Expressiveness of Efficient Semi-Deterministic Choice Constructs*

Marc Gyssens[1], Jan Van den Bussche**[2], and Dirk Van Gucht[3]

[1] University of Limburg (LUC), Dept. WNI, B-3590 Diepenbeek, Belgium,
e-mail: marc.gyssens@uia.ac.be
[2] University of Antwerp (UIA), Dept. WISINF, B-2610 Antwerp, Belgium,
e-mail: jan.vandenbussche@uia.ac.be
[3] Indiana University, Computer Sci. Dept., Bloomington, IN 47405, USA,
e-mail: vgucht@cs.indiana.edu

Abstract. Recently, Abiteboul and Kanellakis introduced the notion of *determinate query* to describe database queries having the ability to create new domain elements. As there are no natural determinate-complete query languages known, more restrictive (the *constructive queries*) and more general (the *semi-deterministic queries*) notions of query were considered. Here, we show that the advantage of the second approach over the first is not so much in increased expressiveness, but in the ability of expressing queries more efficiently.

1 Introduction

Over a decade ago, Chandra and Harel [7] proposed a language-independent notion of completeness with respect to domain-preserving database queries: a query language is *complete* if it can express all Turing-computable partial functions from databases to databases that are invariant under every permutation of the universe of possible domain values. The latter criterion, nowadays known as *genericity* (e.g., [4]), ensures that queries can be computed in a way independent of the encoding of the universe of possible domain values.

More recently, object-oriented database applications motivated researchers to relax domain preservation by allowing the appearance of new objects in the result of a query [2]. Extending the notion of query in this sense, however, necessitates a significant revision of the definition of Chandra and Harel. Indeed, queries which introduce new objects in their results are necessarily non-deterministic due to the genericity criterion. Obviously, there is a need to control the degree of non-determinism one wishes to allow.

An important proposal to revise the definition of Chandra and Harel came from Abiteboul and Kanellakis [2]. They introduced the notion of *determinate*

* The following extended abstract presents research results of the Belgian Incentive Program "Information Technology" – Computer Science of the future, initiated by the Belgian State – Prime Minister's Service – Science Policy Office. The scientific responsability is assumed by its authors.

** Research Assistant of the Belgian National Fund for Scientific Research.

query: a generic, non-deterministic query for which the possible results of the query applied to a given input database are equal *up to renaming* of the new objects. In addition, Abiteboul and Kanellakis introduced the language IQL to express determinate queries. Contrary to what was at first expected, IQL is *not* complete for the determinate queries; its incompleteness stems from the fact that the definition of determinacy does not take into account that new objects in IQL are always created in terms of existing objects. Therefore, Andries and the present authors defined a more restrictive notion of query, called *constructive query* [18], and showed that IQL is complete with respect to the constructive queries. It should be noted that this completeness result is by no means specific to IQL, as IQL is equivalent to any minimal language that can express first-order queries, object creation, and unbounded looping. Therefore, many other object-creating query languages that have been considered (e.g., [11, 12, 13, 15, 18]) are essentially equivalent to IQL.

Extending IQL to a complete language requires the introduction of an involved determinate *copy elimination* mechanism [2, 8], suggesting that the concept of determinate query is perhaps less natural than originally anticipated. From this viewpoint, restricting the class of determinate queries to the class of constructive queries is a way to obtain a more natural class of queries.

An equally valid approach to obtain a more natural class of queries is to *extend* the class of determinate queries by also allowing certain strictly non-deterministic queries. The observation that many important strictly non-deterministic queries involve choices based on the symmetries of the input database and therefore use only a limited form of non-determinism led to the notion of *semi-deterministic query* [17]. A semi-deterministic query is a generic, non-deterministic query for which the results of the query applied to a given input database are *isomorphic via isomorphisms that are symmetries* (i.e, automorphisms) *of the input database*. In particular, the most obvious way to perform the copy elimination required to make IQL determinate-complete is not determinate but non-deterministic, notably by choosing one of the isomorphic copies.

Of course, we must ask what is gained by taking the semi-deterministic approach rather than the constructive. To evaluate the semi-deterministic approach we need to consider languages in which semi-deterministics queries can be expressed. The most obvious way to obtain such a language is augmenting IQL with a choice operation. Abiteboul and Vianu [3] considered a choice operation, called **witness**, in the context of general non-deterministic queries. In [17] it was shown, however, that *(i)* it is undecidable to check at compile time whether a program in IQL + **witness** expresses a semi-deterministic query, and *(ii)* run-time checking for semi-determinism is polynomial-time equivalent to checking graph isomorphism.

These negative results should not be held against the notion of semi-determinism as queries making symmetry-based choices are often intractable precisely because they involve finding these symmetries. On the other hand, there is a large class of natural queries requiring symmetry-based choices that can be accomplished in polynomial time. For instance, it was shown in the full version of

[17] that the polynomial-time *counting queries* (e.g., [9, 10]) can be expressed efficiently by uniformly semi-deterministic IQL + **witness** programs. In view of the importance of this class of queries, it is the purpose of the present paper to characterize a minimal class of semi-deterministic queries in which the polynomial-time counting queries can be expressed efficiently.

Thereto, we augment IQL with a choice operation which is much more restrictive than **witness**, called **swap-choice**. The **swap-choice** operation was first suggested in [17] and selects one representative for each class of swap-equivalent objects in the database; two objects are called swap-equivalent if the transposition of these objects is an automorphism of the database. Swap-equivalence is first-order definable and therefore in polynomial time. As a result, **swap-choice** is efficiently computable. Unlike **witness**, **swap-choice** is moreover guaranteed to act semi-deterministically. In this paper, we argue that there are no reasonable alternatives for this operation with equally desirable properties.

Next, we examine the precise expressive power of IQL + **swap-choice**. More concretely, we provide a characterization which is reminiscent of the characterization of IQL as a constructive-complete language. We establish that the introduction of **swap-choice** provides little extra computational power, and in particular, that **swap-choice** provides no extra power at all when applied to newly-created objects only.

Hence, the advantage of the semi-deterministic approach over the constructive one is not that more queries can be expressed, but that more queries can be expressed *efficiently*. Indeed, the arguments developed in the full version of [17] imply that the polynomial-time counting queries can be expressed efficiently in IQL + **swap-choice**, even when the applications of **swap-choice** are only applied to newly-created objects. Although all polynomial-time counting queries are constructive, most of these nevertheless cannot be expressed efficiently in IQL alone [4].

In this paper we show a much stronger result. We show that IQL + counting and IQL + **swap-choice** applied to newly-created objects only are polynomial-time equivalent, thereby substantiating our earlier claim: the queries expressible in IQL + **swap-choice** form the smallest natural class of semi-deterministic queries containing the constructive queries in which the polynomial-time counting queries can be expressed efficiently.

This result sheds new light on the polynomial-time counting queries. It is well-know that every polynomial-time deterministic query can be computed efficiently in the relational calculus augmented with iteration *on ordered databases* [14, 19]. In the context of general non-determinism, the **witness** operation can be used to compute an order on the database objects. Hence, in the absence of order, all polynomial-time queries can be computed efficiently in IQL + **witness**. Our equivalence result shows that, for an important subclass of the polynomial-time queries, namely the polynomial-time counting queries, a very limited form of non-determinism is necessary and sufficient for efficient computation.

This abstract is organized as follows. In Section 2, we review the definitions of determinate and constructive object-creating queries and present the query

language CQL as an abstract formulation for IQL. In Section 3, we present the semi-deterministic queries and the **swap-choice** operation. In Section 4, we characterize the expressive power of CQL + **swap-choice**. In Section 5, we finally show the polynomial-time equivalence of swap-choice applied to newly-created objects only and counting and discuss the ramifications of this result.

2 Preliminaries

We assume the existence of an infinitely enumerable set of *relation names*. Each relation name R has an associated *arity* $\alpha(R)$. A *database scheme* is a finite set of relation names. We further assume there is an infinitely enumerable universe $\mathcal{O}$ of *objects*. A *database instance* I over a database scheme $\mathcal{S}$ assigns to each R in $\mathcal{S}$ a finite relation $R^I \subset \mathcal{O}^{\alpha(R)}$. In the sequel, $inst(\mathcal{S})$ denotes the set of all instances over scheme $\mathcal{S}$, $dom(I) \subseteq \mathcal{O}$ the set of all objects appearing in instance I, and $Aut(I)$ the automorphism group of I.

We now turn to *queries*. The following definition is adapted from [2]:

Definition 1. Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes. A *determinate query* Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a recursively enumerable binary relationship $Q \subset inst(\mathcal{S}_{\text{in}}) \times inst(\mathcal{S}_{\text{out}})$ such that

1. if $Q(I, J)$, then $R^I = R^J$ for all $R \in \mathcal{S}_{\text{in}}$;
2. if $Q(I, J)$, and f is a permutation of $\mathcal{O}$, then also $Q(f(I), f(J))$; and
3. if $Q(I, J_1)$ and $Q(I, J_2)$, then J_1 and J_2 are isomorphic via an isomorphism that is the identity on $dom(I)$.

Item 1 of Definition 1 states that a determinate query does not cause side-effects on the input database; item 2 states the genericity criterion; and item 3 states that two results of a determinate query are equal up to "renaming" of the new objects.

To obtain results that are not tied to a particular language, we propose the general query language CQL as an abstract formulation for IQL, the language introduced in [2] to express determinate queries.[4] Programs in CQL are built from *FO-statements*, *new-statements*, *abstraction-statements*, and *while-statements*.

Syntactically, an *FO-statement* is of the form $R := \Phi$, in which R is a k-ary relation name, and Φ a k-ary relational-calculus-like first-order (FO-) expression. Given a scheme $\mathcal{S}$ over which Φ is defined, the FO-statement $R := \Phi$ defines a binary relationship $Q \subset inst(\mathcal{S}) \times inst(\mathcal{S} \cup \{R\})$ in the obvious way.

We need two types of statements to create new objects: *new-statements* associate new domain elements to tuples, and *abstraction-statements* to sets.

To define new-statements, we use the **new** operation. Let $\mathcal{S}$ be a scheme, and Φ a k-ary FO-expression defined over $\mathcal{S}$. Let I be an instance over $\mathcal{S}$,

[4] The language CQL is a variation on the language FO + **powerset** + **while** in [18] and is better suited for the complexity arguments that are the main focus of this paper.

and $\Phi(I) = \{t_1, \ldots, t_n\}$ with $t_1, \ldots, t_n$ k-ary tuples. Then **new** $\Phi(I)$ non-deterministically selects n different new objects $o_1, \ldots, o_n$ (i.e., objects not having occurred previously in the computation) and yields the $k+1$-ary relation $(\{t_1\} \times \{o_1\}) \cup \cdots \cup (\{t_n\} \times \{o_n\})$, in which each tuple of $\Phi(I)$ is tagged by a unique, new object. Finally, a *new-statement* has the form $R :=$ **new** Φ, with R a relation name and Φ as above. Its semantics is obvious.

To define abstraction-statements, we use the **abstraction** operation. Let $\mathcal{S}$ be a scheme, and Φ a *binary* FO-expression defined over $\mathcal{S}$. Let I be an instance over $\mathcal{S}$, and $\Phi(I) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. The binary relation $\Phi(I)$ can be interpreted as a set-valued function, say f. The operation **abstraction** $\Phi(I)$ non-deterministically selects new objects $o_1, \ldots, o_n$ satisfying $o_i = o_j$ if and only if $f(x_i) = f(x_j)$, and yields the binary relation $\{(x_1, o_1), \ldots, (x_n, o_n)\}$ in which each object x_i is associated to the tag of the corresponding set $f(x_i)$. Finally, an *abstraction-statement* has the form $R :=$ **abstraction** Φ, with R a relation name and Φ as above. Its semantics is obvious.

Example 1. Consider Figure 1. Let I be the instance over the scheme $\{R\}$. The instance over the scheme $\{R, S\}$ is a possible result of $S :=$ **new** $\{(x, y) \mid R(x, y) \wedge x \neq y\}$ applied to I, and the instance over the scheme $\{R, T\}$ is a possible result of $T :=$ **abstraction** $\{(x, y) \mid R(x, y)\}$ applied to I. In the relation T, the object "1" represents the set $\{b, c\}$, and the object "2" the set $\{d\}$.

$$R: \begin{array}{|cc|}\hline a & b \\ a & c \\ b & b \\ b & c \\ c & d \\ d & d \\ \hline \end{array} \qquad S: \begin{array}{|ccc|}\hline a & b & 1 \\ a & c & 2 \\ b & c & 3 \\ c & d & 4 \\ \hline \end{array} \qquad T: \begin{array}{|cc|}\hline a & 1 \\ b & 1 \\ c & 2 \\ d & 2 \\ \hline \end{array}$$

Fig. 1. Possible results of the new-statement $S :=$ **new** $\{(x, y) \mid R(x, y) \wedge x \neq y\}$ and the abstraction-statement $T :=$ **abstraction**$\{(x, y) \mid R(x, y)\}$.

Finally, we bring in iteration. A *while-statement* is any expression of the form **while** φ **do** P **od** with φ an FO-sentence and P a CQL program. The semantics of a while-statement is obvious.

Given input and output schemes $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$, a CQL *program* in which the relation names in the left-hand sides of FO-, new-, and abstraction-statements are *not* contained in $\mathcal{S}_{\text{in}}$ and resulting in relations over a superscheme of $\mathcal{S}_{\text{out}}$[5] can be interpreted as a *determinate query* from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$.

The language CQL, however, is *not* complete with respect to the determinate queries. This can already be seen at the level of individual input-output pairs of instances [1, 5]. Let Q be a determinate query, and suppose $Q(I, J)$. The only

[5] Relation names used only for intermediate computations may be ignored.

correspondence between I and J that can be derived from Definition 1 is that each automorphism in $Aut(I)$ can be extended to an automorphism in $Aut(J)$. A stronger correspondence exists if Q can be computed by a CQL program, however, because, in a CQL program, new objects are always created *in terms of existing objects*. Hence, if Q can be computed by a CQL program, there exists a *natural* extension mapping from $Aut(I)$ to $Aut(J)$. As this natural extension mapping preserves composition, it is a *group homomorphism* from $Aut(I)$ to $Aut(J)$. This observation led Andries and the present authors to the following definition and theorem in [18]:

Definition 2. Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes. A *constructive query* Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a determinate query satisfying

4. if $Q(I, J)$, then there exists an extension homomorphism from $Aut(I)$ to $Aut(J)$.

Theorem 3. *The language* CQL *is complete with respect to the constructive queries.*

3 Semi-deterministic queries and query languages

As mentioned in the Introduction, the constructive queries were proposed in [18] to restrict the determinate queries to a more natural class. Another natural class can be obtained by *extending* the class of determinate queries to the class of *semi-deterministic queries* [17]:

Definition 4. Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes. A *semi-deterministic query* Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a recursively enumerable binary relationship $Q \subset inst(\mathcal{S}_{\text{in}}) \times inst(\mathcal{S}_{\text{out}})$ satisfying items 1 and 2 of Definition 1 and

$3'$. if $Q(I, J_1)$ and $Q(I, J_2)$, then J_1 and J_2 are isomorphic.

Notice that items 1 and $3'$ imply that if φ is an isomorphism between J_1 and J_2, then φ restricted to $dom(I)$ is in $Aut(I)$.

The naturalness of semi-determinism is confirmed by the following alternative characterization of constructive queries, which immediately follows from a result in [17]:

Theorem 5. *Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes. A constructive query Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a* semi-deterministic *query satisfying item 4 of Definition 2.*

In order to express semi-deterministic queries, a choice mechanism is required. For this purpose, we consider the **swap-choice** operation first suggested in [17].

Let $\mathcal{S}$ be a scheme, Φ an FO-expression defined over $\mathcal{S}$, and I an instance over $\mathcal{S}$. Two objects a and b in $dom(\Phi(I))$ are called *swap-equivalent* if the

transposition $(a\ b)$ is in $Aut(I)$.[6] The operation **swap-choice** $\Phi(I)$ yields a unary relation obtained by non-deterministically selecting one representative for each equivalence class of swap-equivalent objects in $dom(\Phi(I))$. Finally a *swap-choice statement* has the form $R :=$ **swap-choice** Φ, with R a relation name and Φ as above. Its semantics is obvious.

Example 2. Consider Figure 2. Let I be the instance over the scheme $\{R\}$. The equivalence classes of swap-equivalent objects are $\{a, b\}$, $\{c, d\}$, $\{e, f\}$, $\{g\}$, and $\{h\}$. Hence the instance over the scheme $\{R, S\}$ is a possible result of the statement $S :=$ **swap-choice** $\{(x, y) \mid R(x, y) \wedge x \neq y\}$ applied to I.

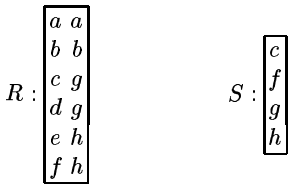

Fig. 2. A possible result of $S :=$ **swap-choice** $\{(x, y) \mid R(x, y) \wedge x \neq y\}$.

As swap-equivalence is first-order definable and hence in polynomial time, **swap-choice** is an efficient operation. Furthermore, **swap-choice** is guaranteed to act semi-deterministically. It is the authors' belief that there are no reasonable alternatives for this operation. Of course, non-symmetry-based choice mechanisms are not sound since they can compute non-semi-deterministic queries. Even symmetry-based equivalence relationships do not necessarily give rise to strictly semi-deterministic operators. In this respect, it might come as a surprise that the relationship under which a and b are equivalent if there is an automorphism in $Aut(I)$ mapping a to b does *not* give rise to a strictly semi-deterministic operator. Moreover, equivalences based on general symmetries are hard to compute. Hence, the symmetries under consideration need to be restricted. Clearly, the transpositions on which swap-equivalence is based are the most primitive symmetries that can be considered. Finally, it is not obvious how to allow more symmetries than just transpositions without sacrificing transitivity.

In the following sections, we characterize the expressiveness of the language CQL + **swap-choice** obtained by augmenting CQL with swap-choice-statements and investigate the efficiency gains resulting from this augmentation.

4 Expressiveness of swap-choice

Since CQL + **swap-choice** is an extension of the constructive queries (Definition 2), and since the constructive queries can be elegantly defined as a restriction

[6] The transitivity of swap-*equivalence* follows from the equality $(a\ c) = (b\ c)(a\ b)(b\ c)$.

of the semi-deterministic queries (Theorem 5), it is reasonable to search for a characterization by generalizing the condition in item 4 of Definition 2.

Thereto, suppose J is a possible result of a sequence of subsequent swap-choice-statements applied to an instance I. While in general an extension mapping from $Aut(I)$ to $Aut(J)$ will no longer exist [17], it is still possible to find a natural homomorphism from $Aut(I)$ to $Aut(J)$. To see this, one has to observe two facts: *(i)* an automorphism of I respects the swap-equivalence classes of $dom(I)$ with respect to I, and *(ii)* a sequence of subsequent swap-choice-statements induces a partial order on each of these swap-equivalence classes defined by the order in which objects were selected. Now consider total orders on each of these swap-equivalence classes compatible with the partial orders. Then the mapping sending an automorphism of I to the permutation of $dom(J) = dom(I)$ that has the same global effect on the swap-equivalence classes of $dom(I)$ with respect to I but respects each of the total orders thereon is a homomorphism from $Aut(I)$ to $Aut(J)$ with kernel $Swap(I)$, the group generated by the transpositions in $Aut(I)$. Definition 6 generalizes the relevant properties of this homomorphism:

Definition 6. Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes. A *swap-generic query* Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a semi-deterministic query satisfying

$4'$. if $Q(I, J)$, then there exists a homomorphism h from $Aut(I)$ to $Aut(J)$ with kernel $Ker(h) = Swap(I)$ such that, for all $\varphi \in Aut(I)$ and for all swap-equivalence classes S of $dom(I)$ with respect to I, *(i)* $h(\varphi)(S) = \varphi(S)$; and *(ii)* $\varphi(S) = S$ implies $h(\varphi)|_S = \text{Id}_S$.

It can be shown (proof omitted) that swap-genericity is closed under composition. We now establish the following characterization:

Theorem 7. *A query is swap-generic if and only if it can be expressed by a* CQL + **swap-choice** *program.*

Proof. (Sketch.) By the argument in the beginning of this section, a swap-choice statement is swap-generic; a variation of that argument can be used to show that each CQL statement is swap-generic, whence the "if."

To see the "only if," let Q be a swap-generic query from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ and let R be a binary relation name. The query Q_1 from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{in}} \cup \{R\}$ computing under R an arbitrary total order on each of the swap-equivalence classes of $dom(I)$ with respect to I can easily be expressed by a CQL + **swap-choice** program, say P_1. Furthermore it can be shown (details omitted) that there exists a *constructive* query Q_2 from $\mathcal{S}_{\text{in}} \cup \{R\}$ to $\mathcal{S}_{\text{out}} \cup \{R\}$ satisfying the following property: $Q(I, J)$ if and only if there exist I' and J' such that $Q_2(I', J')$, $I'|_{\mathcal{S}_{\text{in}}} = I$, $J'|_{\mathcal{S}_{\text{out}}} = J$, and $J'^R = I'^R$ defines a total order on each of the swap-equivalence classes of $dom(I)$ with respect to I. By Theorem 3, there is a CQL program P_2 expressing Q_2. Since Q can be obtained from $Q_2 \circ Q_1$ simply by omitting R from the output scheme, it is expressed by the CQL + **swap-choice** program $P_1; P_2$.

The proof of Theorem 7 furthermore yields a normal form for CQL + **swap-choice** programs: each swap-generic query can be expressed by a program in which all the swap-choice-statements precede all the object-creating statements.

It is enlightening to rephrase Definition 6 in more group-theoretic terms. Thereto, notice that the swap-equivalence classes of $dom(I)$ with respect to I are precisely the orbits S of $dom(I)$ with respect to $Swap(I)$. Generalizing this observation, one can define the following:

Definition 8. Let $\mathcal{S}_{\text{in}} \subseteq \mathcal{S}_{\text{out}}$ be two database schemes, and $N(I)$ a normal subgroup of $Aut(I)$ for each instance I over $\mathcal{S}_{\text{in}}$. An *N-generic query* Q from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a semi-deterministic query satisfying

4''. if $Q(I,J)$, then there exists a homomorphism h from $Aut(I)$ to $Aut(J)$ with $Ker(h) = N(I)$ such that, for all $\varphi \in Aut(I)$ and for all orbits S of $dom(I)$ with respect to $N(I)$, *(i)* $h(\varphi)(S) = \varphi(S)$; and *(ii)* $\varphi(S) = S$ implies $h(\varphi)|_S = \mathrm{Id}_S$.

Definition 8 provides a unifying framework to compare several notions of query. Indeed, for $N(I) = \{\mathrm{Id}_{dom(I)}\}$, the definition reduces to constructivity, for $N(I) = Swap(I)$ to swap-genericity, and for $N(I) = Aut(I)$ to general semi-determinism. As "on average" $Swap(I)$ is only a small subgroup of $Aut(I)$, the above comparison suggests that swap-genericity is very close to constructivity. This feeling is further confirmed by the following result:

Theorem 9. *A* CQL + **swap-choice** *program in which swap-choice is only applied to created objects*[7] *expresses a constructive query.*

As a consequence, a CQL + **swap-choice** program in which swap-choice is only applied to created objects can be simulated by a pure CQL program. Theorem 9 is shown by establishing the existence of an extension homomorphism between the automorphism groups of each input-output pair. In the following section, we give an alternative, constructive proof.

5 Swap-choice and counting

Often a pure CQL program simulating a program in CQL + **swap-choice** in which **swap-choice** is only applied to created objects (Theorem 9) is much less efficient than the original one. To see this, let R be a binary relation name, and consider the query Q from $\{R\}$ to $\{R,T,F,S\}$ returning the unary relations $T = \{(o_T)\}$, $F = \{(o_F)\}$ with o_T and o_F new objects representing *true* and *false*, respectively, and the binary relation S defined by $S(x,o_T)$ if $\{y \mid R(x,y)\}$ has even cardinality and $S(x,o_F)$ if $\{y \mid R(x,y)\}$ has odd cardinality. The query Q is expressed by the following CQL + **swap-choice** program:

[7] This means that, for every application of a statement $R :=$ **swap-choice** Φ to an intermediate instance K, $dom(\Phi(K)) \cap dom(I) = \emptyset$.

$R' :=$ **new** $\{(x,y) \mid R(x,y)\}$; $U := \{(x,z) \mid (\exists y)R'(x,y,z)\}$; $R' = \emptyset$;
$T :=$ **new** $\{()\}$; $F :=$ **new** $\{()\}$;
$S := \{(x,w) \mid (\exists y)R(x,y) \wedge T(w)\}$; $V := \emptyset$; $U' := \{(z) \mid (\exists x)U(x,z)\}$;
while $V \neq U'$ **do**
  $V' :=$ **swap-choice** $\{(z) \mid U'(z) \wedge \neg V(z)\}$; $V := \{(z) \mid V(z) \vee V'(z)\}$;
  $S := \{(x,w) \mid (\exists z)(V'(z) \wedge U(x,z)) \wedge (T(w) \vee F(w)) \wedge \neg S(x,w) \vee$
    $\neg(\exists z)(V'(z) \wedge U(x,z)) \wedge S(x,w)\}$
od

First, a unique new object is created for each tuple in R which is then disassociated from the objects in the second projection of R. Hence, the binary relation U associates to each object x in the first projection of R a set of swap-equivalent new objects with the same cardinality as $\{y \mid R(x,y)\}$. Next, S is initialized by associating *true* to each object x in the first projection of R. Finally, the loop computes the correct value for S by selecting one by one the new objects associated to x while flipping its boolean flag. The obvious implementation of the above program runs in low polynomial time. In pure CQL, however, the query Q cannot even be computed in polynomial space [4].

The above query Q is but one example of a large class of polynomial-time *counting queries*. This class is a two-sorted extension of fixpoint logic[8] with counting terms of the form $count\{\bar{y} \mid \Phi(\bar{x},\bar{y})\}$, which can be used as arguments of numerical functions computable in polynomial-time assuming unary notation for numbers, and was studied by Grädel and Otto [9] and Grumbach and Tollu [10].

In the context of object creation, we can alternatively define the counting queries without having to introduce a separate sort for the natural numbers, as a natural number in unary notation is an ordered list of objects. Using this representation, polynomial-time functions on natural numbers can be simulated faithfully by straightforward adaptation of known techniques showing that fixpoint logic equals polynomial time on ordered databases. Hence, we can get at the counting queries simply by augmenting CQL with an operation creating for each $\bar{x}$ a list of new objects of length $count\{\bar{y} \mid \Phi(\bar{x},\bar{y})\} + 1$.

Formally, let $\mathcal{S}$ be a scheme, and Φ a binary FO-expression defined over $\mathcal{S}$. Let I be an instance over $\mathcal{S}$, and $\{x_1,\ldots,x_p\} = \{x \mid (\exists y)(x,y) \in \Phi(I)\}$. For each $i = 1,\ldots,p$, let n_i be the cardinality of $\{y \mid (x_i,y) \in \Phi(I)\}$. The operation **count** $\Phi(I)$ non-deterministically selects, for each i, new objects $o_i^1,\ldots,o_i^{n_i+1}$, and yields the ternary relation

$$\{(x_1,o_1^1,o_1^2),(x_1,o_1^2,o_1^3),\ldots,(x_1,o_1^{n_1},o_1^{n_1+1}),(x_2,o_2^1,o_2^2),\ldots,(x_p,o_p^{n_p},o_p^{n_p+1})\}$$

in which each x_i is associated with a list of new objects of length $n_i + 1$.

The **count** operation can be simulated in pure CQL, but not efficiently. To see the latter, it suffices to observe that the following CQL + **count** analogue of the above CQL + **swap-choice** program also expresses Q efficiently:

[8] In our context, fixpoint logic can be thought of as the CQL programs using only FO-statements and inflationary while-loops.

$R' :=$ **count** $\{(x,y) \mid R(x,y)\}$; $U := \{(x,z) \mid (\exists z')R'(x,z,z')\}$; $R' = \emptyset$;
$T :=$ **new** $\{()\}$; $F :=$ **new** $\{()\}$;
$S := \{(x,w) \mid (\exists y)R(x,y) \wedge T(w)\}$; $V := \emptyset$; $U' := \{(z) \mid (\exists x)U(x,z)\}$;
while $V \neq U'$ **do**
  $V' := \{(z) \mid U'(z) \wedge \neg V(z) \wedge (\forall x)(\forall z')(R(x,z',z) \Rightarrow V(z')\}$;
  $V := \{(z) \mid V(z) \vee V'(z)\}$;
  $S := \{(x,w) \mid (\exists z)(V'(z) \wedge U(x,z)) \wedge (T(w) \vee F(w)) \wedge \neg S(x,w) \vee$
    $\neg(\exists z)(V'(z) \wedge U(x,z)) \wedge S(x,w)$ $\}$
od

The two programs exhibited in the preceding discussion are very similar in computational organization. The following result generalizes this observation:

Theorem 10. CQL + **swap-choice** *applied to new objects only and* CQL + **count** *are polynomial-time equivalent.*

Proof. (Sketch.) In the full version of [17], it was shown that the **count** operation is efficiently expressible by a semi-deterministic program in CQL + **witness**. Closer inspection of the proof reveals that the choices in this program can actually by expressed by applications of **swap-choice** to new objects only.

Conversely, an application of **swap-choice** to new objects only can be efficiently simulated in CQL + **count** as follows. First, the swap-equivalence relationship is materialized in a binary relation. This can be done efficiently since swap-equivalence is first-order definable. The **abstraction** operation when applied to the constructed binary relation will associate a unique new object to each swap-equivalence class. By applying **count**, a list of length one more than the cardinality of the corresponding swap-equivalence class can be attached to each such new object. By a simple CQL program, the objects in each swap-equivalence class are replaced by the non-tail objects in the corresponding list. The simulation of the **swap-choice** operation is now completed by selecting the first object in each of these lists.

As count-statements can be (unefficiently) expressed in pure CQL, the above argument also provides an alternative, constructive proof of Theorem 9. Furthermore, it should be noted that the proof of Theorem 10 heavily relies on the use of the **abstraction** operation. In [16] it was shown that the use of **abstraction** is necessary for the faithful representation of sets in CQL. We conjecture that **abstraction** is equally crucial for the simulation of **count**.

Most importantly, Theorem 10 shows that, in the absence of order, a very limited form of non-determinism is necessary and sufficient for efficient computation of the polynomial-time counting queries, and this without having to resort to the explicit introduction of the natural numbers as a special sort of domain values. The characterization in this paper of the nature of this non-determinism thus situates the polynomial-time counting queries within the class of all polynomial-time queries, for which unrestricted non-determinism is required, putting the results of Cai, Fürer, and Immerman [6] in perspective.

References

1. S. Abiteboul. Personal communications, 1990.
2. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. 1989 ACM SIGMOD Int. Conf. Management of Data*, in *SIGMOD Rec.*, 18(2):159–173, 1989.
3. S. Abiteboul and V. Vianu. Non-determinism in logic-based languages. *Ann. Math. Artif. Intell.*, 3:151–186, 1991.
4. S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. 23rd ACM Symp. Theory of Computing*, 209–219. 1991.
5. M. Andries and J. Paredaens. A language for generic graph-transformations. In *Graph-Theoretic Concepts in Computer Science, Proc. Int. Workshop WG 91*, *LNCS* 570, 63–74. Springer-Verlag, 1992.
6. J. Cai, M. Furer, and N. Immerman. An optimal lower bound on the number of variables for graph identification. In *Proc. 30th IEEE Symp. Foundations of Computer Science*, 612–617, 1989.
7. A. Chandra and D. Harel. Computable queries for relational database systems. *J. Comput. Syst. Sci.*, 21(2):156–178, 1980.
8. K. Denninghoff and V. Vianu. Database method schemas and object creation. In *Proc. 12th ACM Symp. Principles of Database Systems*, 265–275. 1993.
9. E. Grädel and M. Otto. Inductive definability with counting on finite structures. In *Proc. 6th Workshop on Computer Science Logic*, *LNCS* 702, 231–247. Springer-Verlag, 1993.
10. S. Grumbach and C. Tollu. Query languages with counters. In *Proc. 4th Int. Conf. Database Theory*, *LNCS* 646, 124–139. Springer-Verlag, 1992.
11. M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *Proc. 9th ACM Symp. Principles of Database Systems*, 417–424. 1990.
12. M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model for database end-user interfaces. In *Proc. 1990 ACM SIGMOD Int. Conf. Management of Data*, in *SIGMOD Rec.*, 19(2):24–33. 1990.
13. R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. 16th Int. Conf. on Very Large Data Bases*, 455–468. 1990.
14. N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.
15. M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proc. 8th ACM Symp. Principles of Database Systems*, 379–393. 1989.
16. J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proc. 10th ACM Symp. Principles of Database Systems*, 291–299. 1991.
17. J. Van den Bussche and D. Van Gucht. Semi-determinism. In *Proc. 11th ACM Symp. Principles of Database Systems*, 191–201. 1992. Full version submitted.
18. J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, 372–379. 1992.
19. M. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. Theory of Computing*, 137–146. 1982.

This article was processed using the LaTeX macro package with LLNCS style