Non-Deterministic Aspects of Object-Creating Database Transformations

Jan Van den Bussche*
University of Antwerp
Antwerp, Belgium

Dirk Van Gucht†
Indiana University
Bloomington, IN, USA

Abstract

Various non-deterministic aspects of object creation in database transformations are discussed, from a modeling as well as from an expressive power point of view.

1 Introduction

In the past few years, a lot of attention has been paid to *database transformations* [AK89, AV88, AV90, AV91a, GPVG90, Hul86, HWWY91, HY91]. Database transformations are binary accessibility relationships between database instances, and provide a simple, unifying model for the study of database query languages and important notions in database dynamics, such as updates, updatable views, and database restructurings.

In that study, the issue of *object creation* has attracted particular interest recently. After an object-creating database transformation (as opposed to a *domain-preserving* one), the output can contain domain elements that were not present in the input. Object creation differs from conventional insertion of a new value in that it contains an implicit form of *non-determinism*, i.e., the particular choice of the new domain value to be added is irrelevant, as long as it is new. This weak form of non-determinism is called *determinacy* [AK89].

Object creation finds an obvious application in object-oriented database systems, where, if a new object has to be added, the identity of this object must be distinct from those of all objects already present, but the particular choice of this new identifier is an implementation detail. Similarly, object creation is sometimes unavoidable to fully support updatable views in object-oriented database systems, i.e., derived information that is to be explicitly added in the database [AB91, HS91a], as well as database restructuring [AK89, GPVG90].

The natural links between object-creating and non-deterministic database transformations [AV91b] were recently explored by the authors [VdBVG92], for the special case of queries and views (modeled as type-increasing transformations.) In the present paper, we widen our scope to arbitrary updates, present some new results and indicate some new open problems.

*Research Assistant of the N.F.W.O. Address: University of Antwerp (UIA), Dept. Math. & Comp. Science, Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: vdbuss@uia.ac.be.

†Indiana University, Comp. Science Dept., Bloomington, IN 47405-4101, USA. E-mail: vgucht@cs.indiana.edu.

In Section 2, we define a simplified version of the IQL* model [AK89] for reasoning about object-creating database transformations. We discuss determinacy, originally explored for the special case of queries, and show that some care must be taken when moving to arbitrary updates. We also compare the *determinate approach* to object-creation of [AK89] as used in this paper, to the *functional approach* proposed in [KW89]. There, newly created objects are interpreted as the result of functions applied to the existing objects. These functions are usually expressed as Skolem function terms, as is used in Logic Programming. We will show that the two approaches can be reconciled; in both approaches, object creation can be interpreted as untyped tuple and set construction. This reconciliation will also clarify the issue of the *completeness up to copies* of IQL* w.r.t. determinate transformations [AK89].

In Section 3, we discuss *semi-determinism*, a natural generalization of determinacy. We present a new result, linking object creation and non-determinism. More concretely, we show that object creation in function of sets is in a precise sense equivalent to a special semi-deterministic choice operation. Semi-determinism was again originally explored only for queries [VdBVG92]. Extending the concept to arbitrary updates will turn out to yield several problems. First, it is not clear how the composition of two semi-deterministic transformations should be defined. Second, it is not clear what a language complete for all semi-deterministic transformations should look like. This contrasts sharply with the situation in the special case of queries, as considered in [VdBVG92], where these two problems can be solved quite naturally. Perhaps we have encountered a fundamental distinction between queries and updates?

2 Determinate object-creating database transformations

For simplicity, we shall work in the relational model. Generalizations to richer complex object, semantic, or object-based models are straightforward.

Let $\mathcal{S}_{\text{in}}, \mathcal{S}_{\text{out}}$ be database schemes. A *database transformation* of type $\mathcal{S}_{\text{in}} \to \mathcal{S}_{\text{out}}$ is a binary relationship $T \subseteq \mathit{inst}(\mathcal{S}_{\text{in}}) \times \mathit{inst}(\mathcal{S}_{\text{out}})$,[1] such that for some finite set C of constants, T is *C-generic*, meaning that for each C-permutation[2] f of the domain, $I \xrightarrow{T} J$ implies $f(I) \xrightarrow{T} f(J)$. (Transformations that are actually functions are called *deterministic*.)

Genericity is a consistency criterion, well-established in the database field [AU79]. Intuitively, it says that data elements should be treated uniformly (apart from a finite set of constants C that are particular to the application at hand). Pragmatically, it says that we are considering database computations, not arbitrary computations on bit streams.

We say that transformation T above is *object creating* if there are instances $I \xrightarrow{T} J$ for which the active domain of the output, J, contains elements that were not in the input, I, nor in C.

For example, if P is a relation containing person tuples, then the insertion:

$$P \xleftarrow{+} [\text{name} : \text{john}, \text{age} : 40, \text{address} : \text{new york}] \qquad (1)$$

[1] For a scheme $\mathcal{S}$, $\mathit{inst}(\mathcal{S})$ denotes the set of database instances over $\mathcal{S}$.

[2] A C-permutation fixes every element of C.

name	age	address
jeff	20	dallas
john	40	new york

id	name	age	address
α	jeff	20	dallas
β	john	40	new york

id	name	age	address
α	jeff	20	dallas
β	john	40	new york
γ	john	40	new york

Figure 1: Object creation.

is not object-creating, but simply C-generic, with $C = \{\text{john}, 40, \text{new york}\}$. On the other hand, suppose PO is a class containing person objects, and we want to insert a new object with a new id. The update:

$$PO \overset{+}{\Leftarrow} \textbf{new}[\text{id}]\ [\text{name} : \text{john}, \text{age} : 40, \text{address} : \text{new york}] \tag{2}$$

where **new**[id] stands for an arbitrary new id not yet present in the input database, is object-creating (and C-generic.)

We next define a simple language for expressing database transformations. Programs in the language are built from basic statements. Basic statements can be divided into relational statements and object-creating statements. Finally, programs can be iterated or called recursively. In this paper, we will not need to be concerned with the particular form of the iteration construct.

The *relational* statements are of the forms

$$P \overset{+}{\Leftarrow} E,\ P \overset{-}{\Leftarrow} E, \quad \text{or } \textbf{drop}\ P$$

where P is a relation name and E is an expression in the relational algebra or calculus. The **drop** statement allows the destruction of an old relation with name P. For the $+$ and $-$ statements, the semantics is that E is evaluated, and the result is added to (in case of $+$) or deleted from (in case of $-$) relation P. In the case of $+$, P may be a new relation name, in which case a new relation with name P is created and initialized to the value of E. For example, the simple insertion statement (1) shown in the earlier example is a trivial example of a relational statement (the relational calculus expression is constant in this example.)

As just defined, relational statements are not object-creating. In order to express object creation, simple variations of these statements can be used. The **new** statement:

$$P \overset{+}{\Leftarrow} \textbf{new}[A]E$$

creates a new object for each tuple in the value of E. More precisely, the value of the expression **new**$[A]E$ is the relation obtained from the value of the expression E, after tagging each tuple with a different, new, A-attribute value.

To illustrate these concepts, consider the following example. Starting from the person relation P shown in Figure 1, left, we can create the set PO of person objects with identity shown in Figure 1, right, with the statement

$$PO \overset{+}{\Leftarrow} \textbf{new}[\text{id}]\ P. \tag{3}$$

R_1	R_2	
a	c	c
b	d	d

R_1	R_2	
a	c	a
b	d	b

R_1	R_2	
a	e	a
b	f	b

Figure 2: Composition does not preserve determinacy.

Note that this set PO contains a person object with attributes [name:john, age:40, address:new york]. If we now apply the earlier object insertion update statement (2), another person with these attributes will be added to PO, with another identity, as shown in Figure 1, middle. If we want to add a new person only if a person with the same attributes is not already present in the database, we must use a more elaborate statement:

$$\mathrm{PO} \stackrel{+}{\Longleftarrow} \mathbf{new}[\mathrm{id}]\ [\text{name:john,age:40,address:new york}] \wedge (\nexists t \in PO) : t(\text{name}) = \text{john} \wedge t(\text{age}) = 40 \wedge t(\text{address}) = \text{new york}$$

We point out that the **new** operation is but a common denominator for many mechanisms for object creation considered in the literature [AV91a, AK89, GPVG90, HS91b, HY90, KW89, Zan89].

An immediate consequence of genericity is that an object creating transformation must be non-deterministic. For example, in Figure 1, the particular choice of α and β as new id's is not crucial: we could have taken ε and δ as well. However, we can leave it to that, and restrict the non-determinism only to the particular choice of the newly created objects. This results in the notion of *determinate* transformation [AK89].

A transformation T is determinate if whenever $I \stackrel{T}{\longrightarrow} J_1$ and $I \stackrel{T}{\longrightarrow} J_2$, there exists an isomorphism $f : J_1 \to J_2$ that is the identity on the objects in I.

So, determinacy is determinism up to the choice of the new objects in the output.

Determinacy was originally considered in the context of queries (and views). These can be modeled as transformations that are *type-increasing*. A transformation T of type $\mathcal{S}_{\text{in}} \to \mathcal{S}_{\text{out}}$ is type-increasing if $\mathcal{S}_{\text{in}} \subset \mathcal{S}_{\text{out}}$, and for each $I \stackrel{T}{\longrightarrow} J$ and each relation name $R \in \mathcal{S}_{\text{in}}$, $R^J = R^I$.

For type-increasing transformations, determinacy is a robust notion: determinacy is preserved under composition. For arbitrary updates, however, this robustness property breaks down. For example, let $\mathcal{S}_{\text{in}}$ consist of two relations R_1, R_2. Consider transformation T_1, which erases relation R_2, and transformation T_2, which adds to relation R_2 as many new objects as there are in R_1. So, $T_1 \equiv R_2 \stackrel{-}{\Longleftarrow} R_2$ and $T_2 \equiv R_2 \stackrel{+}{\Longleftarrow} \mathbf{new} R_1$. Then Figure 2, middle and right, shows two possible results of the composition $T_1; T_2$ applied to the instance shown in Figure 2, left. The determinacy requirement is clearly not satisfied, the reason being that in the middle instance, objects are created having belonged to R_2 in the past, while in the right instance, brand new objects are created.

This difficulty can be remedied by disallowing such situations: once an object has "died" (has been deleted), it can never be "reborn" (created). This is a commonly heard requirement in discussions on object identity, but the above discussion provides, to our knowledge, the first formal argument of why this is

indeed necessary. Thus, rather than defining composition $T_1; T_2$ classically as: $\{(I,K) \mid (\exists J) T_1(I,J) \wedge T_2(J,K)\}$, we formally define it as: $\{(I,K) \mid (\exists J) T_1(I,J) \wedge T_2(J,K) \wedge$ for each object o appearing in I and K, o also appears in $J\}$. Using this new definition of composition, we can now show:

Theorem: If T_1, T_2 are determinate database transformations, then so is $T_1; T_2$.

Proof: We know that T_1, T_2 are C-generic for some C. To check whether $T_1; T_2$ is C-generic as well, assume $T_1; T_2(I,J)$, and let f be a C-permutation of the universe. We show that also $T_1; T_2(f(I), f(J))$. There exists a K such that $Q_1(I,K)$, $Q_2(K,J)$, and every object in $I \cap J$ is also in K. We have $T_1(f(I), f(K))$ and $T_2(f(K), f(J))$. Every object in $f(I) \cap f(J)$ is also in $f(K)$. Therefore, $T_1; T_2(f(I), f(J))$.

To check determinacy, assume $T_1; T_2(I, J_1)$ and $T_1; T_2(I, J_2)$. Then K_r, for $r = 1, 2$, exists such that $T_1(I, K_r)$, $T_2(K_r, J_r)$, and every object in $I \cap J_r$ is also in K_r. Since T_1 is determinate, there exists an isomorphism $f_1 : K_1 \to K_2$ such that f_1 is the identity on $I \cap K_1$. Since T_2 is generic, $T_2(K_2, f_1(J_1))$. Since T_2 is determinate, there exists an isomorphism $f_2 : f_1(J_1) \to J_2$ such that f_2 is the identity on $K_2 \cap f(J_1)$. Clearly, $f := f_2 \circ f_1 : J_1 \to J_2$ is an isomorphism. It remains to show that f is the identity on $I \cap J_1$. So, let $o \in I \cap J_1$. Then also $o \in K_1$, and hence $f_1(o) \in K_2 \cap f(J_1)$ as well as $f_1(o) = o$. Therefore, $f_2(f_1(o)) = f_1(o) = o$. $\square$

The **new** operation just introduced creates new objects in function of the tuples in some relation. One also needs a mechanism that creates objects in function of sets. This is provided by the *abstraction* operation in the following way. Let r be a binary relation. Then r can be viewed as a set-valued function, in the standard way. The result of $\mathbf{abstr}[A](r)$ is a ternary relation, obtained from r after tagging each tuple with a new A-attribute, such that two tuples (x_1, y_1) and (x_2, y_2) get the same tag if and only if the sets $r(x_1)$ and $r(x_2)$ are equal. This as opposed to the **new** operation, where *every* tuple gets a different tag.

The abstraction operation is determinate. We can incorporate abstraction in our language by allowing statements of the form: $P \stackrel{+}{\Leftarrow} \mathbf{abstr}[A]E$. For example, if PC is a parent-child relation, then the transformation:

$SPC \stackrel{+}{\Leftarrow} \mathbf{abstr}[S]PC;$
$SC \stackrel{+}{\Leftarrow} \pi_{S,C}(SPC)$

creates a unique object for each set of siblings and represents these sets in relation SC. See Figure 3. If we now also apply the statements

$PS \stackrel{+}{\Leftarrow} \pi_{P,S}(SPC);$
drop $PC;$
drop SPC

then the new relations PS and SC correspond to a useful *restructuring* of the original PC relation. See Figure 3.

Abstraction is the operational equivalent of the ability to treat set values as first-class citizens. As defined here, it was originally introduced in [GPVG90, VdBP91], but it is a mechanism that has been considered frequently in the

P	C
sam	toto
sam	zaza
mary	toto
mary	zaza
fred	nini

A	P	C
α	sam	toto
α	sam	zaza
α	mary	toto
α	mary	zaza
β	fred	nini

S	C
α	toto
α	zaza
β	nini

P	S
sam	α
mary	α
fred	β

Figure 3: Abstraction.

literature. Abstraction corresponds to such concepts as grouping over sets [BNR$^+$87], nesting [TF86], set-valued functions [AG91], and set-valued object assignments [AK89].

The transformation language with relational statements (giving us first-order logic), **new** (giving us tuple-id creation), **abstr** (giving us set-id creation) and recursion is essentially equivalent to the IQL* language, proposed in the seminal paper [AK89]. Therefore, we take the liberty to refer to it with the same name IQL*.

The semantics for object-creation considered above, has not been the only approach adopted in the literature. An alternative approach is the one proposed in [KW89], where object creation is interpreted as the construction of function terms over the existing objects, as is done in Logic Programming. We will call this the *functional* approach. The functional approach typically uses a calculus framework, where the relational calculus is extended with first-order function symbols. There is no need then for a special **new** operation.

For example, recall the **new** statement (3). The analogue in the functional approach would be:

$$PO \stackrel{+}{\Longleftarrow} \{[\text{id: } f(x,y,z), \text{name: } x, \text{age: } y, \text{address: } z] \mid P(x,y,z)\}$$

where f is a function symbol. Applying this statement to Figure 1 (left) will produce Figure 1 (right); however now, α and β are not arbitrary new values, but equal the first-order terms $f(\text{jeff}, 20, \text{dallas})$ and $f(\text{john}, 40, \text{new york})$, respectively. If g is another function symbol, applying the functional analogue of the insertion (1):

$$PO \stackrel{+}{\Longleftarrow} \{[\text{id: } g(\text{john}, 40, \text{new york}), \text{name: john}, \text{age: } 40, \text{address: new york}]\}$$

to Figure 1 (right) will produce Figure 1 (middle) with γ equal to $g(\text{john}, 40, \text{new york})$.

Note that if we would have used the same f, then no new object would have been created. This technical difference between the determinate approach and the functional approach is not really important, as we will see momentarily. In fact, there are no fundamental differences at all between the two approaches: we will next sketch how the one can simulate the other.[3]

We start with simulating the determinate approach by the functional one. For each relation name R that is used in the program, we have a different function symbol f_R. Then the statement:

$$R \stackrel{+}{\Longleftarrow} \textbf{new}[A]E$$

[3] A similar observation was made in [HY90], without proof.

is simulated by:

$$R \stackrel{+}{\Leftarrow} \{[A : f_R(x_1, \ldots, x_n), A_1 : x_1, \ldots, A_n : x_n] \mid E(A_1 : x_1, \ldots, A_n : x_n)\}.$$

As just mentioned, this naive simulation fails in case object $f_R(x_1, \ldots, x_n)$ already exists in the database (because it was created by a previous **new** statement). This problem can be circumvented by storing in an auxiliary table Created_R for each tuple $x_1, \ldots, x_n$ the last object o created in function of that tuple. The above statement is then modified to test first if there was already an object creation in function of $x_1, \ldots, x_n$; if yes, then $f_R(o, x_2, \ldots, x_n)$ is used instead of $f_R(x_1, \ldots, x_n)$.[4]

To simulate the functional approach by the determinate one, it suffices to store for each function symbol f used in the program, an auxiliary table R_f storing the current extension of f. If $f_R(x_1, \ldots, x_n)$ is created, then, using the **new** operation, a tuple $(o, x_1, \ldots, x_n)$ is stored in R_f with o an arbitrary new domain element playing the role of $f_R(x_1, \ldots, x_n)$. Further references to $f_R(x_1, \ldots, x_n)$ can then be handled by lookup in the auxiliary table.

The object-creation functions considered above are "syntactical" functions; they form first-order terms from their arguments. This is the standard Logic Programming approach. An alternative formalism for the functional approach to object creation, based on "semantical" functions, was presented in [GSVG92], where the so-called *tagging functions* are defined. There, all object creation happens through a single tagging function. The syntactical functional approach is a special case of the tagging functional approach, where this tagging function is simply the identity, mapping its arguments $x_1, \ldots, x_n$ to the tuple $(x_1, \ldots, x_n)$. (See also [VG85].) That there is one single tagging function is not a real restriction; if multiple function symbols are needed, one takes for each such symbol f a different constant c_f and simulates $f(x_1, \ldots, x_n)$ by $(c_f, x_1, \ldots, x_n)$.

So, object creation through the **new** operation, or, equivalently, functional object creation, amounts to tuple construction. Note that this tuple construction is necessarily *untyped*, since the nested structure of the various tuples is not fixed. Generalizing this observation to include the **abstr** operation, which creates objects representing sets, not tuples, means moving from tuples to sets. Indeed, tuples are a special case of sets by the well-known axioms: $(x, y) = \{\{x\}, \{x, y\}\}$ and $(x_1, \ldots, x_n) = (x_1, (x_2, \ldots, x_n))$. Including set-object creation in the functional approach could be done by introducing set-valued functions (as in [AG91]) or set-terms (as in [Kup90]). We have thus arrived at the important insight that *object creation in IQL* can be interpreted as untyped-set construction.*

The use of untyped sets in database query languages has been studied in the literature [DM87, HS89]. It is known that the relational calculus or algebra, extended with untyped sets and recursion or iteration, is computationally complete for all computable generic untyped-sets queries. It thus follows that IQL* can express exactly those determinate queries that correspond to a generic untyped-sets query. This has been made precise in [VdBVGAG92], where the *constructive* queries were defined as a natural subclass of the determinate ones. Informally, a query is constructive if symmetries of the input database can be naturally extended to symmetries of the query result. It thus follows that

[4] This is only one trick to circumvent the problem; others can be used as well.

P	C
sam	toto
mary	zaza
fred	nini

P	C
sam	zaza
mary	zaza
fred	nini

Figure 4: Witness.

IQL* can express exactly the constructive queries. This clarifies the incompleteness of IQL* w.r.t. the determinate queries; indeed, IQL* is known to be only determinate-complete *up to copies*.[5] One possible philosophical interpretation of all this is that, a posteriori, the notion of determinacy is not quite restrictive enough. At any rate, there does not appear to be a *natural* example of a determinate, non-constructive query.

The foregoing discussion (and the presentation in [VdBVGAG92]) went on for the special case of queries. However, it can be shown that everything goes through in general. In particular, one can consider general *constructive database transformations*, and prove that IQL* expresses exactly all constructive database transformations.

To conclude this section, we point out that in the functional approach, once the object-creating function(s) have been fixed, object-creation becomes a *deterministic* operation. In other words, the non-deterministic characteristic of object creation is entirely captured in the particular *choice* of the function. As a consequence, there is no problem in using the classical composition of database transformations, as in the determinate case. Also, it is interesting to observe that the function-terms semantics was adopted in [AB91] to study the view-update problem in the context of object-oriented database systems.

3 Semi-determinism

IQL* is a determinate, hence nearly deterministic language. In order to express arbitrarily non-deterministic transformations one can add the *witness* operation to the language. (See [Abi88] for motivations for non-determinism, and [AV91b] for a survey.)

Witness is a non-deterministic choice operation. Let r be a relation, and let X be a subset of its attributes. Then $W_X(r)$ evaluates to a sub-instance of r, obtained by choosing exactly one representative from each class of X-equivalent tuples of r. Here, two tuples are X-equivalent if they agree outside X.

For example, if r is the PC relation of Figure 3, then two possible results of $W_C(r)$ are shown in Figure 4. For every parent, an arbitrary child is chosen (as a "witness" for his/her parenthood.)

Witness can be incorporated in IQL* by allowing statements of the form: $P \overset{+/-}{\Longleftarrow} W_X E$. The resulting language will be called NIQL* (for non-deterministic IQL*). It is easy to see that NIQL* is equivalent to the languages TL [AV90] and DL* [AV91a], which are known to express exactly all computable database transformations. So, NIQL* is well-understood, and also very powerful.

[5] In [AK89], it was conjectured that that IQL* is fully determinate-complete, but this was later shown to be false [Abi90, AP92].

It is interesting to consider restrictions on the unlimited non-determinism that NIQL* programs allow. (For motivations we refer to [VdBVG92].) To find sensible such restrictions, one can get inspiration from the definition of determinate transformation. That definition required that two possible outcomes of the transformations be isomorphic, *but* via an isomorphism that keeps the existing objects fixed. In [VdBVG92], the present authors considered a weakened version of this qualification, for the special case of queries. The adopted qualification was to require that two possible outcomes of the query be isomorphic, *but* via an isomorphism that is an *automorphism* (also called *symmetry*) of the input database. Such queries were called *semi-deterministic*.

Denote by SNIQL^q, the sublanguage of NIQL* consisting of the programs that express semi-deterministic queries. The class of semi-deterministic queries exhibits a number of good properties [VdBVG92]. It is therefore not surprising that membership in SNIQL^q is undecidable. Fortunately, we can replace the witness operation by a weaker choice construct, called *swap-choice*, which is guaranteed to be semi-deterministic (and still efficient to implement.) Swap-choice was already implicit in [VdBVG92]; we next define it explicitly.

Let I be a database instance and R a relation name in the scheme of I. The *swap-choice* $\mathbf{swap}[C](R, I)$ evaluates to a unary relation with attribute C, obtained by choosing from each equivalence class of *swap-equivalent* elements of the domain of R^I exactly one representative. Here, two domain elements o_1, o_2 are called swap-equivalent if the transposition $(o_1\ o_2)$ is a symmetry of I.

We can incorporate swap-choice in IQL* by allowing statements of the form: $P \overset{+/-}{\Longleftarrow} \mathbf{swap}E$. We will call the resulting language SIQL*, and its restriction to queries SIQL^q. It can be shown that queries expressed in SIQL^q are semi-deterministic.

While swap-choice may seem ad-hoc at first, the following theorem gives evidence for its naturalness. It says that in a semi-deterministic context, swap-choice, and the set-based object creation operation abstraction, are equivalent in expressive power. Since abstraction is naturally related to duplicate elimination in object creating languages, this result gives an intuitive technique to remove duplicates, i.e., retain only one version of otherwise indistinguishable objects. The theorem should also be contrasted with the result in [VdBP91] that abstraction is primitive in IQL*.

Theorem: *Swap-choice is expressible in SNIQL^q, using abstraction. Conversely, abstraction is not primitive in SIQL^q.*

Proof sketch: First, observe that swap-equivalence is definable in the relational calculus. Using abstraction, equivalence classes of swap-equivalent domain elements are grouped together and given a unique object identifier. These new identifiers are swap-equivalent, and using witness, representatives can be chosen in a semi-deterministic manner.

For the converse, observe that abstraction can be simulated using **new** with duplicates. These duplicates are swap-equivalent, yielding a simulation of abstraction using **new** and **swap**. □

The above theorem suggests that swap-choice, *in combination with object creation through* **new**, allows the specification of important data manipulation operations. This was also suggested in [VdBVG92], where it was shown how a subclass of the *counting* queries (i.e., queries that involve the cardinality

of certain sets) can be expressed in SIQL^q in PTIME. We have since then substantially extended the relevant techniques, and can prove that *all* counting queries can be expressed in this way. (We do not present this proof here.) The importance of this result stems from the observation that, unlike [ASV90], we rely only on the very weak form of non-determinism allowed by the swap-choice operation. It is indeed well-known that *unrestricted* non-determinism offers efficient expressibility of all PTIME transformations. In other words, as far as counting applications are concerned, the usage of non-determinism can be heavily constrained, as to appear almost deterministic, however without sacrificing efficiency. It would be interesting to find other applications where this is possible.

As argued in [VdBVG92] and further illustrate here, object-creation is vital in order to make semi-determinism feasible. This is especially true for queries, on which we have focused in our discussion up to now. We next turn to the question of how semi-determinism can be adapted to the general case of arbitrary database transformations. We will not give many answers, but rather indicate directions for further research.

We start with an example which, although trivial, demonstrates how the use of updates admits certain counting queries to be expressed semi-deterministically without a need for object creation. Let I be a database and consider a query based on the number of domain elements in a number of relations of I. This query can be solved semi-deterministically by first computing the active domain of these relations using a standard relational algebra expression. Subsequently every relation in I is dropped. We are now left with a unary relation containing the wanted active domain. We can now order this unary relation by repeatedly choosing elements from this unary relation. The choices are semi-deterministic, since we choose from an unstructed set of elements. Once the set is ordered, it is straightforward to test any PTIME-computable property of its cardinality.

Recall that for queries, which are modeled as type-increasing transformations, semi-determinism requires that two possible outcomes of the query to the same database I must be isomorphic, by an isomorphism that is a symmetry of I. But note that the latter condition on the isomorphism is actually voidly satisfied, since queries are type-increasing. When applying the same definition to arbitrary transformations, the condition becomes a real additional requirement. Formally, the requirement becomes: the isomorphism from J_1 to J_2 must map $I \cap J_1$ to $I \cap J_2$, where J_1 and J_2 are the two possible outcomes. For this definition to be workable, it needs to satisfy a reasonable compositionality criterion. At the end of Section 2 we gave such a criterion for determinate transformations. Since the semi-determinate transformations ought to include the determinate transformations, the searched for criterion needs to be at least as restrictive.

The following example puts a damper on the above proposal for semi-deterministic transformation. Let T_1 be the determinate transformation which adds to a unary relation S as many new elements as it originally contained:[6]

$$T_1 \equiv S \stackrel{+}{\Longleftarrow} \rho_{A \mapsto S} \pi_A \mathbf{new}[A](S).$$

Furthermore, let T_2 be the transformation which deletes an arbitrary element

[6] $\rho_{A \mapsto S}$ denotes a renaming.

from a unary relation S:

$$T_2 \equiv S \Leftarrow\!\!\!= \mathbf{swap}(S).$$

Obviously, T_1 and T_2 are semi-deterministic according to the above proposal. However, when $T_1; T_2$ (defined as in the determinate case) is applied to a singleton relation $S = \{(a)\}$, a possible result of $T_1(S)$ is $\{(a), (b)\}$, and therefore two possible results of $T_1; T_2(S)$ are $\{(a)\}$ itself and $\{(b)\}$, which obviously violates the proposed requirement (since the intersection of S with the second result is empty, while the intersection with the first one is not).

Another possible natural generalization of the notion of semi-determinism from queries to updates is to simply require that two possible results are plainly isomorphic. In this case, compositionality is not a problem. In fact, even the classical composition can be used then.

Many other candidate definitions for semi-deterministic database transformation can be considered. For any of them, apart from the compositionality problem, there is another issue that must be taken into account: *completeness*.

Indeed, for the special case of queries, it can be shown that there exists a decidable sublanguage of SNIQL^q that is semi-deterministic-complete. This can be shown using the strategy to prove that IQL* is determinate-complete *up to copies* [AK89]. In contrast to the determinate situation, these copies do not pose a problem in a semi-deterministic setting, since they are isomorphic and can be eliminated in a semi-deterministic manner.[7] One can furthermore prove that the determinate strategy produces "sufficiently many" possible results; this is in turn not an issue in the determinate setting.[8]

Clearly, it is desirable to achieve semi-deterministic completeness also in the general database transformation case. This is a challenging open problem.

We conclude with the remark that in principle, semi-determinism can also be considered in the functional approach to object creation. However, in contrast to the determinate case, where the functional approach was not essentially different from the determinate approach, it is our belief that with respect to semi-determinism, there is is a fundamental difference between the two approaches. In particular, none of the main theorems presented here and in [VdBVG92] seem to hold for the functional approach.

Acknowledgment

We are indebted to Marc Gyssens for inspiring discussions on the issues of compositionality of determinate database transformations and the completeness of SNIQL^q.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proceedings of the 1991 ACM SIGMOD*

[7] This description is a gross simplification.

[8] We have indeed been able to settle the open problem mentioned in [VdBVG92, page 198] affirmatively, contrary to the intuition expressed there.

International Conference on Management of Data, volume 20:2 of *SIGMOD Record*, pages 238–247. ACM Press, 1991.

[Abi88] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paredaens, and D. Van Gucht, editors, *ICDT'88*, volume 326 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1988.

[Abi90] S. Abiteboul. Personal communication, 1990.

[AG91] S. Abiteboul and S. Grumbach. A rule-based langauge with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 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*, volume 18:2 of *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*, volume 570 of *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 PODS [POD90], pages 218–229.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[AV88] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *Journal of the ACM*, 35:70–120, 1988.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 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. Non-determinism in logic-based languages. *Annals of Mathematics and Artificial Intelligence*, 3:151–186, 1991.

[BNR+87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 21–37. ACM Press, 1987.

[DM87] E. Dahlhaus and J.A. Makowsky. Computable directory queries. In *Logic and Computer Science: New Trends and Applications*, Rend. Sem. Mat. Univ. Pol. Torino, pages 165–197. 1987. An extended abstract appears in *LNCS 214*.

[GPVG90] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In PODS [POD90], pages 417–424.

[GSVG92] M. Gyssens, L.V. Saxton, and D. Van Gucht. Tagging as an alternative to object creation. In D. Maier and G. Vossen, editors, *Query Processing in Object-Oriented, Complex-Object and Nested Relation Databases*. Morgan Kaufmann, 1992. To appear.

[HS89] R. Hull and Y. Su. Untyped sets, invention, and computable queries. In PODS [POD89], pages 347–359.

[HS91a] A. Heuer and P. Sander. Classifying object-oriented query results in a class/type lattice. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, editors, *MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 14–28. Springer-Verlag, 1991.

[HS91b] A. Heuer and P. Sander. Preserving and generating objects in the Living In A Lattice rule language. In *Proceedings Seventh International Conference on Data Engineering*, pages 562–569. IEEE Computer Society Press, 1991.

[Hul86] R. Hull. Relative information capacity of simple relational schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[HWWY91] R. Hull, S. Widjojo, D. Wile, and M. Yoshikawa. On data restructuring and merging with object identity. *IEEE Data Engineering*, 14(2):18–22, June 1991.

[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 PODS [POD91], pages 328–340.

[Kup90] G. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.

[KW89] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In PODS [POD89], pages 379–393.

[POD89] *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*. ACM Press, 1989.

[POD90] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems.* ACM Press, 1990.

[POD91] *Proceedings of the Tenth ACM Symposium on Principles of Database Systems.* ACM Press, 1991.

[TF86] S. Thomas and P. Fischer. Nested relational structures. In P. Kanellakis, editor, *The Theory of Databases*, pages 269–307. JAI Press, 1986.

[VdBP91] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In PODS [POD91], pages 291–299.

[VdBVG92] 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.

[VdBVGAG92] 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.

[VG85] D. Van Gucht. *Theory of unnormalized relational structures.* PhD thesis, Vanderbilt University, 1985.

[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.