Semi-determinism

(extended abstract)

*Jan Van den Bussche**
University of Antwerp (UIA)
Dept. Math. & Comp. Science
Universiteitsplein 1, B-2610 Antwerp, Belgium
vdbuss@ccu.uia.ac.be

Dirk Van Gucht
Indiana University
Comp. Science Dept.
Bloomington, IN 47405, USA
vgucht@cs.indiana.edu

Abstract

We investigate under which conditions a non-deterministic query is *semi-deterministic*, meaning that two different results of the query to a database are isomorphic. We also consider *uniform* semi-determinism, meaning that all intermediate results of the computation are isomorphic. Semi-determinism is a concept bridging the new trends of non-determinism and object generation in database query languages. Our results concern decidability, both at compile time and at run time; expressibility of the infamous counting queries; and completeness, which is related to the issue of copy elimination raised by Abiteboul and Kanellakis.

1 Introduction

To date, the theory of queries in the context of the relational database model is well understood [Cha88]. In particular, the characteristics of *complete* query languages became clear [AV90, CH80, HS89]. At the same time, some outstanding open problems arose. One interesting example is the *query-$\mathcal{P}$* problem: is there a language that expresses exactly the queries computable in polynomial time? A typical illustration of this problem is the EVEN query, which decides whether a set has even cardinality. This query is but the simplest example of a general class of *counting* queries which frequently arise in practice. While counting in general is clearly in query-$\mathcal{P}$, most generic query languages either cannot count or must do so in an intractable manner [AV91b]. In contrast, when having access to a linear order on the domain objects in the database, the language of fixpoint logic expresses exactly query-$\mathcal{P}$ [Imm86, Var82]. Unfortunately, linearly ordering the database is inconsistent with the widely accepted consistency criterion of *genericity* [CH80], which requires a query to be invariant under permutations of the individual domain objects.

Recently, there has been much interest in *non-deterministic* queries [ASV90, AV90, AV91a, NT89, SZ90]. Here, the requirement of a query being a function from databases to databases is relaxed to a binary relationship between databases. (Or equivalently, a non-deterministic function.) The criterion of genericity can be extended from functions to binary relationships in the canonical way. Most non-deterministic query languages achieve non-determinism via a tool for *choosing* among data objects. Using such a choice operation, it is possible to derive a linear order on the database, by repeatedly choosing the "next" object. Hence, not surprisingly, there exist query languages expressing exactly non-deterministic query-$\mathcal{P}$ [AV91a]. In non-deterministic languages, the capability of deriving a linear order is no longer inconsistent with genericity, as every possible order has equal "probability" of being derived. Subsequent computations may however depend on the particular ordering chosen. This may pose difficulties in practice, when trying to combine the computations of different parts of a query which are distributedly

*Aspirant NFWO.

11th Principles of Database Systems/6/92/San Diego, CA
© 1992 ACM 0-89791-520-8/92/0006/0191...$1.50

computed on different systems using different orderings. In particular, the different results of a non-deterministic query can be non-isomorphic, which we do not believe to be very useful in practice. So, the strategy of circumventing the query-$\mathcal{P}$ problem by employing non-deterministic languages to express ordinary, deterministic queries carries its own disadvantages.

Another recent development is that of *object-generating* queries [AK89, GPVG90, HY90, HY91, KLW90, KV84, VdBP91, Zan89], in the context of object-oriented databases [Kim89]. The criterion of genericity can be generalized to object-generating queries in a natural way. However, in order to keep object generation consistent with genericity, object-generating queries are non-deterministic w.r.t. the particular "choice" of the newly generated objects. In order to capture the non-determinism of such queries, in [AK89, Kup85], the notion of *determinate* query was proposed. Unlike for a plainly non-deterministic query, two different results of a determinate query applied to a database must be isomorphic. Moreover, the isomorphism must be the identity on the objects of the input database. Most, if not all, of the object-generating query languages proposed thus far in the field express only determinate queries. Clearly, determination closely approximates plain determinism. The degree of non-determinism is negligible and only due to the generation of new objects. In particular, the query-$\mathcal{P}$ problem also stands for the determinate case: no determinate query language is known that can count in a tractable manner. Also, it became apparent that object generation has a provocative impact on the expressiveness of query languages. For example, consider the language $RA+gen+loop$ (relational algebra augmented with a standard—determinate—object-generating operation and while-loops). $RA + gen + loop$, when restricted so as to express only deterministic queries, was shown to be complete for all deterministic queries [AV90]. Surprisingly, $RA + gen + loop$ *fails* to be complete for all determinate queries [Abi, AK89, AP92]. More specifically, $RA + gen + loop$ lacks the power of *copy elimination*, as defined in [AK89], in order to be complete.

In this paper, we investigate under which conditions a query is *semi-deterministic*, meaning that two different results of the query to a database are isomorphic, by an isomorphism which leaves the input database invariant (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 relaxation of determination, and in particular, unlike determination, is not *exclusively* tied to object generation. We believe that semi-determinism is a natural upper bound on the amount of non-determinism of a query which makes sense in practice. Intuitively, the only source of non-determinism are the similarities (automorphisms) present in the database. A practical example of a semi-deterministic query is: "Given a set of professor names and a set of student names, return a one-to-one assignment of advising profs to students." We also consider the *uniform* semi-determinism of programs written in non-deterministic query languages, meaning that all intermediate stages of the computation of the program are semi-deterministic, whence all intermediate results are isomorphic. Our results indicate that with the aid of object generation, the advantages of non-determinism w.r.t. expressive power can be realized in a uniformly semi-deterministic manner.

Concretely, our results are the following. In view of the fact that semi-determinism is a relaxation of determination, we give a purely algebraic characterization of when a semi-deterministic query is actually determinate. We show that all possible results of a uniformly semi-deterministic loop program are output after the same number of iterations, in contrast to arbitrarily non-deterministic loop programs, thus providing a formal motivation for uniform semi-determinism. We then show that it is undecidable whether a query expressed in $RA+W$ (relational algebra augmented with the *Witness* choice operation of [AV91a]) is semi-deterministic. Hence, the general problem of guaranteeing semi-determinism is computationally unfeasible. If we have object generation however, we show that the techniques for expressing counting using non-determinism can be modified so as to work guaranteed uniformly semi-deterministically. Specifically, we construct a uniformly semi-deterministic sublanguage of $RA + W + gen + loop$ in which a natural subclass of the counting queries, which we call *global* count-

ing, can be efficiently expressed. We show that the problem of copy elimination of [AK89] can be resolved in a semi-deterministic way. This in turn leads to a uniformly semi-deterministic sublanguage of $RA + W + gen + loop$, expressing exactly the determinate queries. We exhibit a variant of the latter language which expresses exactly the semi-deterministic queries. Finally, we look into the issue of *run-time checking* for semi-determinism, in contrast to the previous results which guarantee semi-determinism at *compile time*. The problem here is to check whether a given program applied to a given database will behave semi-deterministically. We are not aware of a polynomial time algorithm for this problem, and show that it is *(i)* not expressible in RA; and *(ii)* $\mathcal{P}$-reducible to the problem of graph-isomorphism. We also present a stronger form of semi-determinism, which does allow efficient run-time checking (expressible in RA), and is still sufficiently expressive for counting (although not for copy elimination).

2 Preliminaries

We 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. Assuming the existence of an infinitely enumerable domain $\mathcal{O}$ of *objects*, an *instance* I over a scheme $\mathcal{S}$ is an assignment, assigning to each relation name R of $\mathcal{S}$ a finite relation $I(R) \subset \mathcal{O}^{a(R)}$. The *active domain* of an instance I is the set of all objects occurring in it, and is denoted by $adom(I)$. 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 non-deterministic 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 $\mathcal{S}$. A *query of type* $\mathcal{S} \rightarrow A_1, \ldots, A_n$ is a computable, binary relationship $Q \subset inst(\mathcal{S}) \times inst(\mathcal{S} \cup \{A_1, \ldots, A_n\})$ such that: If $Q(I, J)$ then J is equal to I on $\mathcal{S}$; and if $Q(I, J)$ and f is a permutation of $\mathcal{O}$ then $Q(f(I), f(J))$ (*genericity*). Importantly, the composition of two queries (of the appropriate types) is again a query.

We will use the following notation: if γ is a binary relationship, then $\gamma(x)$ stands for the set $\{y \mid \gamma(x, y)\}$. For a query Q and instance I, $Q(I)$ is the set of *possible results* of Q for I.

Queries that are functions are called *deterministic*. A query Q is *object-generating* if there exist I, J such that $Q(I, J)$ and $adom(J) \not\subseteq adom(I)$. Otherwise, the query is called *object-preserving*.

A basic language for expressing deterministic, object-preserving queries is RA (a variant of relational algebra). Let $\mathcal{S}$ be a scheme and let A be a relation name not in $\mathcal{S}$. An *RA assignment* is an expression of the form $A := E$, where E is an expression of the relational algebra using only relation names in $\mathcal{S}$. Such an assignment expresses a query of type $\mathcal{S} \rightarrow A$ in the well-known way. An *RA program* is a sequence of RA assignments $A_1 := E_1; \ldots; A_n := E_n$, such that the i-th assignment is of type $\mathcal{S} \cup \{A_1, \ldots, A_{i-1}\} \rightarrow A_i$. Such a program expresses a query of type $\mathcal{S} \rightarrow A_1, \ldots, A_n$, being the composition of the queries expressed by the assignments. Actually, for any subset $C \subseteq \{A_1, \ldots, A_n\}$ (commonly called a *carrier*), the program can be also used to express a query of type $\mathcal{S} \rightarrow C$, by simply ignoring the relations with name not in C, in the final result.

By genericity, an object-generating query cannot be deterministic. However, it can still be "nearly" deterministic, called *determinate* in [AK89].[1] A query Q is determinate if whenever $Q(I, J_1)$ and $Q(I, J_2)$, then J_1 and J_2 must be *I-isomorphic*, which means that there must be an permutation f of $\mathcal{O}$, that is the identity on $adom(I)$, such that $f(J_1) = J_2$. The composition of two determinate queries is again determinate.

All operations of the relational algebra are object-preserving and deterministic. We can define a natural additional operation, *gen*, which is object-generating and determinate. Let $\mathcal{S}$ be a scheme, $I \in inst(\mathcal{S})$, and $R \in \mathcal{S}$ with $a(R) = \alpha$. A relation r of arity $\alpha + 1$ is a possible result of $gen(R)$ applied to I if r can be obtained from $I(R)$ by extending each tuple t of $I(R)$ with a different additional component $t(\alpha + 1)$ that is not in $adom(I)$. For example, if $\mathcal{S} = \{R\}$ with $a(R) = 2$ and $I(R) = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle\}$,

[1] The definition we give here is adapted to our slightly more general setting.

then a possible result of $gen(R)$ applied to I is $\{\langle a,b,x\rangle, \langle a,c,y\rangle, \langle b,c,z\rangle\}$.

We can extend the language RA to $RA + gen$ by allowing also assignments of the form $A := gen(R)$. The definition of query expressed by an $RA + gen$-program is still the same. We thus obtain a language that can express object-generating, determinate queries.

We can also define an operation, W (*Witness*, [AV91a]), which is object-preserving, but arbitrarily non-deterministic. 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^c-equivalent tuples of $I(R)$ exactly one representative. Here, two tuples are called *X^c-equivalent* if they are equal outside X. In particular, if $X = \{1,\ldots,a(R)\}$, then any two tuples are X^c-equivalent. For example, with $\mathcal{S}$ and I as in the previous example, the only two possible results of $W_2(R)$ applied to I are $r_1 = \{\langle a,b\rangle, \langle b,c\rangle\}$ and $r_2 = \{\langle a,c\rangle, \langle b,c\rangle\}$.

We can extend the language RA (resp. $RA + gen$) to $RA + W$ (resp. $RA + gen + W$) by allowing also assignments of the form $A := W_X(R)$. These languages can express arbitrarily non-deterministic queries.

We end these preliminaries by introducing a version of a loop construct which is commonly used to augment the expressive power of query languages. Let $\mathcal{S}$ be a scheme, and A, B relation names with the same arity not in $\mathcal{S}$. Define the "constant-$\emptyset$" query $Q_{A:=\emptyset}$ of type $\mathcal{S} \to A$ by: $Q_{A:=\emptyset}(I,J)$ if $J(A) = \emptyset$. If Q is a query of type $\mathcal{S} \cup \{A\} \to B$, and n is a natural number, then define the query $loop_{n,B\mapsto A}[Q]$ of type $\mathcal{S} \to A$ as: $Q_{A:=\emptyset}; (Q; \rho_{B\mapsto A})^n$, where ';' stands for composition of binary relationships, and $\rho_{B\mapsto A}$ is the function mapping an instance $I \in inst(\mathcal{S} \cup \{A,B\})$ to the instance $(\mathcal{S} : I|_{\mathcal{S}}, A : I(B))$ over $\mathcal{S} \cup \{A\}$. The exponent n naturally denotes an n-fold composition. Finally, the query $loop_{B\mapsto A}[Q]$ of type $\mathcal{S} \to A$ is defined as follows: $J \in loop_{B\mapsto A}[Q](I)$ if there exists an n such that $J \in loop_{n,B\mapsto A}[Q](I) \cap loop_{n+1,B\mapsto A}[Q](I)$.

We can extend any query language $\mathcal{L}$ to $\mathcal{L} + loop$ by saying that *(i)* if P is a program expressing a query Q of type $\mathcal{S} \cup \{A\} \to B$, then also $loop_{B\mapsto A}[P]$ is a program, expressing the query $loop_{B\mapsto A}[Q]$ of type $\mathcal{S} \to A$; and *(ii)* programs can be composed using ';'.

3 Semi-determinism

A natural and desirable restriction on the amount of non-determinism of a query Q is to require that whenever $Q(I,J_1)$ and $Q(I,J_2)$, then J_1 and J_2 must be isomorphic. We call this requirement *semi-determinism*. By definition, it follows that the isomorphism from J_1 to J_2 must be an automorphism of I.[2] Determinate queries are very restricted semi-deterministic queries, for which this automorphism is the identity on I. Actually, the difference between determination and semi-determinism can be characterized as follows:

Proposition 3.1 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.

Proof: *If:* Assume $Q(I,J_1)$ and $Q(I,J_2)$. Then there is a permutation f such that $f(J_1) = J_2$ and f is an automorphism of I. By the given, there is then 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 $adom(I)$, and define $h := g^{(p-1)}f$. Then $h(J_1) = J_2$, and h is the identity on $adom(I)$.

Only if: Let f be an automorphism of I. By genericity, $Q(I,f(J))$. By determination, 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 $h(J) = J$. ■

Since we defined queries as augmentations, the following holds:

Lemma 3.2 The composition of two semi-deterministic queries is semi-deterministic.

We also state the following lemma, which is used in several proofs, and is also interesting in its own right, although it easily follows from genericity. Informally, it says the "converse" of the definition of semi-determinism.

[2] An automorphism of I is a permutation of $\mathcal{O}$ such that $f(I) = I$.

Lemma 3.3 Let Q be a semi-deterministic query, and assume $Q(I, J)$. Let J' be an arbitrary instance. If there is a permutation f of $\mathcal{O}$ such that $f(I) = I$ and $f(J) = J'$, then also $Q(I, J')$.

The following example shows that arbitrary applications of the W operation are not semi-deterministic in general.

Example 3.4 Let $\mathcal{S}$ be the database scheme $\{R\}$, with $a(R) = 2$. Let I be the instance over $\mathcal{S}$ defined by $I(R) = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle\}$. Consider the $RA+W$-program: $A_1 := \pi_1(R) \cup \pi_2(R)$; $A_2 := W_1(A_1)$. 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 three singleton subsets of $adom(I)$. Among these three 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 permutation of $\{a, b, c\}$ mapping a to b and leaving $I(R)$ invariant. So the program is not semi-deterministic.

It is known [ASV90] 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) actually expresses a deterministic query. Although semi-determinism is less restrictive than plain determinism, the analogue of this result still holds:

Proposition 3.5 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 a program of the form P; $A_2 := W_X(R)$; $A_3 := A_1 - A_2$. 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' for I for which $J(A_3) = \emptyset$. Since the only relation isomorphic to $\emptyset$ is $\emptyset$ itself, it follows that P' is semi-deterministic iff P' is deterministic iff P is deterministic, which is undecidable. ■

The above proof exhibits a class of queries for which semi-determinism simply boils down to plain determinism. This is also the case for *boolean* queries, i.e., queries of type $\mathcal{S} \to A$ where $a(A) = 0$: semi-determinism of a boolean query reduces to determinism.

We also consider *uniform* semi-determinism, which is a property of *programs* rather than queries. A program in the language $RA+W$ or one of its extensions is uniformly semi-deterministic if it is semi-deterministic at every intermediate stage of its execution. Formally, a sequence of assignments $\sigma_1; \ldots; \sigma_n$ is uniformly semi-deterministic if $\sigma_1; \ldots; \sigma_i$ expresses a semi-deterministic query for each $i \leq n$. A loop program $loop[P]$ is uniformly semi-deterministic if for each n, the body of the loop repeated n times, denoted $loop_n[P]$, is uniformly semi-deterministic.

We can show the following desirable property of uniformly semi-deterministic loops, which does not hold for arbitrarily non-deterministic loops. Let P be a program in $RA + W$ or one of its extensions, expressing the query Q. Consider the loop program $loop[P]$. Assume instance J is a possible result of $loop[P]$ for I. The *output stage* of J is defined as the smallest natural number n such that $J \in loop_n[P](I) \cap loop_{n+1}[P](I)$.

Proposition 3.6 If $loop[P]$ is uniformly semi-deterministic, then all possible results of $loop[P]$ for I are output in the same stage.

Proof: Let J_1, J_2 be possible results of $loop_{B \mapsto A}[Q]$ for I. By semi-determinism of $loop_{B \mapsto A}[Q]$, J_1 and J_2 are isomorphic. For $i = 1, 2$, let n_i be the output stage of J_i. Then $J_1 \in loop_{n_1, B \mapsto A}[Q](I) \cap loop_{n_1+1, B \mapsto A}[Q](I)$. By uniform semi-determinism of $loop_{B \mapsto A}[P]$ and Lemma 3.3, also $J_2 \in loop_{n_1, B \mapsto A}[Q](I) \cap loop_{n_1+1, B \mapsto A}[Q](I)$. Hence, $n_2 \leq n_1$. By symmetry, also $n_1 \leq n_2$. ■

4 Semi-determinism and counting

It is known that the deterministic fragment of $RA+W+loop$, restricted such that loops are inflationary, expresses exactly query-$\mathcal{P}$. In particular, using non-deterministic techniques, "counting" can be simulated in a tractable manner [ASV90]. Let us see for example how the EVEN query can be expressed in $RA + W + loop$.

Example 4.1 Let $\mathcal{S}$ be a scheme and $R \in \mathcal{S}$. The deterministic, boolean query EVEN(R) of type $\mathcal{S} \to E$, with $a(E) = 0$, is defined by:

EVEN$(R)(I,J)$ iff $\#I(R)$ is even and $J(E)$ = True, or $\#I(R)$ is odd and $J(E)$ = False.[3]

The following $RA + W +$ *loop*-program, denoted by $P_{EVEN(R)}$, efficiently expresses EVEN(R): (A is a relation name of the same arity as R)

loop$_{B \mapsto A}$ [
  $A_1 := R - A$;
  if $\#A_1 = 1$ or $A_1 = \emptyset$ **then** $B := A$
  else $A_2 := W_X(A_1)$; $A_3 := W_X(A_1 - A_2)$;
    $B := A \cup A_2 \cup A_3$ **fi**
];
if $R - A = \emptyset$ **then** $E :=$ True **else** $E :=$ False

The if-then-else tests used in the program are only shorthands, and are expressible in RA. The parameter X of the W operations is $\{1, \ldots, a(R)\}$. Intuitively, what happens is that repeatedly pairs of tuples of the R-relation are chosen and collected in A, until only one tuple remains (and then EVEN=False) or no remain (and then EVEN=True).

Program $P_{EVEN(R)}$ is clearly not uniformly semi-deterministic: since tuples are chosen arbitrarily, the intermediate results will not be isomorphic in general. By making use of object generation however, the program can be adapted so as to make it uniformly semi-deterministic. Let $\mathcal{S}$ be a scheme and $R \in \mathcal{S}$ with $a(R) = \alpha$. Consider the following $RA +$ *gen* program $P_{GEN(R)}$, expressing a query of type $\mathcal{S} \to R'$ with $a(R') = 1$: $A_1 := gen(R)$; $R' := \pi_{\alpha+1}(A_1)$. Each possible result of $P_{GEN(R)}$ applied to an instance I has as R'-relation a set of newly generated objects, exactly as many as the number of tuples in $I(R)$. We have the following important property. For an arbitrary instance J, call two objects $o_1, o_2 \in adom(J)$ *equivalent in* J if there is an automorphism of J mapping o_1 to o_2.

Lemma 4.2 Let J be a possible result of $P_{GEN(R)}$ applied to I. Then any two objects in $J(R')$ are equivalent in J.

Considering now the $RA + W +$ *loop*-program $P_{EVEN(R')}$, we obtain:

Proposition 4.3 The $RA + W +$ *gen* $+$ *loop*-program $P_{GEN(R)}$; $P_{EVEN(R')}$ is uniformly semi-deterministic and expresses EVEN(R).

[3] We use here the standard encoding of True (resp. False) by the non-empty (resp. empty) relation of arity 0.

This follows because the choices made in $P_{EVEN(R')}$ are now among equivalent objects, by Lemma 4.2, and therefore have a semi-deterministic effect.

The above reasoning on expressing EVEN in a uniformly semi-deterministic manner can be generalized to a whole class of counting queries. For our purposes, we define a *counting query* as a query defined by a first-order logic formula, in which additionally variables standing for natural numbers can be used, over which certain $\mathcal{P}$-computable predicates can be evaluated (like $<$, $+$ or $\times$), and which can be combined with the normal domain variables by the *counting quantifiers* [Imm86] of the form $(Qn\,x)\Phi$, where Q is a quantifier, n a natural number variable, x a domain variable, and Φ a formula in which x occurs free. We call a counting query *global* if in each counting-quantified formula $(Qn\,x)\Phi$, x is the *only* free variable in Φ. For example, EVEN(R) is global, since it is expressible as: $(\exists! j\,x)R(x) \wedge (\exists i < j) 2 \times i = j$. For another example, $(\exists! j_1\,x)R_1(x) \wedge (\exists! j_2\,x)R_2(x) \wedge j_1 < j_2$, expressing that R_1 has fewer elements than R_2, is also global. We can show:

Proposition 4.4 Every global counting query can be uniformly semi-deterministically expressed in $RA + W +$ *gen* $+$ *loop*.

Proof sketch: First, we observe that global counting queries can be handled in $RA + W +$ *loop* in a "canonical" manner. Each counting quantification $(Qn\,x)\Phi(x)$ is handled by inducing a linear order on the set $\{x \mid \Phi(x)\}$ (by making repetitive choices), after which all counting and subsequent predicates on natural numbers can be evaluated without making use of the W operation. Now it can be seen that the canonical program P for any global counting query can be adapted using object generation so as to make it uniformly semi-deterministic, in an analogous way as was shown in Section 4 for EVEN. ■

We have an effective and efficient construction from a given global counting query CQ to a uniformly semi-deterministic $RA + W +$ *gen* $+$ *loop*-program expressing CQ. This construction never introduces the use of object generation within loops. This is important: it is known that if object generation and looping interact, an uncontrollable increase in complexity results. Here, if we count

within a loop, the object generation used to express the counting will not interact with the loop, whence the complexity will not increase. (This is a refinement of the notions of *recursion freedom* and *no recursion through oid creation* of [AK89, HY90].)

An interesting question is to which extent the requirement of globality in Proposition 4.4 is tight. An example of a non-global counting query is: "Give the parents with an even number of children". See also the discussion in Section 7.

5 Complete languages and semi-determinism

The language $RA+gen+loop$ is a very powerful, determinate language, and the obvious candidate for being complete for all determinate queries. However, this is not the case. Consider a relation name A of arity 1. Let B, C be two relation names of arity 2. A query Q of type $\{A\} \rightarrow B, C$ is called *difficult*[4] if it is determinate and $Q(I^{diff}, J^{diff})$. 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 newly generated 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 generated such that two opposite b-nodes are associated to a common a-node through the B-relation. It can be shown that difficult queries are not expressible in $RA + gen + loop$, and hence this language is not determinate-complete.

This problem can be put in a more general framework using the notion of *instance with copies* [AK89]. 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 $\overline{J}$ is called 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 $k = 1, \ldots, n$;

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

such that

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

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 is equal to J on $\mathcal{S}_0$, and hence also $\overline{J}$ is.

For example, the following instance $\overline{J^{diff}}$ over scheme $\{A, CB, CC\}$ is an instance with two copies of J^{diff} w.r.t. $\{A\}$:

$$\begin{array}{l}(A : \{a_1, a_2\},\\ CB :\\ \{\langle b_{11}, a_1, \varepsilon_1\rangle, \langle b_{31}, a_1, \varepsilon_1\rangle, \langle b_{21}, a_2, \varepsilon_1\rangle, \langle b_{41}, a_2, \varepsilon_1\rangle,\\ \langle b_{12}, a_1, \varepsilon_2\rangle, \langle b_{32}, a_1, \varepsilon_2\rangle, \langle b_{22}, a_2, \varepsilon_2\rangle, \langle b_{42}, a_2, \varepsilon_2\rangle\},\\ CC :\\ \{\langle b_{11}, b_{21}, \varepsilon_1\rangle, \langle b_{21}, b_{31}, \varepsilon_1\rangle, \langle b_{31}, b_{41}, \varepsilon_1\rangle, \langle b_{41}, b_{11}, \varepsilon_1\rangle,\\ \langle b_{12}, b_{22}, \varepsilon_2\rangle, \langle b_{22}, b_{32}, \varepsilon_2\rangle, \langle b_{32}, b_{42}, \varepsilon_2\rangle, \langle b_{42}, b_{12}, \varepsilon_2\rangle\}).\end{array}$$

Although no query expressed in $RA + gen + loop$ can contain a "difficult pair" (I^{diff}, J^{diff}), it is an easy exercise to write a program P in $RA + gen + loop$ such that $\overline{J^{diff}}$ is a possible result of P applied to I^{diff}. More generally, $RA+gen+loop$ is complete *up to copies* [AK89]:

Fact 5.1 For each determinate query Q there is a program P in $RA + gen + loop$ expressing a query $\overline{Q}$ such that $Q(I, J)$ iff $\overline{Q}(I, \overline{J})$, with $\overline{J}$ an instance with copies of J.

Using the same notations as above, it follows that the query CE (for Copy Elimination) of type $\overline{\mathcal{S}} \rightarrow R_1, \ldots, R_p$, where $\mathcal{S} - \mathcal{S}_0 = \{R_1, \ldots, R_p\}$, defined by: $\mathrm{CE}(\overline{J}, J)$ if $\overline{J}$ is an instance with copies of $(\mathcal{S}_0 : J|_{\mathcal{S}_0}, R_1 : J(R_1), \ldots, R_p : J(R_p))$, is not expressible in $RA + gen + loop$. Note that CE is a determinate query. Furthermore, to make $RA + gen + loop$ complete for all determinate queries, it suffices to add a construct for expressing the CE query. The obvious candidate is the W operation. We can show:

[4] "Difficult" queries were discovered by S. Abiteboul, and presented in his tutorial on query languages for object-oriented databases at PODS 1991.

Proposition 5.2 CE can be expressed by a *uniformly semi-deterministic* program in $RA + gen + W$.

Proof sketch: First, the program checks whether its input indeed is an instance with copies of another instance. If not, an infinite loop is entered. (If this initial check is not needed, we do not need the *loop* construct; it is with this understanding that the proposition is stated.) Using W, exactly one of the copy identifiers, say ε_k, is chosen. Using object-generation, exactly one copy of J_k is generated. The uniform semi-determinism follows from the fact that all copy identifiers are equivalent in $\overline{J}$. Therefore, the choice of ε_k is semi-deterministic. ■

As an immediate corollary, we obtain a sublanguage of $RA + W + gen + loop$, expressing exactly the determinate queries. Indeed, by Fact 5.1, each determinate query can be expressed up to copies by a program in $RA + gen + loop$. It now suffices to attach to this program the program of Proposition 5.2 to eliminate the copies. Note that each program in this language contains exactly one, semi-deterministic application of W.

A variation of the above idea leads to a sublanguage of $RA+W+gen+loop$, expressing exactly the semi-deterministic queries. Fact 5.1 can be modified to show that for each *semi-deterministic* query Q there is a program in $RA+gen+loop$ expressing a query $\overline{Q}$ such that for each instance I for which $Q(I) \neq \emptyset$, there exist instances J and $\overline{J}$ such that $Q(I,J)$, $\overline{Q}(I,\overline{J})$, and $\overline{J}$ is an instance with copies of J. As for determinate queries, we attach to this program a copy elimination phase. But now we also attach a third phase, which computes all automorphisms of the original input database, chooses one of them, and applies the chosen automorphism to the output of the copy elimination phase. This can be done in $RA + W + gen + loop$ by encoding the automorphisms as newly generated objects. Denote the thus obtained program as P_Q^{SD}. It can be shown using Lemma 3.3 that P_Q^{SD} indeed expresses Q. Hence:

Proposition 5.3 The language $\{P_Q^{SD} \mid Q \text{ semi-deterministic}\}$ expresses exactly the semi-deterministic queries.

Observe that every P_Q^{SD} contains exactly two applications of the W operation. The first one, choosing one of the copies in the copy elimination phase, is semi-deterministic, by Proposition 5.2. However, the second one, choosing one of the automorphisms in the third phase, is not semi-deterministic, since two objects representing two automorphisms are generally not equivalent. Hence, P_Q^{SD}, although expressing a semi-deterministic query, is not *uniformly* semi-deterministic. This raises the intriguing:

Problem Can every semi-deterministic query be expressed by a uniformly semi-deterministic program? In the above reasoning, this reduces to: Can the effect of the third phase be realized in a uniformly semi-deterministic manner?

We conjecture the negative; see the discussion in Section 7.

6 Run-time checking of semi-determinism

In the previous sections we saw several classes of programs expressed in one of the extensions of $RA + W$ that were guaranteed to be semi-deterministic "at compile time". We now study the "run time" checking of semi-determinism: Given a program P containing an occurrence of the W operation, and given an instance I, when applying P to I, does the application of W have a semi-deterministic effect for I? For simplicity, we restrict ourselves to applications of W of the form $W_1(U)$, where U is a unary relation. Such applications simply choose among the individual objects in a set.

Let R be a relation name of arity 2, and reconsider the program P of type $\{R\} \rightarrow A$, already mentioned in Example 3.4: $A_1 := \pi_1(R) \cup \pi_2(R)$; $A := W_1(A_1)$. So, when applied to an instance I, P chooses one object from $adom(I)$. Clearly, P is semi-deterministic for I iff I is *transitive*, i.e., for any pair of objects $o_1, o_2 \in adom(I)$, o_1 and o_2 are equivalent in I. We are not aware of a polynomial time algorithm for checking transitivity. Using Ehrenfeucht-Fraïssé games, we can show:

Proposition 6.1 Transitivity (seen as a boolean query) is not expressible in RA.

We would like to know whether transitivity is expressible in $RA + loop$. Furthermore, we observe:

Proposition 6.2 Transitivity is $\mathcal{P}$-reducible to Graph Isomorphism.

We can define a stronger form of semi-determinism, *swap* semi-determinism, which is more practical for run-time checking. In general, let $\mathcal{S}$ be a scheme and let $U \in \mathcal{S}$ with $a(U) = 1$. Consider the application $W_1(U)$. We know that this application is semi-deterministic for an instance I, if for any pair of objects $o_1, o_2 \in I(U)$, o_1 and o_2 are equivalent in I. We now call the application *swap semi-deterministic* if any such o_1 and o_2 are *swap-equivalent* in I, meaning that the permutation $swap[o_1, o_2]$ which exchanges o_1 and o_2 and fixes everything else is an automorphism of I. If P is indeed swap semi-deterministic for I, then we call I *swap-transitive*. Clearly, swap semi-determinism is stronger than semi-determinism. We have:

Proposition 6.3 Swap-transitivity (seen as a boolean query) is expressible in RA.

Proof idea: Two vertices x and y of a graph r are swap-equivalent if the following predicate, definable in the relational calculus, is true for x and y: $(\forall z : z \neq x \wedge z \neq y)((r(z,x) \leftrightarrow r(z,y)) \wedge (r(x,z) \leftrightarrow r(y,z))) \wedge (r(x,x) \leftrightarrow r(y,y)) \wedge (r(x,y) \leftrightarrow r(y,x))$. ■

We can now define *uniformly swap semi-deterministic* programs similarly as we defined uniformly semi-deterministic programs. The results of Section 4 on counting carry over from uniform semi-determinism to uniform swap semi-determinism. Indeed, it can be shown that Lemma 4.2 also holds for swap-equivalence. (A problem here is how to devise programs that are uniformly swap semi-deterministic; see the discussion in Section 7.) On the contrary, the results of Section 5 do *not* carry over to the swap case. Indeed: copy identifiers in an instance with copies are equivalent, but not swap-equivalent. This may indicate that no language that can be efficiently run-time checked for semi-determinism is powerful enough to express copy elimination.

7 Discussion

This section contains some additional ideas, which will be discussed in more detail in the full paper.

1. Observe that in the discussion preceding Proposition 5.3, instead of choosing one particular automorphism in the third phase, one can apply *all* automorphisms of I to J. This yields an instance $\hat{J}$ "with copies", where the notion "with copies" is a generalization of the original notion defined in Section 5. This generalization is of the same nature as the way in which semi-determinism is a generalization of determination. Thus, we find that the situation of completeness up to copies in the determinate case has an analogue on the uniform semi-deterministic level. (Assuming that the answer to the open problem stated at the end of Section 5 is negative, as we conjecture. Confirming this conjecture requires finding an analogue, for the uniformly semi-determinstic case, of the "difficult" queries.)

2. The data model used in the present paper is the standard, relational model. It is interesting to cast our treatment in a more complex data modeling setting, e.g., one where set values are first-class citizens. Moving to such a data model can be equivalently perceived as adding, besides the standard object generation operation *gen*, an extra object generation operation to the language, called *abstraction*, introduced in [GPVG90] and further studied in [VdBP91]. It turns out that using abstraction, also non-global counting problems can be expressed in a uniformly semi-deterministic manner, thus extending the results of Section 4.

3. The previous comment is also related to the issue of swap semi-determinism. Indeed, it turns out that using the abstraction operation, certain applications of the W operation can be guaranteed to be swap semi-deterministic. Actually, the converse holds as well, in the following sense. Define a semi-deterministic *swap choice* operation, which chooses for each equivalence class of swap-equivalent objects one representative. Then, in the context of $RA + gen$, this swap choice is equivalent to the abstraction operation. We conjecture that swap choice cannot be expressed in $RA + W + gen + loop$. In view of the just mentioned equivalence, this would be an enforcement of

the result of [VdBP91] that abstraction cannot be expressed in $RA + gen + loop$.

References

[Abi] S. Abiteboul. Personal communication. 1990.

[ACM90] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems.* ACM Press, 1990.

[ACM91] *Proceedings of the Tenth ACM Symposium on Principles of Database Systems.* ACM Press, 1991.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, number 18:2 in SIGMOD Record, pages 159–173. ACM Press, 1989.

[AP92] M. Andries and J. Paredaens. A language for generic graph-transformations. In *Graph-Theoretic Concepts in Computer Science, Proceedings of the International Workshop WG 91*, number 570 in Lecture Notes in Computer Science, pages 63–74. Springer-Verlag, 1992.

[ASV90] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In ACM [ACM90], pages 218–229.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2), 1990.

[AV91a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1), 1991.

[AV91b] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on Theory of Computing*, 1991.

[CH80] A. Chandra and D. Harel. Computable queries for relational database systems. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[Cha88] A. Chandra. Theory of database queries. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1988.

[GPVG90] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In ACM [ACM90], pages 417–424.

[HS89] R. Hull and Y. Su. Untyped sets, invention, and computable queries. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 347–359. ACM Press, 1989.

[HY90] 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.* Morgan Kaufmann, 1990.

[HY91] R. Hull and M. Yoshikawa. On the equivalence of database restructurings involving object identifiers. In ACM [ACM91], pages 328–340.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, (68):86–104, 1986.

[Kim89] 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.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14, Dept. Comp. Science, SUNY Stony Brook, 1990.

[Kup85] G. Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford University, 1985.

[KV84] G. Kuper and M. Vardi. A new approach to database logic. In *Proceedings of the Third ACM Symposium on Principles of Database Systems*, pages 86–96. ACM Press, 1984.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[SZ90] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In ACM [ACM90], pages 205–217.

[Var82] M. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[VdBP91] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In ACM [ACM91], pages 291–299.

[Zan89] C. Zaniolo. Object identity and inheritance in deductive databases—an evolutionary approach. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, pages 2–19. Elsevier Science Publishers, 1989.