counting purposes.

Concluding remarks are in Section 6.

2 Preliminaries

In this section, we review the necessary preliminary notions.

We will work in the following version of the well-known relational database model [3]. Formally, assume the existence of sufficiently many *relation names*. Every relation name R has an associated *arity* $a(R)$, a natural number. For each natural number n there are sufficiently many relation names R with $a(R) = n$. A *database scheme* is a finite set of relation names. Furthermore, let $\mathbf{U}$ be an infinite, recursively enumerable universe of data elements called *atomic objects* (or *objects* for short). An *instance* I over a scheme $\mathcal{S}$ is a mapping, assigning to each relation name R of $\mathcal{S}$ a finite relation $I(R) \subset \mathbf{U}^{a(R)}$. The *active domain* of a relation r (or an instance I) is the set of all objects occurring in it, and is denoted by $adom(r)$ ($adom(I)$). The set of all instances over a scheme $\mathcal{S}$ is denoted by $inst(\mathcal{S})$. If $\mathcal{S} = \{R_1, \ldots, R_n\}$, then an instance $I \in inst(\mathcal{S})$ will be denoted as $I = (R_1 : I(R_1), \ldots, R_n : I(R_n))$.

In its most general form, a query can be thought of as a possibly nondeterministic process, augmenting databases with derived information. Formally, let $\mathcal{S}$ be a scheme, and let $A_1, \ldots, A_n$ be relation names not in $\mathcal{S}$. The following definition is adapted from [5].

Definition 2.1 A *query of type* $\mathcal{S} \to A_1, \ldots, A_n$ is a recursively enumerable, binary relationship $Q \subset inst(\mathcal{S}) \times inst(\mathcal{S} \cup \{A_1, \ldots, A_n\})$ such that:

1. if $Q(I, J)$ then J is equal to I on $\mathcal{S}$; and
2. if $Q(I, J)$ and f is a permutation of $\mathbf{U}$ then $Q(f(I), f(J))$.

The latter requirement is now commonly called *genericity* and traces back to references [8] and [10].

We will sometimes use the following notation: if ρ is a binary relationship, then $\rho(x)$ stands for the set $\{y \mid \rho(x, y)\}$. For a query Q and instance I, $Q(I)$ is called the set of *possible results* of Q for I. If $Q(I) = \emptyset$ then the result of Q is undefined on input I.

Queries that are functions are called *deterministic*. A basic language for expressing deterministic queries is RA. This is a version of the relational algebra [3], an algebraization of first-order predicate logic (a.k.a. relational calculus), consisting of the relational operators union ($\cup$), difference ($-$), Cartesian product ($\times$), selection ($\sigma_{i=j}$, selects from a relation those tuples for

which the i-th and the j-th component are equal), and projection ($\pi_{i_1,\ldots,i_k}$). Let $\mathcal{S}$ be a scheme and let $R_1, \ldots, R_m$ be a sequence of relation names not in $\mathcal{S}$. They will be used as relation-valued variables, holding intermediate results of the computation. The sequence P of *relation assignments*:

$$R_1 := E_1;\ \ldots;\ R_m := E_m,$$

where each E_i is a relational algebra expression using only relation names in $\mathcal{S} \cup \{R_1, \ldots, R_{i-1}\}$, is a *RA program.* If we choose a subset of *answer relation* names $\{A_1, \ldots, A_n\}$ among the R_i's, then this program P computes a query of type $\mathcal{S} \to A_1, \ldots, A_n$ in the obvious and well-known way. We will not always explicitly indicate the answer relations in a program; they will often be clear from the context.

A query Q is called *object-creating* if there exist I, J such that $Q(I, J)$ and $adom(I) \subset_{\neq} adom(J)$ (note that for any query Q, $Q(I, J)$ implies $adom(I) \subseteq adom(J)$). Object-creating queries are necessarily non-deterministic, because of genericity. In proof, assume $Q(I, J)$ and $o \in adom(J) - adom(I)$. Take an arbitrary $o' \in \mathbf{U}$ such that $o' \notin adom(J)$, and consider the permutation f of $\mathbf{U}$ which transposes o and o': $f = (o\ o')$. Then $f(I) = I$ and $f(J) \neq J$ and, by genericity, $Q(I, f(J))$.

A natural object-creating operation which can be added to RA is *new*. Let $\mathcal{S}$ be a scheme, $I \in inst(\mathcal{S})$, and $R \in \mathcal{S}$ with $a(R) = a$. A relation r of arity $a + 1$ is a possible result of $new(R)$ applied to I if r is obtained from $I(R)$ by extending each tuple t of $I(R)$ with a different new object $t(a + 1)$ that is not yet in $adom(I)$.

Example 2.2 Let $\mathcal{S} = \{R\}$ with $a(R) = 2$ and

$$I(R) = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, d\rangle\}.$$

A possible result of $new(R)$ applied to I is

$$\{\langle a, b, \alpha_1\rangle, \langle a, c, \alpha_2\rangle, \langle b, d, \alpha_3\rangle\},$$

where the α_i are three arbitrary new objects from $\mathbf{U}$.

So, in a sense, *new* is the converse of projection. We can add *new* to the operations of the relation algebra, thus obtaining the query language RA + *new*.

The language RA + *new* provides a basic and general object creation mechanism in function of tuples. Essentially this same mechanism is provided by many other query languages with object-creating capabilities that have been proposed in the literature [4, 5, 6, 16, 17, 18, 22]. Some languages [4, 16, 22, 24] provide also object creation in function of sets. We will not explicitly need this capability to prove our results, but will return to it in Section 6.

While we know that an object-creating query must be non-deterministic, it is clear that the queries expressed by programs of RA + *new* are "nearly" deterministic: different results of these queries to a database differ only in the particular choice of the new objects that have been created. To capture this intuition, the notion of *determinacy* was defined in [4]:

Definition 2.3 Query Q is called *determinate* if whenever $Q(I, J_1)$ and $Q(I, J_2)$, then J_1 and J_2 must be *I-isomorphic*, i.e., there must be a permutation f of $\mathbf{U}$, that is the identity on $adom(I)$, such that $f(J_1) = J_2$.

We now define an operation, W (*Witness*, [3]), which allows for arbitrary non-determinism. Let $\mathcal{S}$ be a scheme, $I \in inst(\mathcal{S})$, and $R \in \mathcal{S}$ with $X \subseteq \{1, \ldots, a(R)\}$. A relation r of arity $a(R)$ is a possible result of $W_X(R)$ applied to I if r is a subset of $I(R)$ obtained by choosing for each class of X-equivalent tuples of $I(R)$ exactly one representative. Here, two tuples are called *X-equivalent* if they are equal outside X. In particular, if $X = \{1, \ldots, a(R)\}$, then any two tuples are X-equivalent.

Example 2.4 Let $\mathcal{S}$ and I be as in Example 2.2. If we interpret $I(R)$ as a parent-child relation, then $W_2(R)$ amounts to choosing a child for each parent. The two possible results of $W_2(R)$ applied to I are

$$r_1 = \{\langle a, b\rangle\,,\, \langle b, d\rangle\}$$

and

$$r_2 = \{\langle a, c\rangle\,,\, \langle b, d\rangle\}.$$

On the other hand, $W_{1,2}(R)$ chooses an arbitrary tuple from $I(R)$, so the three possible results of $W_{1,2}(R)$ applied to I are $\{\langle a, b\rangle\}$, $\{\langle a, c\rangle\}$, and $\{\langle b, d\rangle\}$.

We can extend RA and RA + *new* with non-determinism by allowing assignments of the form $A := W_X(E)$, where E is a relational algebra expression.

The resulting language is denoted by RA + W or RA + *new* + W. Note that we allow applications of the Witness operation only at the end of a relational algebra expression in an assignment, not within expressions. This is no real restriction and will make it easier to define semi-deterministic programs in Section 3.

Finally, all languages introduced so far must often be enriched with a looping construct to increase expressiveness. We will use here a loop with partial fixpoint semantics [3]. This is a "repeat-while-change" looping semantics, but special care must be taken to define it in the presence of non-determinism. Let P be a program in RA + *new* + W. Let Q be the query expressed by P and let Q^n be the query expressed by P^n, the n-fold concatenation of P. Then the query Q' expressed by the program $loop[P]$ is defined as follows: On input instance I, $J \in Q'(I)$ if there exists an n such that $J \in Q^n(I) \cap Q(J)$. We can extend any query language $\mathcal{L}$ to $\mathcal{L} + loop$ as follows: (i) if P is an $\mathcal{L}$-program, then $loop[P]$ is also a program; (ii) programs can be composed using ';'.

If P is determinate, then the above amounts to the typical fixpoint semantics. So either $loop[P]$ is undefined on I since it does not halt, or the result equals $Q^n(I)$ for the least n such that $Q^n(I) = Q^{n+1}(I)$. However, if P is arbitrarily non-deterministic, $loop[P]$ can have a much more complicated behavior. Some possible computations may halt, while others may not halt. Furthermore, not all possible results of the loop will be computed after the same number of iterations.

Example 2.5 Let R be a binary relation, viewed as (the set of edges of) a directed graph. Let Q be the query of type $\{R\} \to A$, where A is binary, defined by $Q(I, J)$ if J^A lists each node x in I together with an arbitrary node reachable from x in R^I. This query is expressible by the following program:

$D := \pi_1(R) \cup \pi_2(R)$;
$E := \sigma_{1=2}(D \times D)$;
$A := W_2(R)$;
loop [$A := W_2(\pi_{1,4}\sigma_{2=3}(A \times (R \cup E)))$].

Possible computations of the loop in this program, when applied to a fixed instance, may halt after any number of iterations, or may loop indefinitely.

We will show later (Theorem 3.13) that non-deterministic loop programs do have a nice behavior if the non-determinism is in fact semi-deterministic.

The language which includes all features introduced in this section, RA + *new* + W + *loop*, will often be referred to as TL, since it is equivalent to the language TL of [5] (see also [6]). TL is complete; it can express all queries.

3 Semi-determinism

In this section, we introduce and motivate semi-determinism, and also show that it is hard to achieve in general.

3.1 Definition and general properties

Semi-determinism is a natural restriction on the amount of non-determinism of a query Q:

Definition 3.1 Query Q is called *semi-deterministic* if whenever $Q(I, J_1)$ and $Q(I, J_2)$, then J_1 and J_2 are isomorphic, i.e., there is a permutation f of **U** such that $f(J_1) = J_2$.

By definition of query, J_1 and J_2 are extensions of I, and it follows that the stated isomorphism f from J_1 to J_2 is an automorphism of I. (In this paper, with an automorphism of I we mean a permutation f of **U** such that $f(I) = I$.) So intuitively, the only sources of non-determinism in a semi-deterministic query are the symmetries (automorphisms) in the input database.

Clearly, determinate queries (Definition 2.3) are very restricted semi-deterministic queries, for which the above mentioned automorphism is actually the identity on I. The difference between determination and semi-determinism can also be characterized by the next Theorem. We say that an automorphism f of an instance I can be *extended* to an automorphism of an instance J, with $adom(J) \supseteq adom(I)$, if there is a permutation g of **U** such that $g|_{adom(I)} = f|_{adom(I)}$ and $g(J) = J$.

Theorem 3.2 *A semi-deterministic query Q is determinate if and only if whenever $Q(I, J)$ and f is an automorphism of I, then f can be extended to an automorphism of J_2.*

Proof: *If:* Assume $Q(I, J_1)$ and $Q(I, J_2)$. Since Q is semi-deterministic, there is a permutation f of **U** such that $f(J_1) = J_2$. f is an automorphism

of I, so it can be extended to an automorphism of J_2. This means there is a permutation g such that $g|_{adom(I)} = f|_{adom(I)}$ and $g(J_2) = J_2$. Let p be the order of $g|_{adom(I)}$ in the permutation group of $adom(I)$. I.e., p is the least such that $p > 0$ and $(g|_{adom(I)})^p$ is the identity. Define $h := g^{(p-1)}f$. Then $h(J_1) = J_2$, and h is the identity on $adom(I)$. Therefore, Q is determinate.

Only if: Let f be an automorphism of I. By genericity, $Q(I, f(J))$. By determinacy, there is a permutation g such that $g(f(J)) = J$ and g is the identity on $adom(I)$. So, $h := gf$ satisfies $h|_{adom(I)} = f|_{adom(I)}$ and $h(J) = J$, as desired. ■

The above theorem also yields the following:

Corollary 3.3 *A semi-deterministic query Q is deterministic if and only if whenever $Q(I, J)$ and f is an automorphism of I, then f is also an automorphism of J.*

Proof: *Only if:* Suppose for the sake of contradiction that f is not an automorphism of J. This means that $f(J) \neq J$. But by genericity, $Q(I, f(J))$, whence Q is not deterministic, contradiction.

If: If $Q(I, J)$ and f is an automorphism of I, then surely f can be trivially extended to an automorphism of J. Hence, by Theorem 3.2, Q is determinate. So to prove that Q is deterministic, it suffices to prove that Q is not object-creating. Suppose for the sake of contradiction that Q is object-creating. Then there exist I, J such that $Q(I, J)$ and $adom(J) \not\subseteq adom(I)$. Let $o \in adom(J) - adom(I)$. Let f be a permutation of $\mathbf{U}$ such that $f(I) = I$ and $f(o) \notin adom(J)$. But then $f(J) \neq J$, so f is an automorphism of I but not of J, contradiction. ■

The qualification *semi-deterministic* cannot be omitted from Theorem 3.2 or Corollary 3.3. To show this for Theorem 3.2, consider the query Q of type $\{V\} \to W$, where $a(V) = a(W) = 1$, defined as follows: $Q(I, J)$ if $J(W) = I(V)$ or if $J(W) = I(V) \cup \{o\}$ for some arbitrary $o \notin I(V)$. Although every automorphism of I can be extended to an automorphism of J, Q is not determinate. For Corollary 3.3, consider the query Q of type $\{V\} \to R$, where $a(V) = 1$ and $a(R) = 2$, defined as follows: $Q(I, J)$ if $I(V)$ is of the form $\{a, b\}$ and $J(W)$ is either of the form:

$$\{\langle a, a\rangle, \langle b, b\rangle\}$$

or of the form:

$$\{\langle a, b\rangle, \langle b, a\rangle\}.$$

Although every automorphism of I is also an automorphism of J, Q is not deterministic.

We also point out that, since we defined queries as augmentations, the following straightforward but fundamental closure property holds:

Proposition 3.4 *The composition of two semi-deterministic queries is again semi-deterministic.*

We now present an interesting characterization of the semi-deterministic queries [14]. To do this, we need the following auxiliary notion:

Definition 3.5 Let $\mathcal{S}, A_1, \ldots, A_n$ be as in Definition 2.1. A *pre-query of type* $\mathcal{S} \to A_1, \ldots, A_n$ is a recursively enumerable, binary relationship $Q_0 \subset inst(\mathcal{S}) \times inst(\mathcal{S} \cup \{A_1, \ldots, A_n\})$ such that:

1. if $Q_0(I, J)$ then J is equal to I on $\mathcal{S}$; and
2. if $Q_0(I, J)$ and $Q_0(I', J')$ then either $I = I'$ and $J = J'$, or I and I' are not isomorphic.

So, a pre-query is an arbitrary partial recursive function from instances to instances with the property that it is defined on at most one representative of each isomorphism type. In particular, pre-queries are never generic. Therefore, it makes sense to define the closure of a pre-query under the genericity requirement:

Definition 3.6 The *closure* of pre-query Q_0 is the query

$$Q_0^* := \{(f(I), f(J)) \mid Q_0(I, J) \text{ and } f \text{ permutation of } \mathbf{U}\}.$$

We observe:

Proposition 3.7 *For any pre-query Q_0, its closure Q_0^* is recursively enumerable.*

Proof: The standard approach to enumerate Q_0 would be to enumerate the Cartesian product of Q_0 with the set of all permutations f of $\mathbf{U}$. The only problem with this approach is that the latter set is uncountable. However,

since we are working with finite instances, it suffices to consider only the permutations having finite support, i.e., those permutations that are the identity on all but a finite number of elements. Since every such permutation is the composition of a finite number of transpositions, it thus suffices to enumerate the Cartesian product of Q_0 with the set of all finite sequences of transpositions. ■

Hence, Q_0^* is the minimal query containing Q_0. We now establish:

Theorem 3.8 *Query Q is semi-deterministic if and only if $Q = Q_0^*$ for some pre-query Q_0.*

Proof: *If:* Assume $Q(I, J_1)$ and $Q(I, J_2)$. By the definition of Q_0^*, there are instances I_0, J_0 and permutations f_1, f_2 of $\mathbf{U}$ such that $Q_0(I_0, J_0)$, $I = f_1(I_0)$ and $J_1 = f_1(J_0)$, and $I = f_2(I_0)$ and $J_2 = f_2(J_0)$. Hence, $f_2 f_1^{-1}$ is the desired isomorphism from J_1 to J_2.

Only if: Let Q_0 be any maximal pre-query contained in Q. Since Q is recursively enumerable, such a Q_0 exists. We show that $Q = Q_0^*$. Since Q is generic, it suffices to show that $Q \subseteq Q_0^*$. Assume $Q(I, J)$. Since Q_0 is maximal, there exists an I_0 isomorphic to I such that $Q_0(I_0, J_0)$ for some J_0. Let f be a permutation of $\mathbf{U}$ such that $f(I_0) = I$. Since $Q(I_0, J_0)$, we have $Q(I, f(J_0))$, by genericity of Q. Since Q is semi-deterministic, there exists a permutation g of $\mathbf{U}$ such that $g(f(J_0)) = J$. Since $g(I) = I$ and hence $g(f(I_0)) = I$, it follows that $Q_0^*(I, J)$ through application of gf to $Q_0(I_0, J_0)$, as desired. ■

Before we move to semi-deterministic *programs*, we make one final observation concerning semi-deterministic queries in general. A *boolean* query is a query of type $\mathcal{S} \to A$, with $a(A) = 0$. So a boolean query can have only two possible answers: the empty zero-ary relation $\emptyset$, which is interpreted as False, and the non-empty zero-ary relation $\{\langle\ \rangle\}$, which is interpreted as True. Since True and False are non-isomorphic, we immediately obtain:

Proposition 3.9 *Every boolean semi-deterministic query is deterministic.*

3.2 Semi-deterministic programs

Given a TL program P, when would we call it semi-deterministic? We could call it semi-deterministic simply if the query expressed by P is semi-deterministic. But this is not a very interesting definition. It would be more

interesting if the whole of P's computation is semi-deterministic, not just merely the end result. Formally, the semi-deterministic programs are defined as follows:

Definition 3.10 1. We first define the *unfoldings* of a TL program P:

   (a) If P is a program not containing any loops, then the only unfolding of P is P itself.

   (b) If P is of the form $loop[P']$, then the unfoldings of P are all n-fold concatenations $(P')^n$ of P' with itself, for all natural numbers n.

   (c) If P is of the form $P_1; P_2$ then the unfoldings of P are all programs of the form $P_1'; P_2'$, with P_1' (P_2') an unfolding of P_1 (P_2).

2. An unfolding of a program is a sequence of assignments $\sigma_1; \ldots; \sigma_n$. We call such a sequence *semi-deterministic* if for each $i \leq n$, the program $\sigma_1; \ldots; \sigma_i$ expresses a semi-deterministic query.

3. Finally, a program is called *semi-deterministic* if all its unfoldings are.

In item (2) of the above definition, it is required that every intermediate stage is semi-deterministic. In principle it is sufficient to verify the requirement only for those i where σ_i contains an application of the Witness operation, since these are the only places where things can go wrong.

Clearly, the query expressed by a semi-deterministic program is semi-deterministic. Of course, the converse does not hold. In the next example, we will give examples of non-semi-deterministic queries and of semi-deterministic queries expressed by programs that are semi-deterministic and by programs that are not.

Example 3.11 Let $\mathcal{S}$ and I be as in Example 2.2. Consider the RA + W-program P_1:

$$A_1 := \pi_1(R) \cup \pi_2(R);\ A_2 := W_1(A_1).$$

In a possible result of applying this program to I, the A_1-relation will hold $adom(I)$, and the A_2-relation will hold one of the four singleton subsets of $adom(I)$. Among these four possible outcomes, the two possible results $(R : I(R), A_1 : adom(I), A_2 : \{a\})$ and $(R : I(R), A_1 : adom(I), A_2 : \{b\})$ are not isomorphic: there is no automorphism of I mapping a to b. So, the

program does not express a semi-deterministic query, and so it certainly not semi-deterministic.

To illustrate the difference between semi-deterministic programs and semi-deterministic queries, consider the very simple scheme $\{V\}$, with $a(V) = 1$, and the programs P_2:

$$A_1 := W_1(V);\ A_2 := W_1(V)$$

and P_3:

$$A_1 := W_1(V);\ A_2 := W_1(V - A_1).$$

Let us restrict attention to instances I for which $I(V)$ has at least two elements. If in both programs, A_2 is the answer relation, the query Q expressed by P_2 and P_3 is the same. Applied to an instance I, a possible result of Q will hold (in the A_2 relation) an arbitrary element of $I(V)$. Since the elements of $I(V)$ are indistinguishable, Q is semi-deterministic. However, to check whether the programs P_2 and P_3 are semi-deterministic, we also have to take the temporary A_1 relation into account. So we see that P_2 is not semi-deterministic; two possible intermediate results could be the instance J_1 with $J_1(A_1) = \{\langle a\rangle\}$ and $J_1(A_2) = \{\langle a\rangle\}$, and the instance J_2 with $J_2(A_1) = \{\langle b\rangle\}$ and $J_2(A_2) = \{\langle c\rangle\}$. Clearly, J_1 and J_2 are not isomorphic. On the other hand, P_3 is semi-deterministic, since any possible intermediate result of P_3 will contain an element of V in A_1 and *another* element of V in A_2. All such configurations are clearly isomorphic.

Example 3.12 Finally, to give a less abstract example, consider the scheme $\{Prof, Stud\}$ with $a(Prof) = a(Stud) = 1$. The intended meaning of this scheme is that the *Prof*-relation holds a set of professor names, and the *Stud*-relation holds a set of student names. We assume that the sets of student names and professor names are disjoint. The reader is invited to verify that the following program is semi-deterministic:

$Prof\_chosen := \emptyset;$
$Stud\_chosen := \emptyset;$
$Advisor := \emptyset;$
loop [
    $C\_stud := W_1(Stud - Stud\_chosen);$
    $C\_prof := W_1(Prof - Prof\_chosen);$

$Advisor := Advisor \cup (C\_stud \times C\_prof)$;
$Stud\_chosen := Stud\_chosen \cup C\_stud$;
$Prof\_chosen := Prof\_chosen \cup C\_prof$;
]

The query expressed by this program produces in the answer relation *Advisor* an arbitrary one-to-one student-advisor assignment. All possible such assignments are isomorphic by an isomorphism mapping profs to profs and students to students; hence, the query is indeed semi-deterministic.

The above examples do not involve object creation, but it will become clear later that object creation is actually crucial to performing semi-deterministic computations in less contrived situations. For now, we demonstrate a good property of semi-deterministic loop programs which we can prove in the most general setting of all TL programs.

Theorem 3.13 *Let P be a TL program such that $loop[P]$ is semi-deterministic, and let I be an input instance. Then either the result of $loop[P]$ on I is undefined since no possible computation halts, or every possible computation halts after the same number of iterations of the loop.*

Proof: Assume J is a possible result of $loop[P]$ on I. By definition, there exists n such that $J \in P^n(I) \cap P(J)$. So, $J \in P^{n+1}(I)$ and J is output after $n+1$ iterations. Now let J' be another element of $P^{n+1}(I)$, arbitrarily chosen. So, J' is the preliminary result, after $n+1$ iterations, of an arbitrary possible computation of the program. We must show that $J' \in P^n(I) \cap P(J')$. Indeed, if this is the case then J' is actually a final result of $loop[P]$, or in other words, its corresponding computation halts after $n+1$ iterations, as desired.

Since $loop[P]$ is semi-deterministic, P^{n+1} is semi-deterministic. Since both $P^{n+1}(I, J)$ and $P^{n+1}(I, J')$, J' must be isomorphic to J. But then since $P^n(I, J)$, by genericity also $P^n(I, J')$. Moreover, since $P(J, J)$, by genericity, also $P(J', J')$. ■

It is known [3] that it is undecidable whether an RA + W-program, and more specifically, an RA + W-program of the form: $R := E$; $A := W_X(R)$

(where E is an arbitrary relational algebra expression) expresses a deterministic query.[3] Although semi-determinism is less restrictive than plain determinism, the analogue of this result still holds:

Theorem 3.14 *It is undecidable whether a program in* RA + W *expresses a semi-deterministic query.*

Proof: Let P be a program of the form $R := E;\ A_1 := W_X(R)$. Let P' be the program $P;\ A_2 := W_X(R);\ A := A_1 - A_2$, with A the answer relation. Since the assignments to A_1 and A_2 might be the same, we have that for any instance I, there is a possible result J of P' on I for which $J(A) = \emptyset$. Since the only relation isomorphic to $\emptyset$ is $\emptyset$ itself, it follows that the query Q' expressed by P' is semi-deterministic iff Q' is deterministic. But Q' is deterministic iff the query Q expressed by P is deterministic, and this is undecidable. ■

Corollary 3.15 *It is undecidable whether a program in* RA + W *is semi-deterministic.*

Proof: Follows from the observation that programs P' of the special form exhibited in the proof of Theorem 3.14 are semi-deterministic iff the query they express is semi-deterministic, which was just shown to be undecidable. ■

Theorem 3.14 shows that "compile-time" checking for semi-determinism is infeasible. As an alternative, we can check for semi-determinism "at run-time." Given a TL program P and an input instance I, execute P on I and add, a posteriori, an extra checking phase to see that all possible results are pairwise isomorphic. If this check fails, the result of P on I is overruled to some default value, e.g., all empty answer relations. We have thus defined an alternative semantics for TL programs, which we naturally call the *semi-deterministic* semantics. If we want not only the query expressed by the

[3] In short, the reason is the following. The Witness operation $W_X(R)$ is deterministic iff the complement X^c of X is a key for its argument relation R. Since R is the result of algebra expression E, it thus follows that the program is deterministic iff E implies the key dependency $key(X^c)$. This is undecidable because it is undecidable whether a functional dependency is implied by a first-order sentence; a much stronger version of this statement was proven in [11].

program, but also the program itself to be semi-deterministic, we further add a similar check after each application of the Witness operation. This stronger version of the semi-deterministic semantics is called the *uniformly* semi-deterministic semantics.

We now show that run-time checking can be performed in TL. That is, the just mentioned extra checking phases can be simulated in TL itself. This should come as no surprise, in view of the computational completeness of TL. Therefore, we only give a sketch of the proof.

Proposition 3.16 *For every TL program P there is another TL program P^{sd} (resp. P^{usd}) such that the ordinary semantics of P^{sd} corresponds to the (resp. uniformly) semi-deterministic semantics of P.*

Proof: P^{sd} consists of two parts. The first part is a deterministic (or rather determinate) simulation of P. Applications of Witness are simulated by generating all possible results, and keeping track of them in subsequent computations. After the end of the simulation, there is a second part that checks whether the accumulated possible results are pairwise isomorphic. (Between each pair of possible results, all possible bijections are generated by a powerset-like construction using object creation. Then it is verified whether at least one of these bijections is an isomorphism.) If the test fails, the answer relations are assigned empty by default. Otherwise, using Witness, an arbitrary possible result is chosen. The proof for P^{usd} is analogous. ■

We have just shown that run-time checking for semi-determinism, in contrast to compile-time checking, is decidable. However, the tests for isomorphism that are involved are similar to testing graph isomorphism, for which no polynomial-time algorithm is known. Indeed, we next show that the two problems are polynomial-time equivalent.

Proposition 3.17 *Run-time checking for semi-determinism is polynomial-time equivalent to checking graph isomorphism.*

Proof: Run-time checking for semi-determinism can be reduced in polynomial time to checking graph isomorphism, because checking isomorphism of relational structures (i.e., database instances) reduces to checking graph isomorphism [23].

For the converse direction, consider the scheme $\mathcal{S} = \{R, V\}$ with $a(R) = 2$ and $a(V) = 1$, and consider the class $\mathcal{I}$ of instances I over $\mathcal{S}$ for which $I(V)$ contains exactly two elements of $adom(I(R))$. Consider the following one-line program $P \equiv A := W_1(V)$. Checking P for semi-determinism on an instance I of the class $\mathcal{I}$ amounts to checking whether two given nodes o_1, o_2 of a graph (binary relation) are *auto-equivalent*, meaning that there is an automorphism of the graph mapping o_1 to o_2. We conclude the proof by showing that graph isomorphism can be reduced in polynomial time to auto-equivalence. Assume given two graphs G_1, G_2, of which we may assume that they are connected and disjoint. In order to test whether there is an isomorphism between G_1 and G_2, it suffices to test whether there is a pair (o_1, o_2), where o_i is a node of G_i, such that o_1 and o_2 are auto-equivalent within $G_1 \cup G_2$. ■

This section can be concluded by saying that semi-determinism is a desirable and natural notion, but at the same time it is hard to achieve. Nevertheless, in the next two sections we will exhibit two major applications where semi-deterministic computations are possible.

4 Semi-determinism and completeness

In this section, we prove that every semi-deterministic query can be expressed by a semi-deterministic TL program.

4.1 Determinate-completeness up to copies

The language RA+*new*+*loop* is a very powerful, determinate language, which is roughly comparable in expressive power to the language IQL [4]. The only difference is that IQL supports set values. This difference is irrelevant to the issues discussed in the main body of this paper, but we will return to set values in Section 6.

If we restrict the use of object creation syntactically, so that new objects appear only in intermediate relations and not in the final answer relations, we obtain a sublanguage of RA + *new* + *loop*, which we call detTL since it is equivalent to the language detTL of [5, 6]. The queries expressed by detTL programs are not just determinate, but actually deterministic, exactly because no new objects appear in the result. In fact, it is known [3] that

detTL is deterministic-complete: *all* deterministic queries can be expressed in the language.

Therefore, it came much as a surprise that RA + *new* + *loop* is *not* determinate-complete. This is illustrated by the following example.

Example 4.1 Consider a relation name A of arity 1. Let B, C be two relation names of arity 2. Consider any determinate query Q of type $\{A\} \to B, C$ containing a pair $Q(I^{diff}, J^{diff})$. Here, I^{diff} is an instance of the very simple form $(A : \{a_1, a_2\})$, and the corresponding result J^{diff} has the form:

$$\begin{array}{l} A : \{a_1, a_2\}, \\ B : \{\langle b_1, a_1\rangle, \langle b_3, a_1\rangle, \langle b_2, a_2\rangle, \langle b_4, a_2\rangle\}, \\ C : \{\langle b_1, b_2\rangle, \langle b_2, b_3\rangle, \langle b_3, b_4\rangle, \langle b_4, b_1\rangle\}. \end{array}$$

So the b_i's are new objects. This can be easy visualized using graphs: starting from a discrete graph containing two isolated A-nodes a_1, a_2, a C-cycle of four new nodes $b_1, \ldots, b_4$ is created such that two opposite b-nodes are associated to a common a-node through the B-relation. It can be shown [2, 9, 27] that such a query Q is not expressible in RA + *new* + *loop*.

The non-completeness of RA + *new* + *loop* can be put in a more structured framework using the notion of *instance with copies* [4]. Let $\mathcal{S}$ be a scheme, and let $\mathcal{S}_0 \subseteq \mathcal{S}$. For each $R \in \mathcal{S} - \mathcal{S}_0$, let CR be a relation name not in $\mathcal{S}$ for which $a(CR) = a(R) + 1$. All these CR must be different. Let $\overline{\mathcal{S}} := \mathcal{S}_0 \cup \{CR \mid R \in \mathcal{S} - \mathcal{S}_0\}$. Let $J \in inst(\mathcal{S})$, and $\overline{J} \in inst(\overline{\mathcal{S}})$. Then we define:

Definition 4.2 $\overline{J}$ is an instance *with copies of* J *w.r.t.* $\mathcal{S}_0$ if there exist:

(i) A natural number $n > 0$, called the *number of copies*;

(ii) n instances $J_1, \ldots, J_n \in inst(\mathcal{S})$, called *copies*, such that the sets $adom(J) - adom(J|_{\mathcal{S}_0})$, $adom(J_1) - adom(J_1|_{\mathcal{S}_0})$, $\ldots$, $adom(J_n) - adom(J_n|_{\mathcal{S}_0})$ are pairwise disjoint, and J_k and J are $J|_{\mathcal{S}_0}$-isomorphic for $k = 1, \ldots, n$;[4]

[4]Recall from Definition 2.3 that an I-isomorphism, for some instance I, is an isomorphism that is the identity on $adom(I)$.

(iii) n objects $\varepsilon_1, \ldots, \varepsilon_n$, called *copy identifiers*, not appearing in J or any J_k,

such that

$$\overline{J}(CR) = \big(J_1(R) \times \{\langle \varepsilon_1 \rangle\}\big) \cup \cdots \cup \big(J_n(R) \times \{\langle \varepsilon_n \rangle\}\big)$$

for each $R \in \mathcal{S} - \mathcal{S}_0$, and $\overline{J}(R) = J(R)$ for each $R \in \mathcal{S}_0$.

It follows from (ii) that each J_k agrees with J on $\mathcal{S}_0$, and hence also $\overline{J}$ does.

Example 4.3 Recall J^{diff} from Example 4.1. The following instance $\overline{J^{diff}}$ over scheme $\{A, CB, CC\}$ is an instance with two copies of J^{diff} w.r.t. $\{A\}$:

$$\begin{aligned}
A &: \{a_1, a_2\}, \\
CB &: \{\langle b_{11}, a_1, \varepsilon_1\rangle, \langle b_{31}, a_1, \varepsilon_1\rangle, \langle b_{21}, a_2, \varepsilon_1\rangle, \langle b_{41}, a_2, \varepsilon_1\rangle, \\
&\quad \langle b_{12}, a_1, \varepsilon_2\rangle, \langle b_{32}, a_1, \varepsilon_2\rangle, \langle b_{22}, a_2, \varepsilon_2\rangle, \langle b_{42}, a_2, \varepsilon_2\rangle\}, \\
CC &: \{\langle b_{11}, b_{21}, \varepsilon_1\rangle, \langle b_{21}, b_{31}, \varepsilon_1\rangle, \langle b_{31}, b_{41}, \varepsilon_1\rangle, \langle b_{41}, b_{11}, \varepsilon_1\rangle, \\
&\quad \langle b_{12}, b_{22}, \varepsilon_2\rangle, \langle b_{22}, b_{32}, \varepsilon_2\rangle, \langle b_{32}, b_{42}, \varepsilon_2\rangle, \langle b_{42}, b_{12}, \varepsilon_2\rangle\}.
\end{aligned}$$

Although no query expressible in RA + *new* + *loop* can contain a pair (I^{diff}, J^{diff}) as in Example 4.1, it is not difficult to write an RA + *new* + *loop* program P such that $\overline{J^{diff}}$ is a possible result of P applied to I^{diff}. More generally, RA + *new* + *loop* is complete *up to copies*:

Fact 4.4 ([4]) *For each determinate query Q of type $\mathcal{S}_0 \to \mathcal{S} - \mathcal{S}_0$ there is an* RA + *new* + *loop program expressing a query $\overline{Q}$ of type $\mathcal{S}_0 \to \overline{\mathcal{S}} - \mathcal{S}_0$ such that $Q(I, J)$ iff $\overline{Q}(I, \overline{J})$, with $\overline{J}$ an instance with copies of J w.r.t. $\mathcal{S}_0$.*

Proof: Let us briefly review the proof of this important fact. Since Q is r.e., there is a Turing machine M which enumerates Q. So, M takes a natural number as input, and produces a pair (I, J) of (encodings of) instances such that $Q(I, J)$. The range of M is the whole of Q. Then the desired program, on input instance I over $\mathcal{S}_0$, visits pairs (k, ℓ) of natural numbers, in some standard order. For each pair (k, ℓ), k new objects, $o_1, \ldots, o_k$, are created. The collection $\mathcal{C}$ of all instances J over $\mathcal{S}$ that equal I on $\mathcal{S}_0$ and for which $adom(J) - adom(I) = \{o_1, \ldots, o_k\}$, is constructed. The subset $\mathcal{C}'$ of $\mathcal{C}$, consisting of those J for which M on input ℓ produces a pair of instances that is isomorphic to (I, J), is determined. If $\mathcal{C}'$ is empty, the next pair (k, ℓ)

of natural numbers is visited. Otherwise, by the genericity of Q, for each J in $\mathcal{C}'$, we have $Q(I, J)$. Furthermore, by the determinacy of Q, all J in $\mathcal{C}'$ are pairwise isomorphic by a permutation of $\{o_1, \ldots, o_k\}$. Thus, $\mathcal{C}'$ contains all the necessary ingredients from which to construct an instance $\overline{J}$ with copies as desired. ■

It follows that the query CE (for Copy Elimination) of type $\overline{\mathcal{S}} \rightarrow \mathcal{S} - \mathcal{S}_0$, defined by: $\mathrm{CE}(\overline{J}, J)$ if $\overline{J}$ is an instance with copies of $(\mathcal{S}_0 : J|_{\mathcal{S}_0}, \mathcal{S} - \mathcal{S}_0 : J|_{\mathcal{S}-\mathcal{S}_0})$, is not expressible in RA + *new* + *loop*. Copy elimination is a determinate query, and by Fact 4.4, it suffices to add it as a primitive to make RA + *new* + *loop* complete for all determinate queries.

4.2 Semi-deterministic completeness

As argued in the Introduction, the determinate-completeness up to copies of RA + *new* + *loop* is not very satisfactory. In fact, our original motivation for studying semi-determinism was the hope that it could offer a less restrictive setting in which copy elimination could be explained and would appear less ad-hoc. Indeed, an alternative way to look at copy elimination is to consider it as a non-determinate operation which chooses one among several available copies. More formally, we can define a *non-determinate* version of copy elimination, call it $\mathrm{CE}^{\mathrm{nd}}$, as follows. Let C be a relation name not in $\overline{\mathcal{S}}$ with $a(C) = 1$. Then $\mathrm{CE}^{\mathrm{nd}}$ is of type $\overline{\mathcal{S}} \rightarrow C$, and defined by: $\mathrm{CE}^{\mathrm{nd}}(\overline{J}, K)$ if $\overline{J}$ is an instance with copies as in Definition 4.2, and $K(C) = \{\varepsilon_i\}$ for some arbitrarily chosen i.

Using $\mathrm{CE}^{\mathrm{nd}}$ as just defined, we can easily simulate CE as originally defined. It is also readily verified that $\mathrm{CE}^{\mathrm{nd}}$ is semi-deterministic. And, it can be easily expressed by a semi-deterministic TL program. This program first checks whether its input is indeed an instance with copies.[5] Then the Witness operation is applied to the set of all copy identifiers.

This encourages us to generalize the notion of instance with copies to the semi-deterministic setting, in the hope of being able to prove that every semi-deterministic query is expressible by a semi-deterministic TL program.

[5]This initial check is computationally expensive, as it requires checking graph isomorphism. However, in the intended application of copy elimination it is guaranteed that the input has the required format (as with the result of the RA + *new* + *loop* program of Fact 4.4) so the check can in principle be omitted.

Indeed, the strategy which we could follow is to prove (i) that Fact 4.4 can be generalized to the semi-deterministic setting; and (ii) that non-determinate copy elimination, adapted to the semi-deterministic setting, is still expressible by a semi-deterministic TL program. In the remainder of this section, we will show that this strategy works.[6]

First, we generalize the notion of instance with copies as defined in Definition 4.2 to the semi-deterministic setting. To do this, it suffices to note that requirement (ii) in that definition, stating that J_k and J must be $J|_{\mathcal{S}_0}$-isomorphic, is similar to the determinacy condition. Hence, we can generalize the definition of instance with copies in a similar way as we generalized determinacy to semi-determinism. Specifically, we now only require in (ii) that J_k and J are isomorphic. Let us refer to this generalized notion of instance with copies as *instance with semi-deterministic copies*, and to the original notion as *instance with determinate copies*.

We can now observe that the following analogue to Fact 4.4 holds:

Proposition 4.5 *For each semi-deterministic query Q of type $\mathcal{S}_0 \to \mathcal{S} - \mathcal{S}_0$ there is an* RA+*new*+*loop program expressing a query $\overline{Q}$ of type $\mathcal{S}_0 \to \overline{\mathcal{S}} - \mathcal{S}_0$ such that $Q(I, J)$ iff $\overline{Q}(I, \overline{J})$, with $\overline{J}$ an instance with semi-deterministic copies of J w.r.t. $\mathcal{S}_0$.*

Proof: The proof of Fact 4.4 goes through verbatim, except for the point where it is stated that all J in $\mathcal{C}'$ are isomorphic *by a permutation of* $\{o_1, \ldots, o_k\}$. The italicized qualification must now be omitted. ■

Suppose Q is a determinate query as in Fact 4.4, $Q(I, J)$, and $\overline{J}$ is an instance with determinate copies of J. Then trivially, any J' such that $Q(I, J')$ is I-isomorphic to a copy (in fact every copy) contained in $\overline{J}$. The analogue of this property in the semi-deterministic case is not entirely trivial:

Lemma 4.6 *Assume Q is a semi-deterministic query as in Proposition 4.5, $Q(I, J)$, and $\overline{Q}(I, \overline{J})$ as constructed in the proof of said proposition. Then any J' such that $Q(I, J')$ is I-isomorphic to a copy contained in $\overline{J}$.*

Proof: We use the notation from the proof of Proposition 4.5 (and Fact 4.4). Clearly, J' is I-isomorphic to some J'' in $\mathcal{C}$. Furthermore, since Q is semi-

[6]Contrary to the conjecture expressed in a preliminary version of this paper [25].

deterministic, J', and hence also J'', is isomorphic to J by some automorphism of I. But $\mathcal{C}'$ is computed by a (generic) RA + *new* + *loop* program, and therefore if J is in $\mathcal{C}'$, then so is J'', as desired. ■

We may thus conclude that also in the semi-deterministic case, $\overline{J}$ contains "enough" copies. Hence, to express any semi-deterministic query Q, we can in a first phase express Q up to semi-deterministic copies, using the RA + *new* + *loop* program of Proposition 4.5, and then in a second phase choose one of the copies and assign it to the final answer relation, using an obvious TL program. If we can prove that the two-phase program thus obtained is semi-deterministic (again something that was trivial in the determinate case), we can conclude:

Theorem 4.7 *Every semi-deterministic query can be expressed by a semi-deterministic TL program.*

Proof: As just observed, we must prove that choosing among semi-deterministic copies is semi-deterministic. This follows immediately from the following claim:

> Assume Q is a semi-deterministic query as in Proposition 4.5, $Q(I, J)$, and $\overline{Q}(I, \overline{J})$ as constructed in the proof of said proposition. Then any two copies J_1, J_2 contained in $\overline{J}$ are isomorphic by an automorphism of $\overline{J}$.

Let us prove this claim. We know that J_1 and J_2 are isomorphic by an automorphism of I, say f. We must extend f to an automorphism of $\overline{J}$. This means we must find a permutation g of $\mathbf{U}$ that equals f on $adom(J_1)$ such that $g(\overline{J}) = \overline{J}$. Let p be the order of $f|_{adom(I)}$ in the permutation group of $adom(I)$.

If $p = 1$, then J_1 and J_2 are I-isomorphic and the problem is as trivial as in the determinate case (simply define g as f on $adom(J_1)$, f^{-1} on $adom(J_2) - adom(J_1)$, and the identity everywhere else.)

Now suppose $p > 1$. By genericity, we have for each $i = 1, \ldots, p$ that $Q(I, f^i(J_1))$. By Lemma 4.6, for each i there is a copy J_{i+1} contained in $\overline{J}$ such that J_{i+1} is I-isomorphic to $f^i(J_1)$. (Note that J_2 is compatible with this statement.) Furthermore, since $f^p|_{adom(I)}$ is the identity, J_{p+1} can be chosen

to be J_1. So we can define the desired permutation g as f on $adom(J_1)$, and such that $g(J_i) = J_{i+1}$, for $i = 1, \ldots, p$, in the obvious way. ■

Note that we have actually obtained a syntactic, semi-deterministic sublanguage of TL capable of expressing exactly the semi-deterministic queries: namely, the language consisting of all RA + *new* + *loop* programs followed by a semi-deterministic copy elimination step. Of course, this sublanguage is highly artificial.

5 Counting semi-deterministically

In this section, we show that every polynomial-time counting query can be computed efficiently in a semi-deterministic manner. For clarity, we will only consider deterministic queries in this section. So whenever we use the term *query*, we will mean *deterministic query*.

Before coming to the point, we define an intuitive notion of efficiency for programs. There is an obvious implementation of TL programs by nondeterministic Turing machines. We say that a TL program is *efficient* if its thus associated Turing machine is polynomial-time.

5.1 Problem statement

There are queries that are computationally simple, but not expressible in RA + *loop*. The most important class of such queries are those involving counting. A simple example in this class is the boolean parity query: Given input instance I, is the cardinality of $adom(I)$ even? Most queries involving counting cannot be expressed in RA + *loop* because RA + *loop* has a 0-1 law [20].

It is well-known [3] that using non-determinism, more deterministic queries can be expressed. Concretely, every polynomial-time computable query is expressible by an efficient RA + W + *loop* program. This result can be best understood in terms of ordered databases. Ordered databases are databases equipped with a special binary relation containing a linear order of the active domain. It is well-known that every polynomial-time computable query is expressible, *on ordered databases*, by an efficient program in RA + *loop* [3]. Hence, to express an arbitrary polynomial-time query Q

in RA + W + *loop*, it suffices to be able to construct an ordered list of the active domain, which can be done easily by repeatedly choosing elements. In particular, the counting queries can be expressed efficiently in this manner.

A non-deterministic program which generates an arbitrary ordered list of the input database is not semi-deterministic in general: only for completely symmetric databases it is true that any two such lists can be mapped into each other by an automorphism. However, in order to express counting queries, construct an ordering of the entire database domain is not really needed. It is sufficient to generate, for each fragment of the database that has to be counted, a list of *new* objects whose length is equal to the cardinality of the fragment. These new objects are "witnesses" for the fragment to be counted, and whether we count the original fragment or the thus constructed witness list clearly does not matter. Furthermore, a configuration of witness lists can be computed by an efficient TL program, and this program is semi-deterministic, intuitively since any two configurations of witness lists are determinate copies of each other.

So, the purpose of this section is to prove formally that the intuition just expressed is correct, i.e., that every counting query can be expressed efficiently by a semi-deterministic TL program. The qualification *efficiently* is crucial, since otherwise the claim would be trivial. Indeed, we mentioned in Section 4 that RA + *new* + *loop* (or, for that matter, detTL) being a sublanguage of TL can already express *all* deterministic queries, in particular the counting queries. And, RA + *new* + *loop* programs are trivially semi-deterministic. However, the counting queries are not *efficiently* expressible in RA+*new*+*loop*. For example, it follows from results in generic complexity [7] that there is no efficient RA + *new* + *loop* program expressing the parity query.

5.2 Counting semi-deterministically

We still have to specify precisely what is the class of counting queries. A precisely defined extension of fixpoint logic with counting capabilities has been introduced in [12, 13]. This approach, however, involves a two-sorted extension of the relational signature with a sort for natural numbers to express the results of the counting operations. In our object-creating framework, however, an essentially equivalent counting mechanism can be introduced without at the same time introducing natural numbers as well, since a natu-

$$r = \begin{array}{|ll|}\hline a & a_1 \\ b & b_1 \\ b & b_2 \\ c & c_1 \\ c & c_2 \\ c & c_3 \\ \hline \end{array}$$

$$count(r) = \begin{array}{|lll|}\hline a & \alpha_1 & \alpha_2 \\ b & \beta_1 & \beta_2 \\ b & \beta_2 & \beta_3 \\ c & \gamma_1 & \gamma_2 \\ c & \gamma_2 & \gamma_3 \\ c & \gamma_3 & \gamma_4 \\ \hline \end{array}$$

Figure 1: An example of counting

ral number n can be represented by a linked list of newly created objects of length n. This simply corresponds to writing the natural number in unary notation. Before giving a formal definition, we illustrate this operation by means of an example:

Example 5.1 Assume given a binary relation r, interpreted as a parent-child relation. Suppose we want to count the number of children of each parent. A uniform, generic way to do this would be to construct for each parent a list of new objects, the length of which equals the number of his children. This is done by applying the counting operation *count* to r. For example, Figure 1 shows a relation r together with $count(r)$. The Greek letters are newly created objects.

In general, the counting operation is defined as follows:

Definition 5.2 Let $\mathcal{S}$ be a scheme, $I \in inst(\mathcal{S})$, and $R \in \mathcal{S}$ with $X \subseteq \{1, \ldots, a(R)\}$. Let $\pi_X(I(R)) = \{u_1, \ldots, u_p\}$. For each $i = 1, \ldots, p$, define n_i as the cardinality of the set

$$\{t \in I(R) \mid t|_X = u_i\}.$$

Then each possible result of $count_X(R)$ applied to I is a relation of arity $|X|+2$, of the form:

$$\begin{array}{ll} \{ & \langle u_1, \alpha_1^1, \alpha_2^1 \rangle, \ldots, \langle u_1, \alpha_{n_1}^1, \alpha_{n_1+1}^1 \rangle \\ & \vdots \\ & \langle u_p, \alpha_1^p, \alpha_2^p \rangle, \ldots, \langle u_p, \alpha_{n_p}^p, \alpha_{n_p+1}^p \rangle \quad \} \end{array}$$

where the α_j^i are different new objects not in $adom(I)$.

The *count* operation is clearly determinate, and is readily implementable in polynomial-time.

The counting application in Example 5.1 is formally an application of $count_1$. Note that $count_\emptyset(R)$ counts the total number of tuples in R. Note also that the *count* operation contains the *new* operation as a singular case. Indeed,

$$new(R) \equiv \pi_{1,\ldots,a(R)+1} count_{1,\ldots,a(R)}(R).$$

By extending RA + *loop* with the *count* operation, we obtain a language which we denote by RA + *loop* + *count*. By the above remark and the discussion in the previous subsection, this language has exactly the same expressive power as RA + *new* + *loop*, but can express much more queries efficiently. We will refer formally to the class of *counting queries* as consisting of those queries expressed by an efficient RA + *loop* + *count* program.[7]

We now show:

Theorem 5.3 *Every counting query is expressible by an efficient semi-deterministic TL program.*

Proof: We show how to express the *count* operation by an efficient semi-deterministic TL program. It is sufficient to consider applications of the form $count_1(R)$, where R is a binary relation (as in Example 5.1.) Indeed, if R is not binary and we have to compute $count_X(R)$, then by two applications of the *new* operation: $new(\pi_X(R))$ and $new(\pi_{\{1,\ldots,a(R)\}-X}(R))$, we can easily construct, in the relational algebra, an encoding of R into a binary relation which is equivalent for counting purposes. The following program computes

[7]This is not a syntactic definition. The usual approach to syntactically approximate the efficient RA + *loop* programs is to give an inflationary semantics to the loops [3]. This approach does not work in the presence of object creation.

$count_1(R)$ in its answer relation A. (The relational calculus expressions occurring in the program can easily be translated into the relational algebra [3].)

$R' := \pi_{1,3} new(R) \cup new(\pi_1(R))$;
$C_1 := W_2(R')$;
$C_2 := W_2(R' - C_1)$;
$A := \{\langle u, \alpha, \beta\rangle \mid \langle u, \alpha\rangle \in C_1 \text{ and } \langle u, \beta\rangle \in C_2\}$;
$Chosen := C_1 \cup C_2$;
loop [
$C := W_2(R' - Chosen)$;
$Chosen := Chosen \cup C$;
$A := A \cup \{\langle u, \beta, \gamma\rangle \mid \langle u, \gamma\rangle \in C \text{ and } \exists\alpha : \langle u, \alpha, \beta\rangle \in A \text{ and } \neg\exists\gamma' : \langle u, \beta, \gamma'\rangle \in A\}$;
]

Using the familiar parent-child terminology, the R' relation defined in the first assignment can be viewed as being obtained from the R relation by replacing each child by a different, newly created object (and one more.) The remainder of the program then simply orders these "witnesses" (recall the discussion in the first subsection) into lists, as required by the definition of $count_1(R)$. This is accomplished by initializing the lists to length 1, and then completing them by repeatedly choosing the remaining witnesses and appending them to the lists. Hence, the program is correct, and it is clearly also efficient.

It remains to show that the program is semi-deterministic. Observe that in R', any two witnesses of a common parent are logically interchangeable. More precisely, for any two tuples $\langle u, \alpha_1\rangle$ and $\langle u, \alpha_2\rangle$ in R', the transposition $(\alpha_1\ \alpha_2)$ is an automorphism of R'. ¿From this it follows immediately that the first two applications of the Witness operation, in the second and third statements of the program, are semi-deterministic. To show that the loop is semi-deterministic, we similarly observe that before each application of the assignment $C := W_2(R' - Chosen)$, any two witnesses of a common parent remaining in $R' - Chosen$ are logically interchangeable with respect to the whole intermediate result of the computation at that moment. ■

The class of counting queries includes a wide variety of useful queries. In particular, it can be verified that every query expressible in the extension

of fixpoint logic with counting defined in [12, 13] is a counting query in our sense. Rather than proving this claim formally (which is tedious but straightforward), we illustrate it in the remainder of this section.

For example, we can express all "global" boolean counting queries, such as the parity query. This class of queries could be defined as follows. Let E be a relational algebra expression. Let $F(n)$ be a property of natural numbers which can be decided by a Turing machine in polynomial time, given n in unary notation. Then the boolean counting query associated with E and F yields True for an input instance I iff $F(|E(I)|)$ is true. To express this query efficiently and semi-deterministically, first compute $count_{\emptyset}(E)$, which produces an ordered list whose length equals the cardinality of $E(I)$. Then invoke the well-known fact [3] that every polynomial-time property of ordered instances can be expressed efficiently in RA + *loop*.

But we can express much more counting queries than the global boolean ones. The result of a general application of the *count* operation can be interpreted as a relation containing tuples having one entry whose value is a natural number, encoded as a (unary) string. For example, the result relation shown in Figure 1 can be interpreted as the relation

$$\{\langle a, 1\rangle\,,\ \langle b, 2\rangle\,,\ \langle c, 3\rangle\}.$$

By combining the results of different counting operations, we can also construct relations with tuples having multiple entries with natural numbers as values. For example, we can count the number of children of each parent in one relation, and count the number of grandchildren of each parent in another relation. These two relations can then be joined, yielding a relation containing, for each parent, the number of children and the number of grandchildren.

One could now imagine a selection operation for relations containing natural numbers, based on an arbitrary k-ary property $F(n_1, \ldots, n_k)$ of natural numbers computable by a polynomial-time Turing machine, given $n_1, \ldots, n_k$ in unary notation on k input tapes. An example with $k = 1$ would be: *Give all parents with an even number of children.* In this case, F would be the property of being even. An example with $k = 2$ would be: *Give all parents having an equal number of children and grandchildren.* In this case, F would be the property of being equal. It is possible to show that these selection operations are efficiently expressible in RA+*loop*. Indeed, as already mentioned

above, one can simulate in RA + *loop* any polynomial-time Turing machine operating on a unary string (actually, any string). The generalization to a tuple of strings is straightforward. It finally suffices to run the simulations in parallel for each tuple in the relation. The techniques required to do this are straightforward adaptations of the basic simulation technique described in [3].

6 Concluding remarks

1. We would like to have a better understanding of why semi-deterministic computing is desirable, i.e., why it is important to express a query with a semi-deterministic program instead of with just an arbitrarily non-deterministic one. One of the most important characteristics of semi-deterministic programs is that all possible computations behave similarly, as illustrated by Theorem 3.13. This property could be crucial when having to combine different parts of a non-deterministic program which have been executed distributedly on different computers.

2. In Section 5 we have shown that every polynomial-time counting query can be expressed by an efficient, semi-deterministic TL program. Can we do better? More precisely, exactly which polynomial-time computable queries can be thus expressed? In this respect, we point out that on the class of *rigid* instances (i.e., having no non-trivial automorphisms), fixpoint logic is strictly weaker than polynomial time.[8] On rigid structures, semi-deterministic choices among elements are impossible. This strongly indicates that not *all* the polynomial-time computable queries will be expressible by an efficient, semi-deterministic TL program.

3. In the present paper, we have focused on object creation in function of tuples (the *new* operation.) Object creation in function of sets has also been considered in the literature; a simple and uniform operation providing this functionality is the *abstraction* operation [16, 24]. One among the many equivalent ways to define this operation is as follows. Let $I \in inst(\mathcal{S})$, and $R \in \mathcal{S}$ with $a(R) = 2$. If the binary relation $I(R)$ is not an equivalence relation, then the application of $abstr(R)$ to I is undefined. Otherwise, assume

[8] This is because fixpoint logic has a 0-1 law [20], and almost all instances are rigid.

$Z_1, \ldots, Z_k$ is an enumeration of the equivalence classes, and let $\alpha_1, \ldots \alpha_k$ be distinct new objects not in $adom(I)$. Then the binary relation

$$\{\langle o, \alpha_i \rangle \mid o \in adom(I(R)) \text{ and } [o] = Z_i\}$$

is the result of $abstr(R)$ applied to I. ($[o]$ denotes o's equivalence class.) So, while *new* tags tuples, *abstr* tags sets.

Using abstraction, it seems that more queries can be expressed efficiently and semi-deterministically. For example, consider the following semi-deterministic query: given an equivalence relation R, choose for each equivalence class one representative. We conjecture that this query is not expressible by an efficient semi-deterministic TL program. It can however be expressed, efficiently and semi-deterministically, by the following simple program in RA + W + *abstr*:

$C := W_1 abstr(R)$;
$A := \pi_2(C)$.

By Theorem 4.7, abstraction is of course semi-deterministically expressible in TL, but, we conjecture, not efficiently so. We point out that abstraction is not expressible in RA + *new* + *loop* [24].

4. Not every TL program is semi-deterministic or expresses a semi-deterministic query. TL programs are in general not semi-deterministic, and even more, in general do not express a semi-deterministic query. An important issue that has not been considered in the present paper is the design of (preferably efficient) query languages that are guaranteed to express semi-deterministic queries. In investigating this problem, one might take inspiration from the semi-deterministic counting program of Theorem 5.3. A semi-deterministic operation which is implicit in this program is what could be called *swap-choice*: the choice of one representative for each equivalence class of objects that are logically interchangeable. Swap-choice has been studied in [15], where it was shown that swap-choice and the *count* operation are polynomial-time equivalent within RA + *new* + *loop*.

5. In the present paper, we have focused on queries. In [26], we have conducted an initial study on semi-determinism in the context of arbitrary database transformations, including updates. The preliminary conclusion

of this study was that there seems to be a fundamental difference between queries and updates in this respect. Several soundness problems arise and it remains to be seen whether semi-determinism is compatible at all with updates.

Acknowledgments

We thank Serge Abiteboul, Tony Bonner, Marc Gyssens, and Jan Paredaens for interesting discussions and Moshe Vardi for bringing up the question whether every polynomial-time deterministic query is expressible by an efficient semi-deterministic TL program. We also thank the referees for their thorough and insightful comments on an earlier draft of this article, in particular Tova Milo for suggesting a more natural semantics for the looping construct.

References

[1] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*. ACM Press, 1990.

[2] S. Abiteboul. Personal communication, 1990.

[3] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1994.

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. Technical Report 1022, INRIA-Rocquencourt, 1989. Also in *SIGMOD Record*, 18(2):159–173, 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. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1), 1991.

[7] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on Theory of Computing*, pages 209–218. ACM Press, 1991.

[8] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[9] M. Andries and J. Paredaens. A language for generic graph-transformations. In *Graph-Theoretic Concepts in Computer Science*, volume 570 of *Lecture Notes in Computer Science*, pages 63–74. Springer-Verlag, 1992.

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

[11] A.K. Chandra and M.Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[12] E. Grädel and M. Otto. Inductive definability with counting on finite structures. In E. Börger et al., editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 231–247. Springer-Verlag, 1993.

[13] S. Grumbach and C. Tollu. Query languages with counters. In J. Biskup and R. Hull, editors, *Database Theory—ICDT'92*, volume 646 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, 1992.

[14] M. Gyssens. Personal communications, 1992.

[15] M. Gyssens, J. Van den Bussche, and Dirk Van Gucht. Expressiveness of efficient semi-deterministic choice constructs. In S. Abiteboul and E. Shamir, editors, *Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 106–117. Springer-Verlag, 1994.

[16] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[17] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 455–468. Morgan Kaufmann, 1990.

[18] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, 1993.

[19] W. Kim. A model of queries for object-oriented databases. In P. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 423–432. Morgan Kaufmann, 1989.

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

[21] G. Kuper. *The Logical Data Model: A New Approach to Database Logic*. PhD thesis, Stanford University, 1985.

[22] G. Kuper and M. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

[23] G.L. Miller. Graph isomorphism, general remarks. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[24] 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.

[25] J. Van den Bussche and D. Van Gucht. Semi-determinism. In *Proceedings 11th ACM Symposium on Principles of Database Systems*, pages 191–201. ACM Press, 1992.

[26] J. Van den Bussche and D. Van Gucht. Non-deterministic aspects of database transformations involving object creation. In U. Lipeck and B. Thalheim, editors, *Modeling Database Dynamics*, pages 3–16. *Workshops in Computing*, Springer-Verlag, 1993.

[27] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *Proceedings 33rd Symposium on Foundations of Computer Science*, pages 372–379. IEEE Computer Society Press, 1992.