On the Completeness of Object-Creating Database Transformation Languages[1]

Jan Van den Bussche[2]
Dirk Van Gucht[3]
Marc Andries[4]
Marc Gyssens[5]

[1]A preliminary version of part of this paper was presented at the 33rd IEEE Symposium on Foundations of Computer Science (Pittsburgh, October 1992).

[2]University of Limburg (LUC), Department WNI, Universitaire Campus, B-3590 Diepenbeek, Belgium. E-mail: vdbuss@luc.ac.be.

[3]Indiana University, Computer Science Department, Bloomington, IN 47405-4101, USA. E-mail: vgucht@cs.indiana.edu.

[4]University of Antwerp (UIA), Department of Mathematics and Computer Science.

[5]University of Limburg (LUC), Department WNI, Universitaire Campus, B-3590 Diepenbeek, Belgium. E-mail: gyssens@luc.ac.be.

Abstract

Object-oriented applications of database systems require database transformations involving non-standard functionalities such as set manipulation and object creation, i.e., the introduction of new domain elements. To deal with these functionalities, Abiteboul and Kanellakis introduced the "determinate" transformations as a generalization of the standard domain-preserving transformations. The obvious extensions of complete standard database programming languages, however, are not complete for the determinate transformations. To remedy this mismatch, the "constructive" transformations are proposed. It is shown that the constructive transformations are precisely the transformations that can be expressed in said extensions of complete standard languages. Thereto, a close correspondence between object creation and the construction of hereditarily finite sets is established.

A restricted version of the main completeness result for the case where only list manipulations are involved is also presented.

1 Introduction

The present paper is concerned with the expressive power of object-creating database transformation languages, as they occur in object-oriented systems. To enable the reader to put our results in the right perspective, we start with a brief historical overview of the research on the expressive power of database transformation languages.

The study of the expressive power of query languages was initiated by Codd, who in a series of seminal papers [13, 14, 15] laid down the foundations of modern database theory. Some of his major contributions were the suggestions *(i)* to view a database as a relational structure; *(ii)* to view the answer of a query to a relational database as another relation; and *(iii)* to use first-order logic as a query language, which he called the relational calculus. Codd qualified a query language as *complete* if its expressive power is at least that of relational calculus or the equivalent relational algebra.

In an attempt to formulate a language-independent justification for Codd's intuitive completeness notion, Bancilhon [9] and, independently, Paredaens [28] showed the following: a relation R is the result of a relational calculus query applied to a database I if and only if *(i)* the active domain of R is included in the domain of I; and *(ii)* every automorphism of I is also an automorphism of R.

Unfortunately, the characterization of Bancilhon and Paredaens only deals with individual input-output pairs, and does not say anything about queries as a whole. For instance, it is a consequence of the result of Bancilhon and Paredaens that for each binary relation there exists a calculus expression computing its transitive closure. However, there is no *single* calculus expression computing the transitive closure for *all* binary relations [7, 22, 19].

To remedy this deficiency, Chandra and Harel [12] lifted the conditions of Bancilhon and Paredaens from individual input-output pairs to the more global level of queries as partial functions from databases to relations. The consistency criterion that resulted from this approach is that a query must be invariant under every permutation of the universe of possible domain values. In other words, the query must preserve general database isomorphisms as opposed to merely automorphisms.

Earlier, Aho and Ullman [7] argued that this consistency criterion clearly captures the nature of computations typical to database applications: database queries, while operating on the physical level of the database, must

be definable at the logical level. The condition of Chandra and Harel became known eventually as *genericity* [24].[1]

Chandra and Harel qualified a query language as *complete* if that language expresses exactly all computable, generic, queries. Since they cannot express transitive closure of a binary relation, the relational calculus and algebra are *not* complete in the sense of Chandra and Harel.

A first step towards a complete language is adding an iterative construct to the relational calculus or algebra. As this alone is not sufficient to get beyond PSPACE, one additional mechanism is needed to achieve completeness. Chandra and Harel themselves used unranked relation variables for that purpose [12].

Later, Abiteboul and Vianu [5, 6] extended the framework of Chandra and Harel to general *deterministic database transformations*, encompassing both queries and updates. They showed that completeness can also be achieved through the mechanism of *object creation*, i.e., the introduction of new domain elements. In the deterministic languages of Abiteboul and Vianu, these new objects can only appear in intermediate results of the computation and not in the final result, as only domain-preserving transformations were under consideration.

Recently, however, the explicit appearance of new objects in query and update results, in conjunction with a more effective representation and manipulation of set values, turned out to be important to support object-oriented applications of database systems [11].

To deal with this new feature, a number of new data models and query languages were proposed. Particularly influential was the work of Abiteboul and Kanellakis [4]. They proposed the language IQL, which provides the necessary mechanisms for object creation and the representation of set values.

In order to assess the expressiveness of IQL, Abiteboul and Kanellakis first had to extend the notions of genericity and completeness to database transformations allowing new domain elements in the result. This task proved to be delicate, because the creation of new objects introduces a degree of non-determinism into the formalism. Abiteboul and Kanellakis proposed the notion of *determinate transformation*: a generic, non-deterministic transformation for which the various possible results of the transformation applied

[1] Interestingly, genericity is precisely the condition a query must satisfy to be "logical" in the sense of Tarski [30].

to a given input database are equal up to renaming of new domain elements.

Since IQL is a natural extension of the earlier languages of Abiteboul and Vianu [5, 6], complete in the sense of Chandra and Harel, it was expected at first that IQL would be complete for the determinate transformations. Surprisingly, this is not the case; in a sense that can be made precise, IQL lacks the ability to *eliminate copies*.

One could view this deficiency as a weakness which should be remedied. In this vein, Abiteboul and Kanellakis proposed to extend IQL with an extra construct for copy elimination. (Recently, Denninghoff and Vianu [18] proposed an alternative extension with a more efficient construct.)

Most, if not all, transformations of practical interest can effectively be expressed in IQL, however, and hence do *not* require copy elimination. This empirical observation, in our opinion, justifies the search for a subclass of the determinate queries for which object-creating languages, such as IQL, are complete without having to consider copy elimination, and this is the subject of the present paper.

At this point, it must be emphasized that our results are general and can also be applied to a broad class of other object-creating database languages that have been investigated (e.g., [20, 25, 26, 27]). This generality stems from the computational equivalence of these languages to FO + **new** + **while**, a minimal language defined in this paper as the closure of first-order logic under unbounded looping and associating new domain elements to tuples and sets of values, and it is for this language that we prove our results. It goes without saying, however, that FO + **new** + **while** does not capture all aspects of the object-oriented database systems referred to above. We are, for instance, not concerned with typing and inheritance, two essential features of object-orientation. Rather, we see FO + **new** + **while** as a common abstraction of the concrete languages considered *only* for the aspect of object-oriented languages under consideration in this paper, that is the impact of object creation on the expressiveness of a language. In an Appendix, we make this claim precise, by concretely showing that IQL and FO + **new** + **while** have the same expressive power.

A precursor to the present study is the work by Andries and Paredaens [8] in the context of another object-creating database transformation language, called GOOD [20]. Andries and Paredaens showed that a database J is the output of a GOOD program applied to a database I if and only if there exists an extension homomorphism from the group of automorphisms of I to

the group of automorphisms of J. They argued that this condition could be seen as the generalization of the criterion of Bancilhon and Paredaens to the context of object creation. In the above-mentioned Appendix, we show that also GOOD and FO + **new** + **while** have the same expressive power, and, as a consequence, that the result of Andries and Paredaens is in fact a result about FO + **new** + **while**.

In the present paper, we show the unexpected result that simply requiring determinate transformations to satisfy the condition of Andries and Paredaens—although a "local" condition defined on individual pairs of input and output databases—yields the desired characterization of the transformations expressible in FO + **new** + **while**. We furthermore show that, in our characterization, we can replace the requirement of Andries and Paredaens by another condition, relating object creation to the construction of hereditarily finite sets[2] over the original domain elements. With this alternative condition, our characterization can be thought of as a completeness criterion for *constructive transformations*, thus establishing its naturalness and robustness. Finally, we present a restricted version of the main completeness result for the case where only list manipulations are involved. This result can be thought of as a completeness criterion for *list-constructive transformations*. It establishes a link between "pure" object-creation approaches and approaches to treating new objects as lists built over the input objects, as used in logic and functional programming (e.g., [1, 25, 26]).

This paper is further organized as follows. In Section 2, we introduce some notation and terminology, give the necessary mathematical background to capture definitions and results, review determinate object-creating database transformations, and introduce the language FO+**new**+**while**. In Section 3, we introduce and motivate the constructive transformations. In Section 4, we do the same for the list-constructive transformations. In Sections 5 and 6, we prove our completeness results. In Section 7, we present a summary and concluding remarks.

[2]Hereditarily finite sets in the context of databases have already been studied by Dahlhaus and Makowsky [17] and Hull and Su [23].

R^I

a	c	d
b	a	e
c	b	d
a	c	e
b	a	d
c	b	e

S^I

a	b
b	c
c	a

Figure 1: The database instance of Example 2.1.

2 Preliminaries

2.1 The data model

We first describe the data model that we use throughout the paper.

It is assumed that an infinite collection $\mathbf{R}$ of *relation names* is given. To each relation name R a natural number $\alpha(R)$ is associated, called the *arity* of R, such that each number is the arity of infinitely many relation names. A *database scheme* is a finite set of relation names.

It is furthermore assumed that a countably infinite universe $\mathbf{U}$ of abstract atomic values, called *objects*, is given.

An *instance* I of a database scheme $\mathcal{S}$ is a finite relational structure of type $\mathcal{S}$, consisting of a finite subset $|I|$ of $\mathbf{U}$, called the *domain*, and a mapping on $\mathcal{S}$, assigning to each relation name R of $\mathcal{S}$ a relation denoted R^I on $|I|$ of rank $\alpha(R)$ (i.e., a subset of the Cartesian product $|I|^{\alpha(R)}$), called the *content* of R. The set of all database instances of the scheme $\mathcal{S}$ is denoted by $\mathrm{inst}(\mathcal{S})$.

Example 2.1 Consider the database scheme $\mathcal{S} = \{R, S\}$ with $\alpha(R) = 3$ and $\alpha(S) = 2$. The structure I with $|I| = \{a, b, c, d, e\}$ mapping R to the ternary relation R^I and S to the binary relation S^I, both shown in Figure 1, is an instance over $\mathcal{S}$.

2.2 Mathematical notions

In this paragraph, we review some key mathematical notions essential for a good understanding of the remainder of this paper.

$R^{I'}$		
d	h	e
b	d	c
h	b	e
d	h	c
b	d	e
h	b	c

$S^{I'}$	
d	b
b	h
h	d

Figure 2: The database instance I' of Example 2.2.

Consider a function mapping objects to objects. Over these objects, structures such as sets, tuples, relations, and instances can be built. It is common mathematical practice to extend functions defined on objects to functions defined on structures built over these objects, by extending them element- or component-wise, as is illustrated in Example 2.2.

Example 2.2 Let a, b, c, d, e, and h be objects of $\mathbf{U}$ and let f_1 be the function from $\{a, b, c, d, e\}$ to $\{b, c, d, e, h\}$ mapping a to d, b to b, c to h, d to e, and e to c. The function f_1 can be element-wise extended to sets; e.g., $f_1(\{a, b, c\}) = \{d, b, e\}$. The function f_1 can be extended to tuples component-wise; e.g., $f_1((a, c, e)) = (d, h, c)$. The function f_1 can further be extended to relations and database instances; e.g., if I is the database instance over $\mathcal{S} = \{R, S\}$ in Example 2.1, then $I' = f_1(I)$ is also an instance over $\mathcal{S}$, for which $|I'| = f_1(|I|) = \{d, b, h, e, c\} = \{b, c, d, e, h\}$ and $R^{I'}$ and $S^{I'}$ are as shown in Figure 2.

As a second example, let f_2 be the function from $\{a, b, c, d, e\}$ to itself mapping a to b, b to c, c to a, d to e, and e to d. Applying the same principle as above, one can readily verify that $f_2(I) = I$

Each time we use the principle explained above to lift a function defined on objects to the level of structures, we say that we extend that function *in the standard way*.

When we compare the instance I of Example 2.1 with the instance $I' = f_1(I)$ of Example 2.2, we see that, because the function f_1 is a bijection, they are identical upon "renaming" of objects. Two such instances are called *isomorphic*, and the mapping between objects establishing this relationship is called an *isomorphism*. We define these notions formally, as well as some specializations needed further on in this paper:

Definition 2.3 *Let* J_1 *and* J_2 *be two instances of the same scheme* $\mathcal{S}$*. A bijection* $f : |J_1| \to |J_2|$ *is called an* isomorphism *from* J_1 *to* J_2 *if* $f(J_1) = J_2$*, where* f *is extended to instances in the standard way.*

For an arbitrary set of objects $V \subseteq |J_1|$*, a* V-isomorphism *is an isomorphism from* J_1 *to* J_2 *that is the identity on* V*. For an arbitrary instance* I*, an* $|I|$*-isomorphism is also called an* I*-isomorphism.*

So, instance I of Example 2.1 and instance I' of Example 2.2 are isomorphic, and f_1 is an isomorphism from I to I'. In particular, it is a $\{b\}$-isomorphism.

From Definition 2.3, it follows that each instance is isomorphic to itself, because the identity is an isomorphism from an instance to itself. However, there may also be non-identical isomorphisms from an instance to itself. In Example 2.2, we established that the bijection f_2 is an isomorphism from I to itself. Such isomorphisms are called *automorphisms*:

Definition 2.4 *Let* J *be an instance. A bijection* $f : |J| \to |J|$*, i.e., a permutation of* $|J|$*, is called an* automorphism *of* J *if* f *is an isomorphism from* J *to* J*.*

So, the function f_2 of Example 2.2 is an automorphism of the instance I of Example 2.1.

Knowledge of the automorphisms of an instance is important, because this provides information on the degree of symmetry present in that instance. The set of all automorphisms of a given instance yields a mathematical structure, which is called a *group*:

Definition 2.5 *Let* G *be set, let* $\star$ *be a total, binary operation on* G*, and let* n *be in* G*. The structure* $(G, \star, n)$ *is a* group *if the following three properties are satisfied:*

1. *the operation* $\star$ *is* associative*, i.e., for all* x*,* y*, and* z *in* G*,* $(x \star y) \star z = x \star (y \star z)$*;*
2. n *is a* neutral element *with respect to* $\star$*, i.e., for all* x *in* G*,* $x \star n = x$ *and* $n \star x = x$*; and*
3. *each element of* G *has an* inverse *with respect to* n *and* $\star$*, i.e., for all* x *in* G*, there exists* x^{-1} *in* G *such that* $x \star x^{-1} = n$ *and* $x^{-1} \star x = n$*.*

The traditional example of a group is $(\mathbf{Z}, +, 0)$, with $\mathbf{Z}$ the set of all integers and $+$ integer addition. Addition is a total binary operation on the integers, is associative, and has 0 as a neutral element; finally each integer x has $-x$ for its inverse.

Given an instance I, let $\mathrm{Aut}(I)$ denote the set of all automorphisms of I. We observe the following important property of this set:

Proposition 2.6 *Let I be an instance, and let $\mathrm{id}_{|I|}$ denote the identity mapping on $|I|$. Let $\circ$ denote composition of mappings. Then $(\mathrm{Aut}(I), \circ, \mathrm{id}_{|I|})$ is a group.*

Proof. The composition of mappings is associative and has the identity mapping as a neutral element. For a bijection, one can consider the inverse mapping, and this is an inverse for the original mapping with respect to identity and composition. To see that $(\mathrm{Aut}(I), \circ, \mathrm{id}_{|I|})$ is a group, it now suffices to observe that *(i)* the composition of two automorphisms of I is again an automorphism of I; *(ii)* an automorphism is a bijection; and *(iii)* the inverse mapping of an automorphism of I is again an automorphism of I. ■

Example 2.7 The automorphism group $(\mathrm{Aut}(I), \circ, \mathrm{id}_{|I|})$ of the instance I in Example 2.1 consists of the following 6 permutations on $|I| = \{a, b, c, d, e\}$: the identity mapping $\mathrm{id}_{|I|}$; the automorphism f_2 of Example 2.2; the automorphism f_3 mapping a to b, b to c, c to a, d to d, and e to e; the automorphism f_4 mapping a to c, b to a, c to b, d to e, and e to d; the automorphism f_5 mapping a to c, b to a, c to b, d to d and e to e; and the automorphism f_6 mapping a to a, b to b, c to c, d tot e, and e to d. Notice that the composition of two automorphisms of $\mathrm{Aut}(I)$ is again an automorphism of $\mathrm{Aut}(I)$, e.g., $f_3 \circ f_2 = f_4$, and that the inverse of each automorphism of $\mathrm{Aut}(I)$ is again an automorphism of $\mathrm{Aut}(I)$, e.g., $f_2^{-1} = f_4$.

Database transformations typically preserve symmetries present in the input instance in a sense to be made precise later. Since the symmetries of an instance are formally described by its automorphism group, we need a tool to compare automorphism groups of different instance. Such a tool can be found in the general notion of *group homomorphism*:

Definition 2.8 *Let* $(G, \star, n)$ *and* $(H, \diamond, m)$ *be two groups. A total function* $\psi : G \to H$ *is a* group homomorphism *from* $(G, \star, n)$ *to* $(H, \diamond, m)$ *if for all* x *and* y *in* G, $\psi(x \star y) = \psi(x) \diamond \psi(y)$.[3]

Intuitively, a group homomorphism is a mapping between the sets on which the groups are built that is "compatible" with the group structure.

Example 2.9 Let I be the instance of Example 2.1. Then $\mathrm{Aut}(I) = \{\mathrm{id}_{|I|}, f_2, f_3, f_4, f_5, f_6\}$ as in Example 2.7. Let I' be the instance of Example 2.2. It is readily seen that $\mathrm{Aut}(I') = \{\mathrm{id}_{|I'|}, g_2, g_3, g_4, g_5, g_6\}$ where g_2 maps d to b, b to h, h to d, e to c, and c to e; g_3 maps d to b, b to h, h to d, e to e, and c to c; g_4 maps d to h, b to d, h to b, e to c, and c to e; g_5 maps d to h, b to d, h to b, e to e, and c to c; and g_6 maps d to d, b to b, h to h, e to c, and c to e. The function $\psi : \mathrm{Aut}(I) \to \mathrm{Aut}(I')$ mapping $\mathrm{id}_{|I|}$ to $\mathrm{id}_{|I'|}$, f_2 to g_2, f_3 to g_3, f_4 to g_4, f_5 to g_5, and f_6 to g_6 is a group homomorphism from $(\mathrm{Aut}(I), \circ, \mathrm{id}_{|I|})$ to $(\mathrm{Aut}(I'), \circ, \mathrm{id}_{|I'|})$.

Notice that the group homomorphism ψ in Example 2.9 is a bijection; in general, however, group homomorphisms need not be bijective.

We close this paragraph with a notational issue. Whenever in a group $(G, \star, n)$ the operation $\star$ and the neutral element n are implicit from the context, we denote that group simply as G. Thus, we speak about the automorphism group $\mathrm{Aut}(I)$ of an instance I.

2.3 Database transformations

Next, we turn to database transformations. On the most general level, database transformations were defined by Abiteboul and Vianu [5] as follows:

Definition 2.10 *Let* $\mathcal{S}_{\mathrm{in}}$ *and* $\mathcal{S}_{\mathrm{out}}$ *be two database schemes. A* transformation from $\mathcal{S}_{\mathrm{in}}$ to $\mathcal{S}_{\mathrm{out}}$ *is a recursively enumerable input-output relationship* $Q \subseteq \mathrm{inst}(\mathcal{S}_{\mathrm{in}}) \times \mathrm{inst}(\mathcal{S}_{\mathrm{out}})$ *which is invariant under every permutation of* $\mathbf{U}$.

[3]From this condition, it can be inferred that $\psi(n) = m$.

The requirement of invariance under permutations is called *genericity*. It can be visualized by the following commuting diagram:

$$\begin{array}{ccc} I & \xrightarrow{Q} & J \\ \downarrow f & & \downarrow f \\ I' & \xrightarrow{Q} & J' \end{array}$$

The above diagram should be read as follows. The relationship Q is a transformation from a scheme $\mathcal{S}_{\text{in}}$ to a scheme $\mathcal{S}_{\text{out}}$, I and I' are instances of $\mathcal{S}_{\text{in}}$, and J and J' are instances of $\mathcal{S}_{\text{out}}$. If (I, J) is an input-output pair in Q, and if there exists a permutation f on $\mathbf{U}$ such that (for its standard extension) $f(I) = I'$ and $f(J) = J'$, then (I', J') is also an input-output pair in Q. Thus, generic transformations treat isomorphic instances uniformly; if, e.g., in the above diagram, all instances involve a binary relation symbol R, and R^J happens to be the transitive closure of R^I, then necessarily $R^{J'}$ is the transitive closure of $R^{I'}$.

In general, database transformations as defined in Definition 2.10 are relationships. Database transformations that are functions, i.e., relationships such that to each input instance I there corresponds at most one output instance, are called *deterministic*, and these are the transformations that are encountered in traditional database applications. In this paper, however, we consider transformations involving *object creation*, i.e., the introduction of new elements in the domain, and these are necessarily non-deterministic, by the genericity requirement.

Indeed, assume Q is a transformation containing an input-output pair (I, J) such that $|J|$ contains an object o not in $|I|$. Let o' be another object not in $|I|$ nor in $|J|$, and consider the transposition $f = (o\ o')$ as a permutation of $\mathbf{U}$. After extending f in the standard way, let $I' = f(I)$ and $J' = f(J)$. Notice that $I' = I$ (neither o nor o' occurs in I) and $J' \neq J$ (o occurs in J but not in J' and o' occurs in J' but not in J). By the genericity requirement, $(I', J') = (I, J')$ must be an input-output pair in Q. Hence, Q can yield at least the two different outputs J and J' on input I, and is therefore non-deterministic.

Nonetheless, the transformation Q can still have a "deterministic effect" if the particular choice of a new object is the only degree of non-determinism that is allowed. A comparable situation arises in programming languages allowing dynamic allocation of memory cells: although the computation is

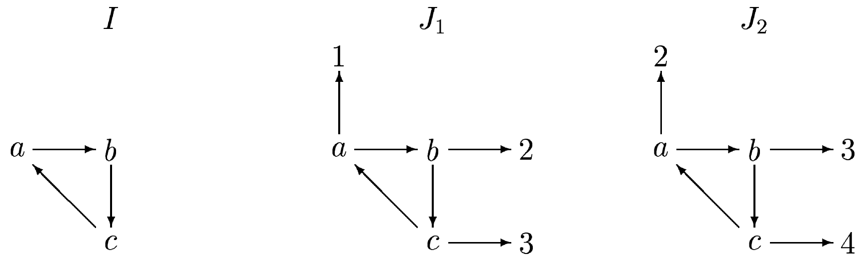


Figure 3: Application of transformation Q_1 of Example 2.11.

essentially deterministic, the programmer does not know in advance which cells will be allocated as they are chosen by the system.

The desire to study database transformations involving object creation, but having a "deterministic effect," have led Abiteboul and Kanellakis [4] to introduce the term *determinate transformation* for a database transformation in which any non-determinism solely stems from the particular choice of new domain elements.

Example 2.11 Let G be a binary relation name. An instance I of $\{G\}$ can be interpreted as a directed graph with set of nodes $|I|$ and set of edges G^I. Consider the transformation Q_1 from $\{G\}$ to $\{G\}$ defined as follows: $Q_1(I, J)$ if J is obtained from I by adding for each node x a new node x' with an edge from x to x'. Formally, if $|I| = \{x_1, \ldots, x_n\}$, then $|J| = \{x_1, \ldots, x_n, x'_1, \ldots, x'_n\}$ and $G^J = G^I \cup \{(x_i, x'_i) \mid i = 1, \ldots, n\}$. Figure 3 shows three instances I, J_1 and J_2 such that J_1 and J_2 are two possible results of Q_1 applied to I (i.e., $Q_1(I, J_1)$ and $Q_1(I, J_2)$). Transformation Q_1 is determinate: for example, J_2 can be obtained from J_1 by renaming the newly added nodes.

Now consider the non-deterministic transformation Q_2 from $\{G\}$ to $\{G\}$ defined as follows: $Q_2(I, J)$ if $|J| = |I|$ and G^J can be obtained from G^I by deleting an arbitrary edge out of every node that has outgoing edges. Figure 4 shows an instance I' and two possible results J_3 and J_4 of Q_2 applied to I'. Transformation Q_2 is not determinate.

We now formally define determinate transformations:

Definition 2.12 *A transformation Q is* determinate *if the following conditions hold:*

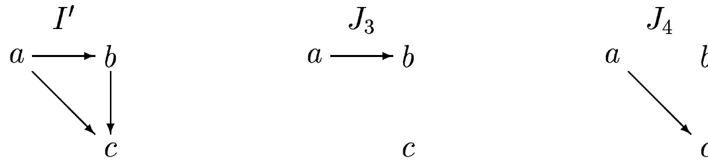


Figure 4: Application of transformation Q_2 of Example 2.11.

1. *if* $Q(I, J)$*, then* $|I| \subseteq |J|$*;*
2. *if* $Q(I, J_1)$ *and* $Q(I, J_2)$*, then* J_1 *and* J_2 *are* I*-isomorphic (Definition 2.3).*

The second requirement of the above definition captures the intuition illustrated in Example 2.11: the I-isomorphism that exists from J_1 to J_2 can be interpreted as a "renaming" of the new domain elements introduced in J_1. Hence, in practice, for a determinate transformation, it suffices to specify only one result which is then representative for all possible results.

Also notice:

Proposition 2.13 *If* Q *is a determinate transformation such that for each pair of instances* (I, J) *with* $Q(I, J)$ *we have* $|I| = |J|$*, then* Q *is deterministic.*

Proof. If $Q(I, J_1)$ and $Q(I, J_2)$ then J_1 and J_2 are I-isomorphic. But $|I| = |J_1| = |J_2|$ and thus the only I-isomorphism from J_1 to J_2 is the identity. Hence, $J_1 = J_2$. ■

The first requirement of Definition 2.12 is of a purely technical nature. Although it does not impose any restriction on deletion of tuples from relations, it does not allow the removal of elements from the domain of an instance, even if these elements no longer occur in the content of any relation name in the output scheme. This technical restriction guarantees determinate transformations to be closed under composition and allows simple definitions of a number of important notions later on in this paper.

A transformation Q satisfying the second requirement of Definition 2.12 but not the first can be "completed" to a determinate transformation as follows: for each input-output pair (I, J), augment $|J|$ with the elements of $|I|$ not yet occurring in $|J|$. In the sequel, for simplicity of presentation, we

will sometimes leave this completion implicit and treat such transformations as if they were formally determinate.

2.4 The programming language FO + new + while

We next introduce a simple and general programming language for expressing determinate transformations, which we denote by FO + **new** + **while**. The language is an extension of first-order logic with object creation, set representation, and iteration, and will serve as an abstract formulation of the languages IQL and GOOD mentioned in the Introduction and the various other object-oriented database languages which are at most as expressive. A formal proof that IQL, GOOD, and FO + **new** + **while** all have the same expressive power is given in the Appendix.

Programs in our language are built from three types of statements and while-loops, defined below.

A key construct in all three types of statements is the FO expression. A *k-ary FO expression* Φ over a scheme $\mathcal{S}$ is an expression of the form $\{(x_1, \ldots, x_k) \mid \varphi\}$ with φ a first-order logic formula over $\mathcal{S}$ whose free variables are among $x_1, \ldots, x_k$. Given an instance I of $\mathcal{S}$, Φ defines a k-ary relation $\Phi(I)$ on $|I|$, as in the relational calculus.

Syntactically, an *FO statement* over a scheme $\mathcal{S}$ is an expression of the form $R := \Phi$, with R a k-ary relation name and Φ a k-ary FO expression over $\mathcal{S}$. Semantically, this statement defines the determinate transformation Q from $\mathcal{S}$ to $\mathcal{S} \cup \{R\}$ given by $Q(I, J)$ if and only if $R^J = \Phi(I)$, $S^J = S^I$ for each $S \neq R$ in $\mathcal{S}$, and $|J| = |I|$.

Notice that the above definition—as well as similar subsequent definitions—covers both the case where R is in $\mathcal{S}$ as the case where R is not in $\mathcal{S}$.

Syntactically, a *tuple-new statement* over a scheme $\mathcal{S}$ is an expression of the form $R :=$ **tuple-new** Φ, with R a $k+1$-ary relation name and Φ a k-ary FO expression over $\mathcal{S}$. Semantically, this statement defines a determinate transformation Q from $\mathcal{S}$ to $\mathcal{S} \cup \{R\}$ as follows. Let I be an instance of $\mathcal{S}$, and let $\Phi(I) = \{t_1, \ldots, t_n\}$. Then $Q(I, J)$ if and only if

- $R^J = (\{t_1\} \times \{o_1\}) \cup \cdots \cup (\{t_n\} \times \{o_n\})$, where $o_1, \ldots, o_n$ are n different objects not in $|I|$;
- $S^J = S^I$ for each $S \neq R$ in $\mathcal{S}$; and

R	
a	b
a	c
b	b
b	c
c	d
d	d

S		
a	b	1
a	c	2
b	b	3
b	c	4

T	
a	5
b	5
c	6
d	6

Figure 5: Illustration to Examples 2.14 and 2.15.

- $|J| = |I| \cup \{o_1, \ldots, o_n\}$.

Tuple-new statements add new domain elements to an instance by a mechanism similar to that employed by Abiteboul and Vianu in the language detDL [6].

Example 2.14 Consider Figure 5. Let I be the instance of $\{R\}$ shown with $|I| = \{a, b, c, d\}$.

The instance of $\{R, S\}$ shown is the output of the statement

$$S := \textbf{tuple-new}\ \{(x, y) \mid R(x, y) \wedge (\exists z)(R(x, z) \wedge z \neq y)\}$$

applied to I.

Syntactically, a *set-new statement* over a scheme $\mathcal{S}$ is an expression of the form $R :=$ **set-new** Φ, with R a binary relation name and Φ a binary FO expression over $\mathcal{S}$. Semantically, this statement defines a determinate transformation Q from $\mathcal{S}$ to $\mathcal{S} \cup \{R\}$ as follows. Let I be an instance of $\mathcal{S}$, and let $\Phi(I) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. Then $Q(I, J)$ if and only if

- $R^J = \{(x_1, o_1), \ldots, (x_n, o_n)\}$, where $o_1, \ldots, o_n$ are objects not in $|I|$ satisfying $o_i = o_j$ if and only if $\{y \mid (x_i, y) \in \Phi(I)\} = \{y \mid (x_j, y) \in \Phi(I)\}$;
- $S^J = S^I$ for each $S \neq R$ in $\mathcal{S}$; and
- $|J| = |I| \cup \{o_1, \ldots, o_n\}$.

Set-new statements associate new domain values to sets of existing values *in a unique way.* (Tuple-new statements are not sufficient for this purpose [31].)

Example 2.15 Consider Figure 5 and the instance I of $\{R\}$ shown with $|I| = \{a, b, c, d\}$.

The instance of $\{R, T\}$ shown is the output of the statement

$$T := \textbf{set-new}\ R$$

applied to I. In the content of T, the object '5' represents the set $\{b, c\}$ and the object '6' the set $\{d\}$.

Programs can now be built inductively from statements using composition (;) and while-loops of the form **while** φ **do** P **od**, with φ a first-order sentence and P a program. Programs express database transformations in the obvious manner. When interpreting programs as database transformations, we allow that relations used only for storing intermediate results of the computation (as well as input relations that are no longer needed) are ignored in the final result. Also, objects occurring only in such intermediate relations may be ignored in the domain of the final result. To syntactically ensure that programs express transformations in a unique way, one can additionally specify the output scheme, and a superset of the output scheme with the names of the relations whose objects constitute the domain of the final result. By structural induction, it is readily verified that transformations expressed by programs are generic, i.e., invariant under permutations of $\mathbf{U}$, and determinate. In the sequel we will not distinguish between a program and the transformation it expresses.

The definition of the programming language FO + **new** + **while** is now complete. We conclude this section with a few examples of concrete programs.

Example 2.16 Transformation Q_1 of Example 2.11 can be expressed in FO + **new** + **while** as follows:

$G' :=$ **tuple-new** $\{(x) \mid \textbf{true}\}$;
$G := G \cup G'$.

In the above program, G' is an auxiliary relation name which is ignored in the final result.

Example 2.17 Consider the if-then construct **if** φ **then** P **fi**, with φ a first-order sentence and P a program, with the standard semantics. Assuming H is a relation name not occurring in P, this transformation can be expressed as follows:

$H := \{() \mid \varphi\}$;
while $H \neq \emptyset$ **do**
  $H := \emptyset$;
  P
od.

Example 2.18 Let R and S be binary relation names. The transformation from $\{R\}$ to $\{S\}$ computing transitive closure can be expressed as follows:

$\mathit{Old} := \emptyset$;
$S := R$;
while $S \neq \mathit{Old}$ **do**
  $\mathit{Old} := S$;
  $S := S \cup \{(x, y) \mid (\exists z)(R(x, z) \wedge S(z, y))\}$
od.

Example 2.19 Zero-ary FO expressions as used in Example 2.17 can also be used to introduce new objects "from scratch." Indeed, the one-line program

$$R := \textbf{tuple-new}\ \{() \mid \textbf{true}\}$$

yields a unary relation, containing one new object.

Example 2.20 For a symmetric, anti-reflexive binary relation G, i.e., an undirected graph without self-loops, the following program computes the dual graph G^* whose nodes are the edges of G and whose edges indicate incidence in G:

if $(\forall x)\neg G(x, x) \wedge (\forall x)(\forall y)(G(x, y) \rightarrow G(y, x))$ **then**
  $E_1 :=$ **tuple-new** G;
  $E_2 := \{(z, x) \mid (\exists w)(E_1(x, w, z) \vee E_1(w, x, z))\}$;

$Reach := \textbf{tuple-new}\ \{(x) \mid \textbf{true}\};$
$Head := \{(l, x) \mid Reach(x, l)\};$
$Tail := \textbf{tuple-new}\ \{(l) \mid (\exists x) Head(l, x)\};$
$Member := Head;$
$Next := \textbf{tuple-new}\ \{(l, t, y) \mid Tail(l, t) \wedge (\exists x)(Head(l, x) \wedge G(x, y) \wedge y \neq x)\};$
while $Next \neq \emptyset$ **do**
  $Head := Head \cup \{(l_1, y) \mid (\exists l)(\exists l_2) Next(l, l_1, y, l_2)\};$
  $Tail := Tail \cup \{(l_1, l_2) \mid (\exists l)(\exists y) Next(l, l_1, y, l_2)\};$
  $Member := Member \cup \{(l, y) \mid (\exists l_1)(\exists l_2) Next(l, l_1, y, l_2)\};$
  $Next := \textbf{tuple-new}\ \{(l, l_2, z) \mid (\exists l_1)(\exists y)(Next(l, l_1, y, l_2) \wedge G(y, z) \wedge \neg Member(l, z))\}$
od.

Figure 6: Program of Example 2.21.

  $E_3 := \textbf{set-new}\ E_2;$
  $E_4 := \{(e, x) \mid (\exists z)(E_3(z, e) \wedge E_2(z, x))\};$
  $G^* := \{(e, f) \mid (\exists x)(E_4(e, x) \wedge E_4(f, x))\}$
fi.

Note the use of **set-new** to create a unique object for each undirected edge.

Example 2.21 Let G be a binary relation, viewed as a directed graph, in which every node has out-degree at most one. We wish to compute the function *Reach* that associates to each node x the list of nodes reachable from x in the order they appear on the path leaving x.

Lists are represented in the well-known way [1] as objects on which *Head* and *Tail* functions are defined. (Empty lists have no head or tail.) The functions *Reach*, *Head*, and *Tail* are stored in the form of binary relations.

The above-described transformation from $\{G\}$ to $\{Reach, Head, Tail\}$ can be expressed by the program shown in Figure 6. In Figure 7, the instance of $\{Reach, Head, Tail\}$ shown is the result of applying this program to the instance I of $\{G\}$ shown with $|I| = \{a, b, c, d, e\}$.

G

a	c
b	c
c	d
d	b

Reach

a	α_1
b	β_1
c	γ_1
d	δ_1
e	ε_1

Tail

α_1	α_2
α_2	α_3
α_3	α_4
α_4	α_5
β_1	β_2
β_2	β_3
β_3	β_4
γ_1	γ_2
γ_2	γ_3
γ_3	γ_4

Head

α_1	a
α_2	c
α_3	d
α_4	b
β_1	b
β_2	c
β_3	d
γ_1	c
γ_2	d
γ_3	b
δ_1	d
δ_2	b
δ_3	c
ε_1	e

Figure 7: Application of the program in Figure 6.

3 Constructive transformations

Although all deterministic transformations are expressible in FO + **new** + **while** [5, 6], there are determinate transformations (involving object creation) not expressible in FO + **new** + **while**, due to the absence of a mechanism to eliminate copies (see the Introduction). All non-expressible transformations known are somewhat artificial, however. We will argue in this paper that all non-expressible transformations are indeed "non-constructive," in a sense which we will make precise in this section.

Our departure point is a result by Bancilhon [9] and, independently, Paredaens [28], giving a necessary and sufficient condition for an instance J to be computable from an instance I by a sequence of FO statements. This condition is stated in terms of the automorphisms of I and J and their relationships.

Proposition 3.1 [9, 28] *Let I and J be instances with $|I| = |J|$. There exists a sequence P of FO statements such that $P(I, J)$, if and only if* $\mathrm{Aut}(I) \subseteq \mathrm{Aut}(J)$.

R^I	
a	b
b	c
c	b
d	e
d	g
g	d

S^J	
a	b
b	c
c	b
a	c
b	b
c	c
d	e
e	g
g	e
d	g
e	e
g	g

Figure 8: Illustration to Example 3.2.

Proposition 3.1 is known as the *BP-completeness* of FO [12].

Example 3.2 Let R and S be binary relation names. Let I be the instance of $\{R\}$ and let J be the instance of $\{S\}$ for which $|I| = |J| = \{a, b, c, d, e, h\}$ and R^I and S^J are as shown in Figure 8. Then $\text{Aut}(I) = \{\text{id}_{|I|}, f_2\}$ with f_2 swapping a and d, b and e, and c and g, and $\text{Aut}(J) = \{\text{id}_{|J|}, f_2, f_3, f_4, f_5, f_6\}$ with f_3 fixing a and d and swapping b and c and e and g; f_4 fixing a, b, c, and d and swapping e and g; f_5 fixing a, d, e, and g and swapping b and c; and f_6 swapping a and d, b and g, and c and e. Thus, $\text{Aut}(I) \subseteq \text{Aut}(J)$, and by Proposition 3.1, there is a sequence P of FO statements such that $P(I, J)$. One such sequence consists of the single statement

$$S := \{(x, y) \mid R(x, y) \vee (\exists z)(R(x, z) \wedge R(z, y))\}.$$

Andries and Paredaens [8] recently generalized the BP-completeness to the context of object creation, where input-output pairs of instances (I, J) with $|I| \subseteq |J|$ are considered. Thereto, they defined the following:

Definition 3.3 *An* extension mapping *from I to J is a mapping ψ : $\text{Aut}(I) \rightarrow \text{Aut}(J)$ such that for each $f \in \text{Aut}(I)$, $\psi(f)$ is an extension of f. An* extension homomorphism *is an extension mapping that is also a group homomorphism.*

Observe that an extension mapping is injective. Hence, an extension homomorphism from I to J faithfully embeds $\text{Aut}(I)$ in $\text{Aut}(J)$ and is therefore the natural generalization of the inclusion $\text{Aut}(I) \subseteq \text{Aut}(J)$ to the case where $|J|$ contains, but is not necessarily equal to, $|I|$. Using the formalism of the GOOD transformation language, Andries and Paredaens obtained the following generalization of Proposition 3.1:

Proposition 3.4 [8] *Let I and J be instances with $|I| \subseteq |J|$. There exists an* FO+**new**+**while** *program P such that $P(I, J)$, if and only if there exists an extension homomorphism from I to J.*

(In the Appendix, it is shown formally that GOOD and FO + **new** + **while** have the same expressive power.)

Example 3.5 Consider the transformation Q_1 of Example 2.11 and its input-output pair (I, J_1) shown in Figure 3. In Example 2.16, we showed that Q_1 can be expressed in FO + **new** + **while**. Thus, by Proposition 3.4, there must exist an extension homomorphism from $\text{Aut}(I)$ to $\text{Aut}(J_1)$, which we are going to show explicitly.

It is readily verfied that $\text{Aut}(I) = \{\text{id}_{|I|}, f_2, f_3\}$, with f_2 the automorphism of I mapping a to b, b to c, and c to a, and f_3 the automorphism of I mapping a to c, b to a, and c to b. Similarly, $\text{Aut}(J_1) = \{\text{id}_{|J_1|}, g_2, g_3\}$, with g_2 the automorphism of J_1 mapping a to b, b to c, c to a, 1 to 2, 2 to 3, and 3 to 1, and g_3 the automorphism of J_1 mapping a to c, b to a, c to b, 1 to 3, 2 to 1, and 3 to 2. The restrictions of $\text{id}_{|J_1|}$, g_2, and g_3 to $|I|$ are $\text{id}_{|I|}$, f_2, and f_3, respectively. Therefore, $\text{id}_{|J_1|}$, g_2, and g_3 are extensions of $\text{id}_{|I|}$, f_2, and f_3, respectively. Thus, the function $\psi : \text{Aut}(I) \to \text{Aut}(J_1)$ mapping $\text{id}_{|I|}$ to $\text{id}_{|J_1|}$, f_2 to g_2, and f_3 to g_3 is an extension mapping. It remains to show that ψ is a group homomorphism. To do this we must show that ψ is "compatible" with composition of automorphisms. The only non-trivial compositions of automorphisms of I to be considered are $f_3 \circ f_2$ and $f_2 \circ f_3$. We have $\psi(f_3 \circ f_2) = \psi(\text{id}_{|I|}) = \text{id}_{|J_1|} = g_3 \circ g_2 = \psi(f_3) \circ \psi(f_2)$, and the same with the indices 2 and 3 reversed. Thus ψ is a group homomorphism.

In order for the reader to gain a better understanding of Proposition 3.4, we need to point out that an input-output pair of a determinate transformation always admits an extension *mapping*. To see this, let Q be a determinate transformation, and assume $Q(I, J)$. We can define an extension mapping ψ

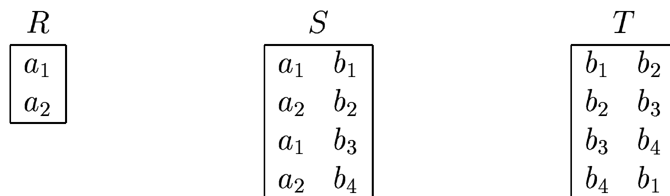


Figure 9: An input-output pair of the transformation Q of Example 3.6.

from I to J as follows. Let $f \in \mathrm{Aut}(I)$. We can extend f to a permutation of the whole of $\mathbf{U}$ by making it the identity outside $|I|$. By the genericity of Q, $Q(f(I), f(J))$. By the determinacy of Q, there exists an I-isomorphism g from $f(J)$ to J. Then $\psi(f) := g \circ f \in \mathrm{Aut}(J)$ is an extension of f.

The difficult part of the proof of Proposition 3.4 (and, for that matter, also of Proposition 3.1) is the if-part. The only-if part is straightforward to prove, and can be used to show that not all determinate transformations are expressible by FO + **new** + **while** programs. Indeed, while all determinate transformations admit an extension mapping for every input-output pair, they do not necessarily admit an extension *homomorphism*, as is shown by the following adaptation of an example by Abiteboul [2]:

Example 3.6 Let R be a unary relation name and S and T be binary relation names. We define a determinate transformation Q from $\{R\}$ to $\{S, T\}$ as follows. Consider Figure 9. Let I be the instance of $\{R\}$ with R^I as shown. Then $Q(I, J)$ if and only if S^J and T^J are as shown with b_1, b_2, b_3, and b_4 four new objects not in $|I|$, and $|J| = |I| \cup \{b_1, b_2, b_3, b_4\}$. All other input-output pairs of Q are of the form $(f(I), f(J))$ with f a permutation of $\mathbf{U}$.

There exist several extension mappings from I to J. Each such extension mapping, however, extends the transposition of a_1 and a_2 (of order 2) to a cyclic permutation of $\{b_1, \ldots, b_4\}$ (of order 4).[4] Hence, since homomorphisms cannot increase the order, there is no extension homomorphism from I to J and, as a consequence, Q cannot be expressed by an FO + **new** + **while** program.

[4]The *order* of an element x in a finite group $(G, \star, n)$ is the least non-zero natural number p such that $x \star \cdots \star x$ (p times) equals the neutral element n.

The example of a determinate transformation not expressible in FO + **new**+**while** given in Example 3.6 is very artificial. This raises the question of understanding the class of transformations expressible in FO + **new** + **while**. Any reasonable class of transformations will certainly have to include all conventional deterministic transformations. As observed in Proposition 2.13, these are precisely the determinate transformations that do not create new objects. Hence, we must find a natural restriction on object creation, i.e., impose conditions on the new domain elements occurring in the output of a determinate transformation.

To find such a condition, we have followed Dahlhaus and Makowsky, who studied the semantics of high-level programming languages based on "hereditarily finite sets" [16, 17]. In [17, page 5], they argued convincingly as follows:

> As much as set theory is rich enough to model virtually all objects encountered in mathematics, the cumulative hierarchy of hereditarily finite sets is rich enough to model all finite objects one may encounter in computer science. [...] the foundational question of what it means to *compute new objects from a given finite set of objects* can be adequately settled in this model.

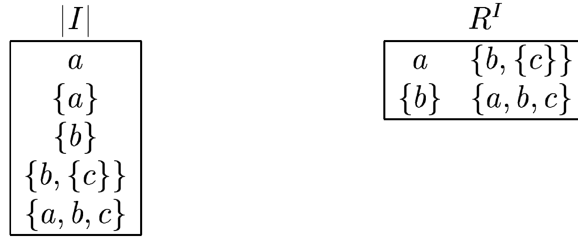
Hereditarily finite sets with "ur-elements" are well-known (e.g., [10]). We incorporate them in our framework as follows:

Definition 3.7 *Let D be a subset of* $\mathbf{U}$. *The set $HF(D)$ of* hereditarily finite sets with ur-elements in D *is the smallest set such that* (i) $D \subseteq HF(D)$; *and* (ii) *each finite subset of $HF(D)$ is also an element of $HF(D)$.*

For example, if $D = \{a, b, c\}$, then a, $\{a\}$, $\{a, b, c\}$, $\emptyset$, $\{\emptyset\}$, and $\{a, b, \{b, c\}, \{a, \{c\}\}\}$ are all in the infinite set $HF(D)$.

The main thesis put forward in this paper is that the "natural" determinate transformations are precisely those for which the new domain elements in the output can alternatively be viewed as hereditarily finite sets constructed over the domain elements of the input. We will call such transformations *constructive.* To define constructive transformations formally, we first need to define HF-instances and HF-transformations.

An *HF-instance* I is defined as an ordinary instance in Section 2, the only difference being that the domain $|I|$ is a subset of $HF(\mathbf{U})$ instead of



$\|I\|$
a
$\{a\}$
$\{b\}$
$\{b, \{c\}\}$
$\{a, b, c\}$

R^I	
a	$\{b, \{c\}\}$
$\{b\}$	$\{a, b, c\}$

Figure 10: Example of an HF-instance.

U. Relations and tuples on $HF(\mathbf{U})$ are called *HF-relations* and *HF-tuples*, respectively. In contrast, ordinary instances, relations, and tuples will sometimes be called *flat*. The set of all HF-instances of a scheme $\mathcal{S}$ is denoted by HF-inst($\mathcal{S}$). A simple example of an HF-instance I of the scheme $\{R\}$, with R binary, is shown in Figure 10, where a, b, and c are elements of **U**.

Definition 3.8 *Let $\mathcal{S}_{\text{in}}$ and $\mathcal{S}_{\text{out}}$ be two schemes. An* HF-transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ *is a partial-recursive function $Q : \text{inst}(\mathcal{S}_{\text{in}}) \rightarrow \text{HF-inst}(\mathcal{S}_{\text{out}})$ which* (i) *(viewed as a binary relationship) is invariant under every permutation of* **U** *(genericity) and* (ii) *satisfies $|Q(I)| \subseteq HF(|I|)$ whenever Q is defined on I.*

We also need the important notion of "isomorphic representation":

Definition 3.9 *Let Q be a transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$, and let Q' be a HF-transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$, such that for each pair of instances (I, J), $Q(I, J)$ if and only if* (i) *$Q'(I)$ is defined; and* (ii) *J is I-isomorphic to $Q'(I)$.*[5] *Then Q is called an* isomorphic representation of Q'.

If Q is an isomorphic representation of Q', we also say for any output pair (I, J) of Q that J is an isomorphic representation of $Q'(I)$.

The proof of the following proposition is trivial:

Proposition 3.10 *For each HF-transformation Q' there is a unique transformation Q that isomorphically represents Q'.*

[5] Isomorphisms from ordinary instances to HF-instances are defined in the same way as isomorphisms from ordinary instances to ordinary instances.

Hence, we can speak about *the* isomorphic representation of an HF-transformation.

We now define the constructive transformations formally as follows:

Definition 3.11 *A transformation is called* constructive *if it is the isomorphic representation of some HF-transformation.*

The following property of constructive transformations, guaranteeing that constructivity implies both determinacy and the condition of Andries and Paredaens, is a first indication of the soundness of the above definition:

Proposition 3.12 *Let Q be a constructive transformation. Then* (i) *Q is determinate; and* (ii) *for each pair of instances (I, J) with $Q(I, J)$, there exists an extension homomorphism from I to J.*

Proof. Let Q' be an HF-transformation of which Q is the isomorphic representation. We first show that Q is determinate. Thereto, assume $Q(I, J_1)$ and $Q(I, J_2)$. Since J_1 and J_2 are isomorphic representations of $Q'(I)$, there are I-isomorphisms f_1 and f_2 from J_1 and J_2 to $Q'(I)$, respectively. Hence $f_2^{-1} \circ f_1$ is an I-isomorphism from J_1 to J_2, as required.

To show that Q admits an extension homomorphism for every input-output pair, assume $Q(I, J)$. Since J is an isomorphic representation of $Q'(I)$, there exists an I-isomorphism g from J to $Q'(I)$. Now define $\psi : \mathrm{Aut}(I) \to \mathrm{Aut}(J) : f \mapsto g^{-1} \circ f \circ g$, where f is extended to $HF(|I|)$ in the standard way. To see that ψ is well-defined, let $f \in \mathrm{Aut}(I)$. Then

$$\begin{aligned} \psi(f)(J) = g^{-1} \circ f \circ g(J) &= g^{-1}(f(Q'(I))) && \text{(definition of } g\text{)} \\ &= g^{-1}(Q'(f(I))) && \text{(genericity of } Q'\text{)} \\ &= g^{-1}(Q'(I)) && (f \in \mathrm{Aut}(I)) \\ &= J, \end{aligned}$$

whence $\psi(f) \in \mathrm{Aut}(J)$. Since g is the identity on $|I|$, ψ is also an extension mapping. Finally, it is readily verified that ψ is a group homomorphism:

$$\begin{aligned} \psi(f \circ h) &= g^{-1} \circ f \circ h \circ g \\ &= g^{-1} \circ f \circ g \circ g^{-1} \circ h \circ g \\ &= \psi(f) \circ \psi(h). \end{aligned}$$

Hence, ψ is an extension homomorphism, as required. ∎

The transformation Q not expressible in FO + **new** + **while** shown in Example 3.6 does not satisfy the Andries-Paredaens condition and thus is not constructive.

We will next show that satisfaction of the Andries-Paredaens condition for each input-output pair is not only necessary, but also sufficient for determinate transformations to be constructive.

Theorem 3.13 *Let Q be a transformation. Then Q is constructive if and only if* (i) *Q is determinate; and* (ii) *for each pair of instances (I, J) with $Q(I, J)$, there exists an extension homomorphism from I to J.*

Proof. Since the only-if implication is given by Proposition 3.12, it suffices to show the if-implication.

For each pair of instances (I, J) with $Q(I, J)$ there exists a program P in FO + **new** + **while** with $P(I, J)$ (Proposition 3.4). Let $P_{I,J}$ be the first such program in some standard recursive enumeration of all programs. By the genericity of FO + **new** + **while** programs, we have

$$P_{I,J} = P_{f(I),f(J)} \qquad \text{for each permutation } f \text{ of } \mathbf{U}. \qquad (*)$$

The execution of $P_{I,J}$ on I traces a finite sequence of FO statements, tuple-new statements and set-new statements. Let $\ell_{I,J}$ be the length of that sequence. For an integer k, $0 \leq k \leq \ell_{I,J}$, let I_k be the intermediate result of the program $P_{I,J}$ applied on I after execution of the k-th statement in the sequence. Notice that $I_0 = I$, $I_{\ell_{I,J}} = J$, and $|I| = |I_0| \subseteq |I_1| \subseteq \cdots \subseteq |I_{\ell_{I,J}}| = |J|$. We construct an injective mapping $f_{I,J} : |J| \to HF(|I|)$ recursively as follows.

Let $o \in |J|$. If $o \in |I_0| = |I|$, we define $f_{I,J}(o) := o$. Now assume that for some k, $0 < k \leq \ell_{I,J}$, $f_{I,J}$ has been defined on all objects in $|I_{k-1}|$. If $o \in |I_k| - |I_{k-1}|$, there are two possibilities:

1. The instance I_k results from I_{k-1} by a tuple-new statement of the form $R :=$ **tuple-new** Φ. Thus o appears in R^{I_k} as a new object associated to a tuple t in $\Phi(I_{k-1})$. We define

   $$f_{I,J}(o) := \text{pair}\Big(\text{tuple}(f_{I,J}(t)), \text{number}(k)\Big).$$

   In the right-hand side formula,

- *pair* is the well-known Kuratowski encoding of ordered pairs as hereditarily finite sets defined by $\text{pair}(x, y) := \{\{x\}, \{x, y\}\}$;
- *tuple* is the well-known encoding of finite sequences (i.e., tuples) as hereditarily finite sets defined by

$$\begin{aligned} \text{tuple}() &= \emptyset; \\ \text{tuple}(x) &= \{x\}; \\ \text{tuple}(x_1, \ldots, x_n) &= \text{pair}(x_1, \text{tuple}(x_2, \ldots, x_n))), \qquad \text{for } n \geq 2; \end{aligned}$$

- *number* is the encoding of natural numbers as hereditarily finite sets in $HF(\emptyset)$ defined by $\text{number}(0) = \emptyset$ and $\text{number}(n+1) = \{\text{number}(n)\}$.

2. The instance I_k results from I_{k-1} by a set-new statement of the form $R :=$ **set-new** Φ. Thus $\Phi(I_{k-1})$ is a binary relation, which can be viewed as the set-valued function

$$s : x \mapsto \{y \mid (x, y) \in \Phi(I_{k-1})\}.$$

There is an x such that the pair (x, o) is in R^{I_k}. We define

$$f_{I,J}(o) := \text{pair}\left(f_{I,J}(s(x)), \text{number}(k)\right).$$

By the very definition of the set-new statement, the set $s(x)$ is independent of the particular choice of x. Hence, $f_{I,J}(o)$ is well-defined.

Notice that $f_{I,J}$ is the identity on $|I|$.

We now define the HF-transformation Q'. Let I be an instance. If there is an instance J such that $Q(I, J)$, then we define $Q'(I) := f_{I,J}(J)$. By the determinacy of Q and Property $(*)$, the HF-instance $f_{I,J}(J)$ does not depend on the particular choice of J. If there is no instance J such that $Q(I, J)$, then Q' is undefined on I. By Property $(*)$, Q' is invariant under permutations of **U**. Moreover, Q' is partial-recursive. To see this, let I be an instance. Since Q is recursively enumerable, it is possible to find an instance J with $Q(I, J)$ if such an instance exists. Next, all FO + **new** + **while** programs are applied to I in a dove-tailed fashion until one stops and yields J;[6] by

[6]Since in practice only one, representative, instance is computed, it must be verified that the result found is I-isomorphic to J.

definition, this program is $P_{I,J}$. Since $P_{I,J}$ can be effectively computed, $f_{I,J}$ is recursive. Consequently, $Q'(I) = f_{I,J}(J)$ can be computed. Thus Q' is an HF-transformation.

Obviously, Q is the isomorphic representation of Q'. Therefore, Q is constructive. ■

Theorem 3.13 thus gives an intrinsic characterization of the class of constructive transformations. As announced in the Introduction, we will show later that the transformations expressible in FO + **new** + **while** are precisely the constructive transformations. Theorem 3.13 therefore also shows that the characterization for the existence of a constructive transformation on the local level of individual input-output pairs, given by the existence of an extension homomorphism (Proposition 3.4), can be "lifted" to the global level of transformations.

4 List-constructive transformations

In the previous section, we proposed a general notion of constructive object creation in terms of hereditarily finite sets. In practice however, data structures are often implemented using *lists* rather than sets. This is for example the case in functional programming [1], as well as in logic programming [29] where first-order term structures are used for this purpose. Approaches to object creation as first-order term construction have been considered in the literature [25, 26].

In order to characterize the expressive power of languages that base object creation on lists rather than sets, we propose the notion of list-constructive transformation in this section. The development of this notion is analogous to the development of constructive transformation in the previous section. As we will see, list-constructive transformations are a special case of constructive transformations.

We first define hereditarily finite lists:

Definition 4.1 *Let D be a subset of* $\mathbf{U}$. *The set $HFl(D)$ of* hereditarily finite lists with ur-elements in D *is the smallest set such that* (i) $D \subseteq HFl(D)$; *and* (ii) *each finite list* $(\lambda_1, \ldots, \lambda_n)$ *of elements of $HFl(D)$ is also an element of $HFl(D)$.*

Lists, viewed as tuples of the appropriate rank, can be interpreted as hereditarily finite sets, by the Kuratowski encoding used in the proof of Theorem 3.13. Therefore hereditarily finite lists are a special case of hereditarily finite sets.

We denote by FO + **tuple-new** + **while** the sublanguage of FO + **new** + **while** consisting of those programs that do not use set-new statements. Andries and Paredaens proved the following analog of Proposition 3.4 for FO + **tuple-new** + **while**:

Proposition 4.2 [8] *Let I and J be instances with $|I| \subseteq |J|$. There exists an* FO + **tuple-new** + **while** *program P with $P(I, J)$, if and only if there exists* (a) *an extension homomorphism ψ from I to J and* (b) *an injective mapping $g : |J| \rightarrow HFl(|I|)$ which is the identity on $|I|$ satisfying*

$$g(\psi(f)(o)) = f(g(o))$$

for each f in $\mathrm{Aut}(I)$ *and each o in $|J|$, where f is extended to $HFl(|I|)$ in the standard way.*

In Example 3.6, Proposition 3.4 was used to show that there are determinate transformations not expressible in FO + **new** + **while**. Similarly, Proposition 4.2 can be used to show that there are transformations expressible in FO + **new** + **while** not expressible in FO + **tuple-new** + **while**:

Example 4.3 Let R be a binary relation name, and consider the transformation Q from the empty database scheme $\emptyset$ to $\{R\}$ defined as follows. Let I be an instance of $\emptyset$ (whence the only informative component of I is its domain $|I|$). Associate to each set p of two elements of $|I|$ a unique new object o_p not in $|I|$. Let $\mathcal{P}$ be the set of all these new objects. Then $Q(I, J)$ if $|J| = |I| \cup \mathcal{P}$ and $R^J = \{(x, o_p) \mid x \in p \text{ and } p \in \mathcal{P}\}$. A concrete pair (I, J) satisfying $Q(I, J)$ is shown in Figure 11. There exists a unique extension homomorphism ψ from I to J; however, there is no injective mapping g from $|J|$ into $HFl(|I|)$ satisfying Proposition 4.2 (for a proof see [31]). Hence, Q is not expressible in FO + **tuple-new** + **while**. Note that Q *is* expressible in FO + **new** + **while** by the following program:

$R_1 :=$ **tuple-new** $\{(x, y) \mid x \neq y\}$;
$R_2 := \{(z, x) \mid (\exists y)(R_1(x, y, z) \vee R_1(y, x, z))\}$;
$R_3 :=$ **set-new** $\{(z, x) \mid R_2(z, x)\}$;
$R := \{(x, o) \mid (\exists z)(R_2(z, x) \wedge R_3(z, o))\}$. ■

$\|I\|$
1
2
3
4

R^J

1	$o_{\{1,2\}}$
2	$o_{\{1,2\}}$
1	$o_{\{1,3\}}$
3	$o_{\{1,3\}}$
⋮	
3	$o_{\{3,4\}}$
4	$o_{\{3,4\}}$

Figure 11: Example of the unordered pair transformation of Example 4.3.

$\|I\|$
a
(a, a)
(b)
$(b, (c))$
(a, b, c)

R_1^I

a	$(b, (c))$
(b)	(a, b, c)

R_2^I

a	$(b, (c))$
(b)	(c, b, a)

Figure 12: Example of an HFl-instance.

The above example also shows that the **set-new** operator is a primitive construct in the language FO + **new** + **while** and cannot be simulated using the other constructs, as already mentioned in Section 2. For a detailed study of the expressive power of **set-new** we refer to [31].

In analogy with the previous section, we will define list-constructive transformations as transformations for which the new domain elements in the output can be viewed as hereditarily finite lists constructed over the domain elements of the input. Thereto, we need to define HFl-instances and HFl-transformations first.

An *HFl-instance* I is defined as an HF-instance, the only difference being that the domain $|I|$ is a subset of $HFl(\mathbf{U})$ instead of $HF(\mathbf{U})$. Relations and tuples on $HFl(\mathbf{U})$ are called HFl-relations and HFl-tuples, respectively. The set of all HFl-instances of a scheme $\mathcal{S}$ is denoted by HFl-inst($\mathcal{S}$). A simple example of an HFl-instance I of the scheme $\{R_1, R_2\}$, with R_1 and R_2 binary, is shown in Figure 12, where a, b, and c are elements of $\mathbf{U}$. Note that R_1^I and R_2^I are different.

Definition 4.4 *Let* $\mathcal{S}_{\text{in}}$ *and* $\mathcal{S}_{\text{out}}$ *be two schemes. An* HFl-transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ *is a partial recursive function* $Q : \text{inst}(\mathcal{S}_{\text{in}}) \to \text{HFl-inst}(\mathcal{S}_{\text{out}})$ *which* (i) *(viewed as a binary relationship) is invariant under every permutation of* $\mathbf{U}$ *and* (ii) *satisfies* $|Q(I)| \subseteq \mathit{HFl}(|I|)$ *whenever* Q *is defined on* I.

We can now define list-constructive transformations formally as follows:

Definition 4.5 *A transformation* Q *from* $\mathcal{S}_{\text{in}}$ *to* $\mathcal{S}_{\text{out}}$ *is called* list-constructive *if there exists an HFl-transformation* Q' *from* $\mathcal{S}_{\text{in}}$ *to* $\mathcal{S}_{\text{out}}$ *such that, for each pair of instances* (I, J), $Q(I, J)$ *if and only if* (i) $Q'(I)$ *is defined; and* (ii) J *is* I*-isomorphic to* $Q'(I)$.

As with constructive transformations, we will call Q the *isomorphic representation of* Q', and for any input-output pair (I, J) of Q, we will call instance J an *isomorphic representation* of HFl-instance $Q'(I)$.

List-constructive transformations are a special case of constructive transformations. In analogy with Theorem 3.13, we have the following intrinsic characterization of list-constructive transformations:

Theorem 4.6 *Let* Q *be a transformation. Then* Q *is list-constructive if and only if* (i) Q *is determinate; and* (ii) *for each pair of instances* (I, J) *with* $Q(I, J)$ *there exists* (a) *an extension homomorphism* ψ *from* I *to* J *and* (b) *an injective mapping* $g : |J| \to \mathit{HFl}(|I|)$ *which is the identity on* $|I|$ *satisfying*

$$g(\psi(f)(o)) = f(g(o))$$

for each f *in* $\text{Aut}(I)$ *and each* o *in* $|J|$*, where* f *is extended to* $\mathit{HFl}(|I|)$ *in the standard way.*

Proof. *Only if.* Since Q is list-constructive and therefore constructive, it follows from Proposition 3.12 that Q is determinate. Let Q' be an HFl-transformation isomorphically represented by Q. Assume $Q(I, J)$. Since J is an isomorphic representation of $Q'(I)$, there exists an I-isomorphism g from J to $Q'(I)$. Since $|Q'(I)| \subseteq \mathit{HFl}(|I|)$, g is an injective mapping from $|J|$ into $\mathit{HFl}(|I|)$. As in the proof of Proposition 3.12, $\psi : \text{Aut}(I) \to \text{Aut}(J) : f \mapsto g^{-1} \circ f \circ g$ can be shown to be an extension homomorphism from I to J. By construction, $g(\psi(f)(o)) = f(g(o))$ for each $f \in \text{Aut}(I)$ and each $o \in |J|$.

If. The proof of this implication is analogous to the proof of the if-implication of Theorem 3.13. For each pair of instances (I, J) with $Q(I, J)$ there exists a program P in FO + **tuple-new** + **while** with $P(I, J)$ (Proposition 4.2). Let $P_{I,J}$ be the first such program in some standard recursive enumeration of all programs. By the genericity of FO + **tuple-new** + **while** programs, we have

$$P_{I,J} = P_{f(I),f(J)}, \qquad \text{for each permutation } f \text{ of } \mathbf{U}. \qquad (*)$$

The execution of $P_{I,J}$ on I traces a finite sequence of FO statements and tuple-new statements. Let $\ell_{I,J}$ be the length of that sequence. For an integer k, $0 \leq k \leq \ell_{I,J}$, let I_k be the intermediate result of the program $P_{I,J}$ applied on I after the execution of the k-the statement in the sequence. Notice that $I_0 = I$, $I_{\ell_{I,J}} = J$, and $|I| = |I_0| \subseteq |I_1| \subseteq \cdots \subseteq |I_{\ell_{I,J}}| = |J|$. We construct an injective mapping $f_{I,J} : |J| \to \mathit{HFl}(|I|)$ recursively as follows.

Let $o \in |J|$. If $o \in |I_0| = |I|$, we define $f_{I,J}(o) := o$. Now assume that for some k, $0 < k \leq \ell_{I,J}$, $f_{I,J}$ has been defined on all objects in $|I_{k-1}|$. If $o \in |I_k| - |I_{k-1}|$, then I_k results from I_{k-1} by a tuple-new statement of the form $R :=$ **tuple-new** Φ. Thus o appears in R^{I_k} as a new object associated to a tuple t in $\Phi(I_{k-1})$. We define

$$f_{I,J}(o) := \left(f_{I,J}(t), \text{number}(k)\right),$$

where *number* is the encoding of natural numbers as hereditarily finite lists in $\mathit{HFl}(\emptyset)$ defined by number$(0) = (\,)$ and number$(n+1) = (\text{number}(n))$.

Notice that $f_{I,J}$ is the identity on $|I|$.

We now define the HFl-transformation Q'. Let I be an instance. If there is an instance J such that $Q(I, J)$, then we define $Q'(I) := f_{I,J}(J)$. If there is no instance J such that $Q(I, J)$, then Q' is undefined on I. By an argumentation analogous to the one used in the proof of Theorem 3.13, we can show that Q' is a well-defined HFl-transformation that is isomorphically represented by Q, whence Q is list-constructive. ∎

Since we will show in the next section that the transformations expressible in FO + **tuple-new** + **while** are precisely the list-constructive transformations, the above theorem shows that Proposition 4.2 can be "lifted" from the local level of individual input-output pairs to the global level of transformations.

5 Completeness results for list-constructive transformations

In this section, we prove that the language FO + **tuple-new** + **while** expresses precisely the list-constructive transformations. This result, already important in its own right, will enable us to prove that the language FO + **new** + **while** expresses precisely the constructive transformations.

To simplify the presentation of rather complicated programs that will be discussed in this section, we will occasionally decompose programs in subprograms (procedures) with the usual Pascal-like sytax and semantics.

We prove the completeness of FO + **tuple-new** + **while** for the list-constructive transformations by a reduction to the seminal completeness result of Chandra and Harel [12]. Chandra and Harel studied the computation of *unranked databases*.

An unranked instance I of a database scheme $\mathcal{S}$ is defined as an ordinary instance, the only difference being that the arities $\alpha(R)$ of the relation names R in $\mathcal{S}$ are ignored. So, for each R, R^I is a relation on $|I|$ not necessarily of rank $\alpha(R)$; the rank of the content of R can vary from instance to instance. The set of all unranked instances of a scheme $\mathcal{S}$ is denoted by UnR-inst($\mathcal{S}$).

The computation of unranked databases from ordinary ones is formalized by the notion of *unranked transformation*:

Definition 5.1 *Let $\mathcal{S}_{\text{in}}$ and $\mathcal{S}_{\text{out}}$ be two schemes. An* unranked transformation *from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ is a partial-recursive function $Q : \text{inst}(\mathcal{S}_{\text{in}}) \rightarrow \text{UnR-inst}(\mathcal{S}_{\text{out}})$ which* (i) *(viewed as a binary relationship) is invariant under every permutation of* $\mathbf{U}$ *and* (ii) *satisfies $|Q(I)| = |I|$ whenever Q is defined on I.*

Example 5.2 For arbitrary relation names R and T, consider the function $Q : \text{inst}(\{R\}) \rightarrow \text{UnR-inst}(\{T\})$ defined by

$$T^{Q(I)} = \underbrace{R^I \times \cdots \times R^I}_{(n \text{ times})},$$

where n is the cardinality of R^I. The function Q is an unranked transformation from $\{R\}$ to $\{T\}$. The rank of the content of T in $Q(I)$ depends on I.

Chandra and Harel also introduced a powerful language for expressing unranked transformations, called QL, and described below.[7]

Let $\mathcal{S}$ be a scheme. A QL program over $\mathcal{S}$ is built from statements over $\mathcal{S}$ using composition and while-loops. A *statement* over $\mathcal{S}$ is of the form $X := \tau$, with X a *variable* and τ a term over $\mathcal{S}$. *Terms* over $\mathcal{S}$ are defined as follows: *(1)* D is a term; *(2)* E is a term; *(3)* a relation name of $\mathcal{S}$ is a term; and *(4)* if Y and Z are variables, then Y, $\downarrow(Y)$, $\uparrow(Y)$, $\tilde{}(Y)$, $(Y \cup Z)$, and $(Y - Z)$ are terms. *While-loops*, finally, are of the form **while** $X = \emptyset$ **do** P **od** with X a variable and P a program.

Semantically, a statement $X := \tau$ assigns to the variable X the relation that is the interpretation of the term τ. More precisely, given an instance I of $\mathcal{S}$,

(1) the term D is interpreted as $|I|$ viewed as a unary relation;

(2) the term E is interpreted as the binary equality relation on $|I|$;

(3) a relation name R is interpreted as R^I; and

(4) if relations r and s are assigned to variables Y and Z, respectively, then

- Y is interpreted as r;
- $\downarrow(Y)$ as $\downarrow(r) := \{(x_2, \ldots, x_k) \mid \exists x_1 \in |I| : (x_1, x_2, \ldots, x_k) \in r\}$;
- $\uparrow(Y)$ as $\uparrow(r) := r \times |I|$;
- $\tilde{}(Y)$ as $\tilde{}(r) := \{(x_1, \ldots, x_{k-2}, x_k, x_{k-1}) \mid (x_1, \ldots, x_k) \in r\}$;
- $(Y \cup Z)$ as $r \cup s$; and
- $(Y - Z)$ as $r - s$.

Note that variables are untyped in that they can take relations of any rank as values. The semantics of a QL program over $\mathcal{S}$ is now obvious. The program expresses an unranked transformation from $\mathcal{S}$ to some output scheme $\mathcal{S}_{\text{out}}$ by designating for each relation name R in $\mathcal{S}_{\text{out}}$ an associated output variable X_R.

Chandra and Harel proved the following seminal completeness result:

[7]For technical convenience, we replaced the complement operator by the difference operator, which does not alter the expressiveness of the language.

Lemma 5.3 [12] *QL expresses precisely the unranked transformations.*

In order to be able to use the above lemma to prove that FO + **tuple-new+while** expresses precisely all list-constructive transformations, we must establish a link between the setting of Chandra and Harel and ours. Thereto, we make three basic observations:

1. A relation is a set of tuples of equal length.
2. A tuple can alternatively be seen as a list.
3. Let J be an instance of a scheme containing the binary relation names *Head* and *Tail*. For any unary relation name V also in the scheme, we can interpret the set V^J as a collection of lists, using the binary relations Head^J and Tail^J as in Example 2.21.

These observations lead us to define:

Definition 5.4 *Let $\mathcal{S}$ be a scheme and let I be an unranked instance of $\mathcal{S}$. Let J be an ordinary instance of the scheme* $\text{list}(\mathcal{S}) := \{\mathit{Head}, \mathit{Tail}\} \cup \{\hat{R} \mid R \in \mathcal{S}\}$*, with $\alpha(\hat{R}) = 1$ for all $R \in \mathcal{S}$. We call J a* list representation of I *if $|J| \supset |I|$ and for each $R \in \mathcal{S}$, the lists of $\hat{R}^J$ are precisely the tuples of R^I.*

Definition 5.5 *Let Q be an unranked transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$. An ordinary transformation Q' from $\mathcal{S}_{\text{in}}$ to* $\text{list}(\mathcal{S}_{\text{out}})$ *is called a* list representation of Q *if, for each instance I of $\mathcal{S}_{\text{in}}$, $Q(I)$ is defined if and only if $Q'(I, J)$ for some J, and in that case J is a list representation of $Q(I)$.*

We now prove that FO+**tuple-new**+**while** can simulate QL. More precisely:

Lemma 5.6 *Every unranked transformation can be list-represented by a program in* FO + **tuple-new** + **while**.

Proof. Assume an unranked transformation expressed by some QL program P is given. We shall explain by an inductive argument how P is translated into an FO + **tuple-new** + **while** program P' which list-represents P. The first two lines of P' are

$D := \{(x) \mid \textbf{true}\};$
$E := \{(x, y) \mid x = y\}.$

Given an input instance I, P' will compute in D^I and E^I the interpretation of the QL-terms D and E, respectively.

For each variable X in P, P' will introduce a unary relation name $\hat{X}$. During the execution of P', the value of $\hat{X}$ will be a collection of lists representing the corresponding value of X during the execution of P. To represent these lists, P' also introduces the binary relation names *Head* and *Tail* which are initialized by

$Head := \emptyset;$
$Tail := \emptyset.$

For convencience, P' also introduces binary relation names *Equallist* and $Tail^*$. When invoked, the procedure *Comp-Equallist* shown in Figure 13 computes in relation *Equallist* all pairs of list objects introduced thus far representing equal lists. Similarly, the procedure *Comp-*$Tail^*$ shown in Figure 14 computes in relation $Tail^*$ the reflexive-transitive closure of the current value of the *Tail* relation. Finally, if R and S are unary relation names and the content of R can be interpreted as a collection of lists, then the procedure $Copy(R; \textbf{var } S)$ shown in Figure 15 computes in relation S a copy of R using a set of new list objects.

Now consider a QL statement of the form $X := D$ or $X := E$ with X a variable. Due to the first two lines of P', the QL terms D and E can alternatively be interpreted as relation names. Thus consider a QL statement of the form $X := R$ with X a variable and R a relation name (which is either D, E, or an element of the input scheme $\mathcal{S}$). The following FO + **tuple-new** + **while** statements simulate $X := R$ in the case that R is binary; from this, the general case immediately follows.

$R_1 := \textbf{tuple-new } R;$
$R_2 := \textbf{tuple-new } R_1;$
$Head := Head \cup \{(l_1, x) \mid (\exists y)(\exists l_2) R_2(x, y, l_1, l_2)\};$
$Tail := Tail \cup \{(l_1, l_2) \mid (\exists x)(\exists y) R_2(x, y, l_1, l_2)\};$
$Head := Head \cup \{(l_2, y) \mid (\exists x)(\exists l_1) R_2(x, y, l_1, l_2)\};$
$\hat{X} := \{(l_1) \mid (\exists x)(\exists y)(\exists l_2) S_2(x, y, l_1, l_2)\}.$

To simulate QL statements of the form $X := Y$, $X := \downarrow(Y)$, $X := \uparrow(Y)$, $X := \tilde{}(Y)$, $X := (Y \cup Z)$, and $X := (Y - Z)$, with X, Y, and Z variables, we first observe that we can assume without loss of generality that X, Y,

$Equallist := \{(l_1, l_2) \mid \textbf{true}\};$
$Next := \{(l_1, l_1, l_2, l_2) \mid Equallist(l_1, l_2)\};$
while $Next \neq \emptyset$ **do**

$$Equallist := Equallist - \{(l_1, l_2) \mid (\exists l_1')(\exists l_2')(\exists h_1)(\exists h_2)(Next(l_1, l_1', l_2, l_2') \land Head(l_1', h_1) \land Head(l_2', h_2) \land h_1 \neq h_2)\};$$

$$Equallist := Equallist - \{(l_1, l_2) \mid (\exists l_1')(\exists l_2')(Next(l_1, l_1', l_2, l_2') \land (\exists t)\, Tail(l_1', t) \land \neg(\exists t)\, Tail(l_2', t))\};$$

$$Equallist := Equallist - \{(l_1, l_2) \mid (\exists l_1')(\exists l_2')(Next(l_1, l_1', l_2, l_2') \land (\exists t)\, Tail(l_2', t) \land \neg(\exists t)\, Tail(l_1', t))\};$$

$$Next := \{(l_1, t_1, l_2, t_2) \mid Equallist(l_1, l_2) \land (\exists l_1')(\exists l_2')(Next(l_1, l_1', l_2, l_2') \land Tail(l_1', t_1) \land Tail(l_2', t_2))\}$$

od.

Figure 13: Procedure *Comp-Equallist*. Computes in *Equallist* all pairs of list objects representing equal lists.

$Prev := \emptyset;$
$Tail^* := \{(l, l') \mid l = l'\};$
while $Tail^* - Prev \neq \emptyset$ **do**

$$Prev := Tail^*;$$

$$Tail^* := Tail^* \cup \{(l, t) \mid (\exists l')(Tail^*(l, l') \land Tail(l', t))\}$$

od.

Figure 14: Procedure *Comp-Tail**. Computes in $Tail^*$ the reflexive-transitive closure of *Tail*.

$Comp\text{-}Tail^*$;
$R' :=$ **tuple-new** $\{(l') \mid (\exists l)(R(l) \wedge Tail^*(l,l'))\}$;
$Head := Head \cup \{(l',h) \mid (\exists l)(Head(l,h) \wedge R'(l,l'))\}$;
$Tail := Tail \cup \{(l',t') \mid (\exists l)(\exists t)(Tail(l,t) \wedge R'(l,l') \wedge R'(t,t'))\}$;
$S := \{(l') \mid (\exists l)(R(l) \wedge R'(l,l'))\}$.

Figure 15: Procedure $Copy(R; \mathbf{var}\ S)$. Computes in S a copy of R using a set of new list objects.

---

and Z are all different. (By introducing auxiliary variables, the program P can indeed be rewritten to meet this condition.)

The statement $X := Y$ is simulated by $Copy(\hat{Y}, \hat{X})$.

The statement $X := \downarrow(Y)$ is simulated by

$Copy(\hat{Y}, \hat{X})$;
$\hat{X} := \{(t) \mid (\exists l)(\hat{X}(l) \wedge Tail(l,t))\}$.

In case the current value of Y contains different tuples with the same first component, the above simulation gives rise to *duplicates* in the collection of lists $\hat{X}$, i.e., to different objects in $\hat{X}$ representing the same list. The possible presence of duplicates is harmless and is not prohibited by Definition 5.4.

The statement $X := \uparrow(Y)$ is simulated by

$Comp\text{-}Tail^*$;
$Y' := \{(l') \mid (\exists l)(\hat{Y}(l) \wedge Tail^*(l,l'))\}$;
$Newlists :=$ **tuple-new** $\{(l,x) \mid Y'(l) \wedge D(x)\}$;
$\hat{X} := \{(l') \mid (\exists l)(\exists x)(Newlists(l,x,l') \wedge \hat{Y}(l))\}$;
$Head := Head \cup \{(l',h) \mid (\exists l)(\exists x) Newlists(l,x,l')\}$;
$Tail := Tail \cup$
$\qquad \{(l',t') \mid (\exists l)(\exists t)(\exists x)(Newlists(l,x,l') \wedge Newlists(t,x,t') \wedge Tail(l,t))\}$;
$Ends :=$ **tuple-new** $\{(l') \mid (\exists l)(\exists x) Newlists(l,x,l') \wedge \neg(\exists h) Head(l',h)\}$;
$Head := Head \cup \{(l',x) \mid (\exists l) Newlists(l,x,l') \wedge (\exists l'') Ends(l',l'')\}$;
$Tail := Tail \cup Ends$.

The statement $X := \tilde{}(Y)$ is simulated by

$Copy(\hat{Y}, \hat{X})$;
$Length\text{-}2 := \{(l_1, l_2, l_3) \mid \hat{X}(l_1) \wedge Tail(l_1, l_2) \wedge Tail(l_2, l_3) \wedge \neg(\exists h)Head(l_3, h)\}$;
if $Length\text{-}2 \neq \emptyset$ **then**
  $\hat{X} := \{(l_2) \mid (\exists l_1)(\exists l_3)Length\text{-}2(l_1, l_2, l_3)\}$;
  $Tail := \{(l, t) \mid Tail(l, t) \wedge$
  $\neg(\exists l_1)(\exists l_2)(\exists l_3)(Length\text{-}2(l_1, l_2, l_3) \wedge (l = l_1 \vee l = l_2))\}$;
  $Tail := Tail \cup \{(l, t) \mid (\exists l_1)(\exists l_2)(\exists l_3)(Length\text{-}2(l_1, l_2, l_3) \wedge$
  $((l, t) = (l_2, l_1) \vee (l, t) = (l_1, l_3)))\}$
else
  $Comp\text{-}Tail^*$;
  $Swap := \{(l_1, l_2, l_3, l_4) \mid (\exists l)(\hat{X}(l) \wedge Tail^*(l, l_1) \wedge Tail(l_1, l_2) \wedge Tail(l_2, l_3) \wedge$
  $Tail(l_3, l_4) \wedge \neg(\exists h)Head(l_4, h))\}$;
  $Tail := \{(l, t) \mid Tail(l, t) \wedge \neg(\exists l_1)(\exists l_2)(\exists l_3)(\exists l_4)(Swap(l_1, l_2, l_3, l_4) \wedge$
  $(l = l_1 \vee l = l_2 \vee l = l_3))\}$;
  $Tail := Tail \cup \{(l, t) \mid (\exists l_1)(\exists l_2)(\exists l_3)(\exists l_4)(Swap(l_1, l_2, l_3, l_4) \wedge$
  $((l, t) = (l_1, l_3) \vee (l, t) = (l_3, l_2) \vee (l, t) = (l_2, l_4)))\}$
fi.

In the above program, the relation *Length-2* is used to test for the special case when the rank of the current value of X is two. The if-then-else construct **if** $Length\text{-}2 \neq \emptyset$ **then** P_1 **else** P_2 can be simulated by the two if-then constructs **if** $Length\text{-}2 \neq \emptyset$ **then** P_1 and **if** $Length\text{-}2 = \emptyset$ **then** P_2, which were shown to be expressible in FO + **tuple-new** + **while** in Example 2.17.

The statement $X := (Y \cup Z)$ is simulated by

$Union := \hat{Y} \cup \hat{Z}$;
$Copy(Union, \hat{X})$.

The statement $X := (Y - Z)$ is simulated by

$Comp\text{-}Equallist$;
$Diff := \{(l_1) \mid \hat{Y}(l_1) \wedge \neg(\exists l_2)(\hat{Z}(l_2) \wedge Equallist(l_1, l_2))\}$;
$Copy(Diff, \hat{X})$.

Finally, a QL while-loop **while** $X = \emptyset$ **do** P_1 **od** with P_1 a QL program over $\mathcal{S}$ is simulated by the FO + **tuple-new** + **while** while-loop **while** $\hat{X} = \emptyset$ **do** P_1' **od** with P_1' the simulation of P_1. ■

Note that $\text{inst}(\mathcal{S}) \subseteq \text{UnR-inst}(\mathcal{S})$: ordinary instances are special unranked instances in which the ranks of the relations *do* conform to the arities of the relation names. Hence, ordinary transformations (Definition 2.10) which are deterministic (i.e., functions) are special unranked transformations which always yield ordinary instances as output. With this remark in mind we can prove the following useful corollary of Lemma 5.6.

Corollary 5.7 *Each deterministic transformation is expressible in* FO + **tuple-new** + **while**.

Proof. By Lemma 5.3, each deterministic transformation is expressible in QL. Lemma 5.6 thus yields that each deterministic transformation can be list-represented in FO + **tuple-new** + **while**.

It therefore suffices to show that there exists a program in FO + **tuple-new** + **while** which, given a list representation of an instance I of some fixed scheme $\mathcal{S}$ as input, produces I itself as output. This program consists of one statement for each $R \in \mathcal{S}$. The following statement is for the case $\alpha(R) = 2$; the other arities are treated analogously.

$$R := \{(x, y) \mid (\exists l)(\exists t)(\hat{R}(l) \wedge \mathit{Head}(l, x) \wedge \mathit{Tail}(l, t) \wedge \mathit{Head}(t, y))\}.$$

■

The completeness of QL for the unranked transformations, and the ability of FO + **tuple-new** + **while** to simulate QL, can be exploited to prove the completeness of FO + **tuple-new** + **while** for the list-constructive transformations. Since list-constructive transformations are defined in terms of HFl-transformations, we thereto need an encoding of HFl-instances as unranked instances, which we first describe.

Let K be an HFl-instance of some scheme $\mathcal{S}$. Denote the set of atomic objects (elements of $\mathbf{U}$) appearing in $|K|$ by $\mathbf{U}_K$. Let '`[`' (left bracket), '`]`' (right bracket) and '`_`' (blank) be three symbols in $\mathbf{U}$ not in $\mathbf{U}_K$. With each HFl-tuple t on $|K|$ we can associate a flat tuple $[t]$ on $\mathbf{U}_K \cup \{\texttt{[}, \texttt{]}\}$ by using the bracket symbols to mark begin and end of subtuples. For example, if t is the ternary HFl-tuple $(a, (b, c, (a)), b)$, then $[t]$ is the 9-ary flat tuple $(a, \texttt{[}, b, c, \texttt{[}, a, \texttt{]}, \texttt{]}, b)$.

Now consider a relation name $R \in \mathcal{S}$ and the HFl-relation R^K. Let n be the maximal arity of a tuple in $\{[t] \mid t \in R^K\}$. We can encode R^K as an n-ary

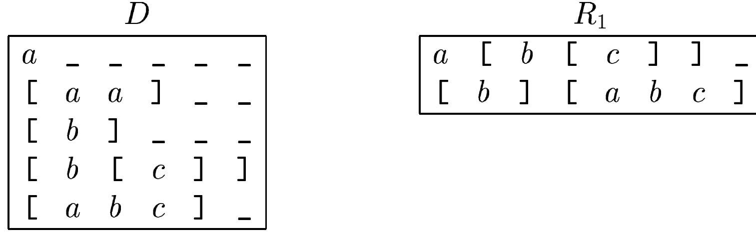


Figure 16: Flat encoding of the HFl-instance in Figure 12.

flat relation $[R^K]$ on $\mathbf{U}_K \cup \{[,],_\}$ by padding $[t]$ to the right with blanks if needed to bring the arity to n, for each $t \in R^K$. In the same way, we can also encode the domain $|K|$ of K as a flat relation $[|K|]$ by interpreting $|K|$ as a unary HFl-relation.

We can thus define the notion of *flat encoding* of an HFl-instance:

Definition 5.8 *Let K be an HFl-instance of some scheme $\mathcal{S}$, and let J be an unranked instance of the scheme*

$$\text{flat}(\mathcal{S}) := \{\textit{Leftbracket}, \textit{Rightbracket}, \textit{Blank}\} \cup \{D\} \cup \mathcal{S}.$$

We call J a flat encoding *of K if the following conditions are satisfied:*

- $|J| = |K| \cup \{[,],_\}$;
- $\textit{Leftbracket}^J = \{([)\}$, $\textit{Rightbracket}^J = \{(])\}$, *and* $\textit{Blank}^J = \{(_)\}$;
- $D^J = [|K|]$;
- *For each* $R \in \mathcal{S}$, $R^J = [R^K]$.

Example 5.9 Recall the HFl-instance I shown in Figure 12. Figure 16 shows relations D and R_1 of a flat encoding of I.

A flat encoding J of an HFl-instance K of a scheme $\mathcal{S}$ is an unranked instance (of the scheme flat($\mathcal{S}$)). According to Definition 5.4, J can be list-represented by an ordinary instance J' of the scheme list(flat($\mathcal{S}$)). We will call J' a *flat-list representation of K*.

We now have two ways of representing an HFl-instance by an ordinary instance: the "flat-list representation" just defined, and the original "isomorphic representation" of Definition 4.5. The following technical lemma says that we can go from the former to the latter in FO + **tuple-new** + **while**.

Lemma 5.10 *Let $\mathcal{S}$ be a scheme. There exists a program P in* FO + **tuple-new** + **while** *expressing a transformation from* list(flat($\mathcal{S}$)) *to $\mathcal{S}$ which, given as input a flat-list representation of some HFl-instance K of $\mathcal{S}$, produces as output an instance which is $\mathbf{U}_K$-isomorphic to K.*

Proof. Consider the following algorithm:

Input: a flat-list representation J of an HFl-instance K of $\mathcal{S}$.

Output: an instance L containing a list representation of an instance M of such that M is $\mathbf{U}_K$-isomorphic to K.

Method:

1. Select all sublists occurring in the current instance which begin with the left bracket, end with the right bracket, and do not contain a bracket symbol in between. Let n be the maximal length of these sublists.

2. Generate a collection of all lists over the current domain of length at most n. No duplicates may be generated, i.e., each list generated must be represented by a unique object in the collection.

3. Replace each sublist selected in step 1 by its unique representative object generated in step 2.

4. Repeat the above steps until no sublist is any longer selected in step 1.

5. Truncate all lists appearing in the current instance having a tail consisting exclusively of blanks.

We first show how this algorithm can be implemented in FO + **tuple-new** + **while**.

1. We will reuse procedure *Comp-Tail** of Figure 14 and its associated binary relation *Tail**, already used in the proof of Lemma 5.6. Note that after invoking this procedure, each pair (l, l') in *Tail** identifies the begin and end of a sublist occurring in the current instance. So, we can formulate step 1 as a deterministic transformation, with a binary output relation named *Select* in which the sublists of *Tail** with the desired properties are collected. By Corollary 5.7, this step is then expressible in FO + **tuple-new** + **while**.

By identifying a natural number n with a relation containing a single n-ary tuple of, say, blanks, we can formulate the computation of the maximal length n of the sublists in *Select* as an unranked transformation. By Lemma 5.6, this unranked transformation can be list-represented in FO + **tuple-new** + **while**. We can thus obtain a unary relation named *Length* holding a list of length n.

2. The collection *Collect* of all lists up to length n is now generated as follows. We use procedure $Copy(R; \textbf{var } S)$ of Figure 15.

   $Dom := \{(x) \mid \textbf{true}\};$
   $Empty\text{-}list := \textbf{tuple-new } \{() \mid \textbf{true}\};$
   $Collect := Empty\text{-}list;$
   $Count := \{(z') \mid (\exists z)(Length(z) \wedge Tail(z, z'))\};$
   while $Count := \emptyset$ **do**
   $\quad Copy(Collect, Collect');$
   $\quad Next := \textbf{tuple-new } \{(l', x) \mid Collect'(l') \wedge Dom(x)\};$
   $\quad Tail := Tail \cup \{(l, t) \mid (\exists x) Next(l, t, x)\};$
   $\quad Head := Head \cup \{(l, x) \mid (\exists l') Next(l, l', x)\};$
   $\quad Collect := Collect \cup \{(l) \mid (\exists l')(\exists x) Next(l, l', x)\};$
   $\quad Count := \{(z') \mid (\exists z)(Count(z) \wedge Tail(z, z'))\};$
   od.

3. Step 3 is then programmed as follows. We use procedure *Comp-Equallist* of Figure 13 and its associated binary relation *Equallist*.

   $Keep\text{-}tails := \{(s_2, t) \mid (\exists s_1) Select(s_1, s_2) \wedge Tail(s_2, t)\};$
   $Tail := Tail - \{(s_2, t) \mid (\exists s_1) Select(s_1, s_2)\}$
   $\qquad \cup \{(s_2, e) \mid (\exists s_1) Select(s_1, s_2) \wedge Empty\text{-}list(e)\};$
   $Comp\text{-}Equallist;$
   $Head := Head - \{(s_1, h) \mid (\exists s_2) Select(s_1, s_2)\}$
   $\qquad \cup \{(s_1, r) \mid (\exists s_2) Select(s_1, s_2) \wedge Collect(r) \wedge Equallist(s_1, r)\};$
   $Tail := Tail - \{(s_1, t) \mid (\exists s_2) Select(s_1, s_2)\}$
   $\qquad \cup \{(s_1, t) \mid (\exists s_2)(Select(s_1, s_2) \wedge Keep\text{-}tails(s_2, t))\}.$

4. Iterating the above steps as specified in the algorithm yields:

Step 1;
while *Select* $\neq \emptyset$ **do**
  Step 2;
  Step 3;
  Step 1
od.

5. Finally, finding the lists to be truncated can be formulated as a deterministic transformation. By Corollary 5.7 this transformation is expressible in FO + **tuple-new** + **while**, so that we can obtain a unary relation *Trunc* holding these lists. The actual truncation is easy to perform:

$$\begin{aligned} \mathit{Tail} := \mathit{Tail} &- \{(l,t) \mid \mathit{Trunc}(l)\} \\ &\cup \{(l,e) \mid \mathit{Trunc}(l) \wedge \mathit{Empty\text{-}list}(e)\}. \end{aligned}$$

Denote the complete program implementing the algorithm by P.

Assume now that P is applied to an input instance J which is a flat-list representation of an HFl-instance K of $\mathcal{S}$. Let L be the output instance.

Fix an arbitrary $R \in \mathcal{S}$. We first verify that all lists in $\hat{R}^L$ have length $\alpha(R)$. Thereto, we establish the following loop invariant of the while-loop specified in item 4 above:

Each list l in $\hat{R}$ can be written as a concatenation $l_1 \dots l_{\alpha(R)} t$, where t is a (possibly empty) tail consisting exclusively of blanks, and each l_i, $1 \leq i \leq \alpha(R)$, is either:

(a) a singleton sublist (o) with $o \in \mathbf{U}_K$; or

(b) a singleton sublist (p) with p a list object generated in step 2 of the algorithm; or

(c) a sublist beginning with the left bracket and ending with the right bracket.

Upon entry of the while-loop, the invariant clearly holds since the input instance J is a flat-list representation of an HFl-instance of $\mathcal{S}$. In this case, sublists of type (b) do not yet occur. After each iteration of the while-loop, the invariant also holds since the only thing that can happen is that sublists

of type (c) are replaced by sublists of type (b). Hence, the invariant also holds upon exit of the while-loop. In this case, sublists of type (c) do no longer occur since the iteration condition of the while-loop is no longer true. In step 5 of the algorithm, the tail t is removed, so that in the output L, each list in $\hat{R}$, being a concatenation of $\alpha(R)$ length-one sublists, indeed has length $\alpha(R)$.

By the same reasoning, we can show that all lists in $\hat{D}^L$ have length one.

Fix again an arbitrary $R \in \mathcal{S}$. We next verify that for each list l in $\hat{R}^L$, every element of l is the element of some (singleton) list in $\hat{D}^L$. Thereto, we establish the following loop invariant of the while-loop in item 4:

> *For each list $l = l_1 \dots l_{\alpha(R)} t$ in $\hat{R}$, and for each $i = 1, \dots, \alpha(R)$, there is a list $l' = l_i t'$ in $\hat{D}$, where t' is a (possibly empty) tail consisting exclusively of blanks.*

That the above property is indeed an invariant can be proven by a similar reasoning as for the previous invariant. The invariant holds upon exit of the while-loop, after which step 5 of the algorithm removes the blank tails. Since we already know that at this point each l_i is a singleton, every element of l thus is indeed the element of some singleton list in $\hat{D}^L$.

The above allows us to define the following instance M of $\mathcal{S}$: $|M|$ is the set (unary relation) list-represented by $\hat{D}^L$, and for each $R \in \mathcal{S}$, R^M is the relation list-represented by $\hat{R}^L$. The output instance L of P thus contains a list representation of M, but not M itself. However, we can complete P with a final step which produces M from its list representation. How this can be done was already shown in the proof of Corollary 5.7.

Hence, in order to conclude the proof of the lemma it is sufficient to prove that M and K are $\mathbf{U}_K$-isomorphic. Thereto, we inductively assign to each hereditarily finite list λ appearing in $|K|$ an identifying object $\mathrm{id}(\lambda) \in |L|$ as follows. As basis, we put $\mathrm{id}(o) := o$ for each $o \in \mathbf{U}_K$. Now let $\lambda = (\lambda_1, \dots, \lambda_m)$ be a hereditarily finite list appearing in $|K|$. By induction, we may assume that $\mathrm{id}(\lambda_i)$ is already known for $i = 1, \dots, m$. Consider the list

$$l = (\texttt{[}, \mathrm{id}(\lambda_1), \dots, \mathrm{id}(\lambda_m), \texttt{]}).$$

At some point during the execution of P, l will be replaced by a single representative object p in step 3 of the algorithm. Then define $\mathrm{id}(\lambda) := p$.

The mapping id just defined is clearly injective. Furthermore, $\mathrm{id}(|K|) = |M|$. Indeed, λ is in $|K|$ iff its flat encoding $[\lambda]$ is in $\hat{D}^J$ (represented as a list). In $\hat{D}^L$, this list will be replaced by $\mathrm{id}(\lambda)$. Similarly, we have $\mathrm{id}(R^K) = R^M$ for each $R \in \mathcal{S}$. Finally, the mapping id was defined to be the identity on $\mathbf{U}_K$. Hence, id is an $\mathbf{U}_K$-isomorphism from K to M. ■

We now have all the necessary ingredients together to prove our first completeness result:

Theorem 5.11 *Every list-constructive transformation is expressible in* FO+ **tuple-new** + **while**.

Proof. Let Q be a list-constructive transformation from $\mathcal{S}_{\mathrm{in}}$ to $\mathcal{S}_{\mathrm{out}}$. We must establish the existence of an FO + **tuple-new** + **while** program P expressing Q.

First, consider the following three-line program:

Leftbracket := **tuple-new** $\{() \mid$ **true**$\}$;
Rightbracket := **tuple-new** $\{() \mid$ **true**$\}$;
Blank := **tuple-new** $\{() \mid$ **true**$\}$.

Applied to an instance I of $\mathcal{S}_{\mathrm{in}}$, this program yields an instance I^+ of the scheme $\mathcal{S} \cup \{\mathit{Leftbracket}, \mathit{Rightbracket}, \mathit{Blank}\}$ which can be interpreted as the augmentation of I with three new unary relations, each consisting of a single object. These three objects are different, and are interpreted as the left bracket, right bracket, and blank symbol, respectively.

Next, we define an unranked transformation Q^{unr} from $\mathcal{S}_{\mathrm{in}} \cup \{\mathit{Leftbracket}, \mathit{Rightbracket}, \mathit{Blank}\}$ to $\mathrm{flat}(\mathcal{S}_{\mathrm{out}})$. Thereto, let Q' be an HFl-transformation isomorphically represented by Q in the sense of Definition 4.5. For an instance I of $\mathcal{S}_{\mathrm{in}}$, define $Q^{\mathrm{unr}}(I^+)$ as the flat encoding of $Q'(I)$ in which the left bracket, right bracket, and blank are those of I^+. By Lemma 5.6, Q^{unr} can be list-represented by an FO + **tuple-new** + **while** program. When applied to I^+, this program yields a flat-list representation of $Q'(I)$.

Finally, by Lemma 5.10, there exists an FO+**tuple-new**+**while** program transforming a flat-list representation of an HFl-instance K into an $\mathbf{U}_K$-isomorphic representation of that instance. Applying this program to the flat-list representation of $Q'(I)$ thus yields an instance J which is $\mathbf{U}_{Q'(I)}$-isomorphic to $Q'(I)$. But $\mathbf{U}_{Q'(I)} = |I|$, so that J is I-isomorphic to $Q'(I)$. Since Q is the isomorphic representation of Q', J satisfies $Q(I, J)$.

Hence, the composition of the above programs yields the desired FO + **tuple-new** + **while** program. ■

6 Completeness results for constructive transformations

In this section, we prove that the language FO + **new** + **while** expresses precisely the constructive transformations.

We will exploit the completeness of FO + **tuple-new** + **while** for the list-constructive transformations, established in the previous section, to establish the completeness of FO + **new** + **while** for the constructive transformations. Since list-constructive transformations are defined in terms of hereditarily finite lists, while constructive transformations are defined in terms of hereditarily finite sets, we thereto need an encoding of HF-instances as HFl-instances, which we first describe.

Let X be a hereditarily finite set. We can associate to X a set $\Lambda(X)$ of hereditarily finite lists, in the following inductive manner:

- For each object $o \in U$, $\Lambda(o) := \{o\}$.
- For each finite set $V = \{V_1, \ldots, V_n\}$ of hereditarily finite sets,

$$\Lambda(V) := \big\{(\lambda_{\pi(1)}, \ldots, \lambda_{\pi(n)}) \mid \pi \text{ a permutation of } \{1, \ldots, n\}, \\ \lambda_i \in \Lambda(V_i) \text{ for } i = 1, \ldots, n\big\}.$$

With an HF-tuple $t = (V_1, \ldots, V_k)$, we then associate the set $\Lambda(t)$ of all HFl-tuples $(\lambda_1, \ldots, \lambda_k)$ such that $\lambda_i \in \Lambda(X_i)$ for $i = 1, \ldots, k$. Furthermore, with an HF-relation r we associate the HFl-relation $\Lambda(r) := \bigcup_{t \in r} \Lambda(t)$. Finally, with an HF-instance K of some scheme $\mathcal{S}$ we associate the HFl-instance $\Lambda(K)$ of $\mathcal{S}$ with the same domain defined by $R^{\Lambda(K)} := \Lambda(R^K)$ for each $R \in \mathcal{S}$. As for HFl-instances, $\mathbf{U}_K$ stands for the set of atomic objects appearing in $|K|$.

The following technical lemma, which is the analogue of Lemma 5.10 for HF-instances, informally says that we can go from $\Lambda(K)$ to K in FO + **new** + **while**:

Lemma 6.1 *Let $\mathcal{S}$ be a scheme. There exists a program P in* $\mathrm{FO} + \mathbf{new} + \mathbf{while}$ *expressing a transformation from* $\mathrm{list}(\mathrm{flat}(\mathcal{S}))$ *to $\mathcal{S}$ which, given as input a flat-list representation of $\Lambda(K)$, for some HF-instance K of $\mathcal{S}$, produces as output an instance which is $\mathbf{U}_K$-isomorphic to K.*

Proof. By Lemma 5.10, there exists an $\mathrm{FO} + \mathbf{tuple\text{-}new} + \mathbf{while}$ program which produces from a flat-list representation of an HFl-instance an isomorphic representation of that instance. Let P_0 be the particular such program constructed in the proof of that lemma. When applied to a flat-list representation J of $\Lambda(K)$, for some HF-instance K of $\mathcal{S}$, P_0 not only produces an instance M of $\mathcal{S}$ which is $\mathbf{U}_K$-isomorphic to $\Lambda(K)$, but as a side effect also *Head* and *Tail* relations which describe the structure of the hereditarily finite lists represented in M.

We can now continue with M and produce an instance $\mathbf{U}_K$-isomorphic to K as follows. Each object appearing in $|M|$ either is an element of $\mathbf{U}_K$, or identifies a hereditarily finite list (which can be accessed through *Head* and *Tail* functions). Using a straightforward program, the subset $\mathbf{U}_K \cap |M|$ can be isolated in a unary relation variable which we give the same name $\mathbf{U}_K$. Each hereditarily finite list is accompanied by all its re-orderings, which together stand for a hereditarily finite set. It thus suffices to generate a unique new identifier for each equivalence class of orderings, using the **set-new** operator in a bottom-up fashion. After each stage of this bottom-up process, the identifiers of the considered lists are replaced by their new representative which then serves as the object representing the corresponding hereditarily finite set.

We now show how this can be formally accomplished in $\mathrm{FO}+\mathbf{new}+\mathbf{while}$. For notational simplicity, the scheme $\mathcal{S}$ is assumed to consist of only one, binary, relation name R.

To generate an identifier for the empty set we use the statement

$$\mathit{Empty} := \mathbf{tuple\text{-}new}\ \{() \mid \mathbf{true}\}.$$

To replace all empty lists by this identifier we use the statements

$$\begin{aligned}
&\mathit{E_list} := \{(l) \mid \neg \mathbf{U}_K(l) \land \neg(\exists h)\mathit{Head}(l,h)\};\\
&\mathit{Head} := \{(l,h) \mid \mathit{Head}(l,h) \land \neg \mathit{E_list}(h)\}\\
&\qquad\qquad \cup \{(l,e) \mid (\exists h)(\mathit{Head}(l,h) \land \mathit{E_list}(h)) \land \mathit{Empty}(e)\};\\
&\mathit{Tail} := \{(l,t) \mid \mathit{Tail}(l,t) \land \neg \mathit{E_list}(t)\}
\end{aligned}$$

$$\cup \{(l,e) \mid (\exists t)(Tail(l,t) \wedge E_list(t)) \wedge Empty(e)\};$$

$$\begin{aligned} R := \{(x',y') \mid (\exists x)(\exists y)(\exists e)(&R(x,y) \wedge Empty(e) \\ &\wedge \neg E_list(x) \rightarrow x' = x \\ &\wedge E_list(x) \rightarrow x' = e \\ &\wedge \neg E_list(y) \rightarrow y' = y \\ &\wedge E_list(y) \rightarrow y' = e)\}. \end{aligned}$$

The bottom-up replacement process is performed by a while-loop initiated as follows:

$Done := \mathbf{U}_K \cup Empty;$
while $\neg(\forall x)(\forall y)(R(x,y) \rightarrow Done(x) \wedge Done(y))$ **do**

In each iteration, we first compute the members of each list using the statements

$Comp\text{-}Tail^*$; (Figure 14)
$Contains := \{(l,x) \mid \neg Done(l) \wedge (\exists t)(Tail^*(l,t) \wedge Head(t,x))\};$
$Contains := \{(l,x) \mid Contains(l,x) \wedge (\forall y)(Contains(l,y) \rightarrow Done(y))\};$

Then the equivalence classes are factored out using the statements

$Equiv :=$ **set-new** $Contains;$
$Done := Done \cup \{(z) \mid (\exists l) Equiv(l,z)\};$

Finally, the identifiers are replaced using the statements

$$\begin{aligned} Head := \{(l',h') \mid (\exists l)(\exists h)(&Head(l,h) \\ &\wedge (\exists z)Equiv(l,z) \rightarrow Equiv(l,l') \\ &\wedge (\exists z)Equiv(h,z) \rightarrow Equiv(h,h'))\}; \end{aligned}$$

$$\begin{aligned} Tail := \{(l',t') \mid (\exists l)(\exists t)(&Tail(l,t) \\ &\wedge (\exists z)Equiv(l,z) \rightarrow Equiv(l,l') \\ &\wedge (\exists z)Equiv(t,z) \rightarrow Equiv(t,t'))\}; \end{aligned}$$

$$\begin{aligned} R := \{(x',y') \mid (\exists x)(\exists y)(&R(x,y) \\ &\wedge (\exists z)Equiv(x,z) \rightarrow Equiv(x,x') \\ &\wedge (\exists z)Equiv(y,z) \rightarrow Equiv(y,y'))\}. \end{aligned}$$

This concludes the body of the while-loop:

od. ■

We can now prove our main completeness result:

Theorem 6.2 *Every constructive transformation is expressible in* FO + **new** + **while**.

Proof. Let Q be a constructive transformation from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$. Let Q' be an HF-transformation isomorphically represented by Q according to Definition 3.11.

Define the HFl-transformation $\Lambda(Q')$ from $\mathcal{S}_{\text{in}}$ to $\mathcal{S}_{\text{out}}$ by the equation

$$\Lambda(Q')(I) = \Lambda(Q'(I)).$$

In the same way as in the proof of Theorem 5.11, we can find a program which produces from an input instance I of $\mathcal{S}_{\text{in}}$ a flat-list representation of $\Lambda(Q'(I))$. By applying Lemma 6.1, we can then obtain an instance J which is $\mathbf{U}_{Q'(I)}$-isomorphic to $Q'(I)$. But $\mathbf{U}_{Q'(I)} = |I|$, so that J is I-isomorphic to $Q'(I)$. Since Q is the isomorphic representation of Q', J satisfies $Q(I, J)$.

Hence, the composition of the above programs yields an FO+**new**+**while** program expressing Q. ■

The proofs of Theorem 6.2 and Lemma 6.1 show that, in order to express an arbitrary constructive transformation in FO + **new** + **while**, the **set-new** operation is only needed in a final "beautifying" stage. In this stage, the set objects needed in the output are obtained from collections of their orderings, represented as list objects, by making abstraction of the ordering information present in these lists. It is precisely this "abstraction" power that is provided by the **set-new** operation. All of the other computations needed for the constructive transformation can be performed on the level of lists rather than sets, and can be expressed in the sublanguage FO + **tuple-new** + **while** of FO + **new** + **while** without the **set-new** operation. The reader interested in a proof that **set-new** cannot be simulated using **tuple-new** is referred to [31].

7 Concluding remarks

Before we conclude with a few remarks on the ramifications of the results obtained in this paper, let us summarize them as follows:

1. Let Q be a transformation. The following are equivalent:

    - Q is constructive.
    - Q is determinate, and for each pair of instances (I, J) with $Q(I, J)$ there exists an extension homomorphism from I to J.
    - Q is expressible in $\mathrm{FO} + \textbf{new} + \textbf{while}$.

2. Let Q be a transformation. The following are equivalent:

    - Q is list-constructive.
    - Q is determinate, and for each pair of instances (I, J) with $Q(I, J)$ there exists *(a)* an extension homomorphism ψ from I to J and *(b)* an injective mapping $g : |J| \to \mathit{HFl}(|I|)$ which is the identity on $|I|$ satisfying
    $$g(\psi(f)(o)) = f(g(o))$$
    for each f in $Aut(I)$ and each o in $|J|$, where f is extended to $\mathit{HFl}(|I|)$ in the standard way.
    - Q is expressible in $\mathrm{FO} + \textbf{tuple-new} + \textbf{while}$.

There are various possible approaches to the representation of hereditarily finite set and list structures in database manipulation. One can support these structures directly as built-in data types of the system. One can use object creation to decompose the structures into atomic, interconnected units and thus work with a sort of graph representation. One can also use object creation in a different way by representing the hereditarily finite sets or lists appearing in the output of the manipulation by new abstract objects and thus obtaining a result formally isomorphic to the original one but ignoring the internal structure of the hereditarily finite sets and lists. Yet another approach is to encode the nested structures into flat lists which are unbounded in length (cf. the unranked relations of Chandra and Harel). In the course of proving our completeness results, we have shown that these various approaches are all interconnected and moreover possess precisely the same expressive power, namely that of the constructive and the list-constructive transformations.

The use of hereditarily finite sets in databases has been studied before by Hull and Su [23], and by Dahlhaus and Makowsky [17].

Hull and Su considered a language analogous to FO + **new** + **while**, but used it only for expressing domain-preserving, deterministic transformations. They showed that the construction of hereditarily finite sets in intermediate stages of the computation, while disallowing them to appear in the end result, can be used to achieve completeness, much as Abiteboul and Vianu did for object creation (see the Introduction). In this paper, we have shown that these two "completeness tools," hereditarily finite set construction and object creation, are in a natural sense equivalent. Furthermore, we have shown that Hull and Su's completeness result can be extended from deterministic transformations to HF-transformations in general. Interestingly, our completeness proofs rely exclusively on the first known completeness tool as originally proposed by Chandra and Harel, namely the unranked relations of QL. This makes the circle complete, since one of the main goals of the work of Abiteboul and Vianu was to propose object creation as an alternative for the use of unranked relations.

Dahlhaus and Makowsky defined *directory queries*, a model of queries on structures similar to HF-instances, as a generalization of the queries on flat instances of Chandra and Harel. They proposed a language, called DL, which they proved to be complete for the directory queries. Although Dahlhaus and Makowsky did not explicitly consider object creation, directory queries fully support the construction of hereditarily finite sets, to which they informally referred as "the computation of new objects from given objects." In this paper, we have demonstrated that this identification was justified and can be achieved in a very formal way.

A posteriori, it seems not unreasonable to argue in this respect that the original notion of determinate transformation proposed by Abiteboul and Kanellakis, although natural and obvious at first sight, does not adequately formalize deterministic object creation. As mentioned in the Introduction, making FO + **new** + **while** complete for the determinate transformations requires the introduction of a copy elimination operation. Van den Bussche and Van Gucht [32] obtained results indicating that the phenomenon of copy elimination can be more naturally explained in a non-deterministic context. Our results put this approach in perspective. Indeed, while in this paper we have *restricted* the determinacy criterion of Abiteboul and Kanellakis with an extension morphism condition, the authors of [32] proposed a *relaxation* of determinacy, called *semi-determinism*. From this perspective, determinacy might be merely an intermediate between constructivity on the one hand and

semi-determinism on the other hand.

But perhaps the main philosophical consequence of our results is a formal reconciliation of the two main approaches to creating new objects that have been considered in the literature on formal query languages for object-oriented database systems. The first approach recognizes created objects as new, atomic entities (e.g., [4, 20, 27]). We took this approach here. The second approach treats new objects in the output as terms, constructed from the objects in the input (e.g., [25, 26]). Since these terms can be interpreted as hereditarily finite lists, the results in this paper highlight the close link between both approaches.

References

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

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

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

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. 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. Also INRIA Rapport de Recherche 1022, 1989.

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

[6] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[7] 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.

[8] M. Andries and J. Paredaens. On instance-completeness of database query languages involving object creation. *Journal of Computer and System Sciences*, 52(2):357–373, 1996.

[9] F. Bancilhon. On the completeness of query languages for relational data bases. In *Proceedings 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1978.

[10] J. Barwise. *Admissible Sets and Structures*. Springer-Verlag, 1975.

[11] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990.

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

[13] E. Codd. A relational model for large shared databanks. *Communications of the ACM*, 13(6):377–387, 1970.

[14] E. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, pages 33–64. Prentice-Hall, 1972.

[15] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, 1972.

[16] E. Dahlhaus and J.A. Makowsky. The choice of programming primitives for SETL-like programming languages. In B. Robinet and R. Wilhelm, editors, *Proceedings ESOP'86*, Lecture Notes in Computer Science, vol. 213, pages 160–172. Springer-Verlag, 1986.

[17] E. Dahlhaus and J.A. Makowsky. Query languages for hierarchic databases. *Information and Computation*, 101(1):1–32, 1992.

[18] K. Denninghoff and V. Vianu. Database method schemas and object creation. In *Proceedings 12th ACM Symposium on Principles of Database Systems*, pages 265–275. ACM Press, 1993.

[19] H. Gaifman and M.Y. Vardi. A simple proof that connectivity is not first-order definable. *Bulletin of the EATCS*, 26:43–45, 1985.

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

[21] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 417–424. ACM Press, 1990.

[22] R. Fagin. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:89–96, 1975.

[23] R. Hull and J. Su. Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, 47(1):121–156, 1993.

[24] R. Hull and C.K. Yap. The format model, a theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[25] 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.

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

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

[28] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2), 1978.

[29] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.

[30] A. Tarski. What are logical notions? *History and Philosophy of Logic*, 7:143–154, 1986. Edited by J. Corcoran.

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

[32] J. Van den Bussche and D. Van Gucht. A semi-deterministic approach to object creation and non-determinism in database queries. *Journal of Computer and System Sciences*, to appear. Also *Proceedings 11th ACM Symposium on Principles of Database Systems*, pages 191–201.

[33] J. van Rossum. Master's thesis, Technical University of Eindhoven, 1992. (in Dutch).

A Appendix

In this appendix we show that FO + **new** + **while** has the same expressive power as the languages GOOD and IQL mentioned in the Introduction. We do this by showing that GOOD can be simulated in FO + **new** + **while** (Section A.1), that FO+**new**+**while** can be simulated in IQL (Section A.2), and that IQL can be simulated in GOOD (Section A.3).

In the GOOD data model, a database scheme is a directed, edge-labeled graph the nodes of which are class names and the edges of which represent relationships between classes. These relationships can be required to be functional. A database instance then is a directed labeled graph the nodes of which are objects, labeled by the name of their class, and the edges of which are relationships among the objects, labeled in accordance with the scheme. Programs in the GOOD language are sequences of graph transformation operations. There are five basic kinds of such operations. GOOD programs can also define and call procedures (called *methods*) which can be recursive.

The IQL data model is based on complex values built from atomic objects using the tuple and set constructors. A database scheme is described by a set of class names and a set of relation names. A complex value type is assigned to each class name and relation name. An instance then populates each class with a set of atomic objects, and each relation with a set of complex values of the correct type. Each atomic object is also assigned a complex value of the correct type. The language of IQL is rule-based.

A.1 From GOOD to FO + **new** + **while**

We assume the reader is familiar with GOOD as described in [20]; we will use the terminology and notation of that paper. We note one exception however:

we will ignore printable objects. The simulation of printable objects would require the introduction of constant symbols in the formalism of FO + **new** + **while**, which we have avoided for reasons of simplicity. It is well-known (e.g., [5]) how the use of constant symbols can be accounted for in the theory of database transformations.

We begin by representing GOOD schemes and instances by the relational schemes and instances of Section 2.

Let $\mathcal{S}$ be a GOOD scheme with set of object labels OL and set of productions $\mathcal{P}$. We define a relational scheme $\mathrm{rel}(\mathcal{S})$ representing $\mathcal{S}$ as follows. For each $K \in \mathrm{OL}$ we have a relation name K of arity 1. Furthermore, for each $p \in \mathcal{P}$ we have a relation name p of arity 2.

Now let $I = (\mathbf{N}, \mathbf{E})$ be an instance of the GOOD scheme $\mathcal{S}$. We define an instance of $\mathrm{rel}(\mathcal{S})$ representing I as follows. The domain equals $\mathbf{N}$. For each $K \in \mathrm{OL}$, the content of K equals $\{\mathbf{n} \in \mathbf{N} \mid \lambda(\mathbf{n}) = K\}$. For each $p = (K, \alpha, L) \in \mathcal{P}$, the content of p equals $\{(\mathbf{n}, \alpha, \mathbf{m}) \in \mathbf{E} \mid \lambda(\mathbf{n}) = K,\ \lambda(\mathbf{m}) = L\}$.

We next show how the set of matchings of a pattern in a GOOD instance can be expressed in FO + **new** + **while**. Let $J = (\mathbf{M}, \mathbf{F})$ be a pattern. We will use the elements of $\mathbf{M}$ as variables in first-order formulas. With each edge $\mathbf{f} = (\mathbf{m}, \alpha, \mathbf{n}) \in \mathbf{F}$ we associate the atomic formula $\mathrm{af}(\mathbf{f}) = (\lambda(\mathbf{m}), \alpha, \lambda(\mathbf{n}))(\mathbf{m}, \mathbf{n})$. For each list $\mathbf{m}_1, \ldots, \mathbf{m}_k$ of elements of $\mathbf{M}$ we then define the first-order formula $\mathrm{match}(J; \mathbf{m}_1, \ldots, \mathbf{m}_k)$ as $(\exists \mathbf{n}_1) \ldots (\exists \mathbf{n}_\ell) \bigwedge_{\mathbf{f} \in \mathbf{F}} \mathrm{af}(\mathbf{f})$, where $\{\mathbf{n}_1, \ldots, \mathbf{n}_\ell\} = \mathbf{N} - \{\mathbf{m}_1, \ldots, \mathbf{m}_k\}$.

We can now consider the five basic operations of GOOD and their simulation in FO + **new** + **while**.

1. A node addition[8] $\mathrm{NA}[J, K, \{(\alpha_1, \mathbf{m}_1), \ldots, (\alpha_k, \mathbf{m}_k)\}]$ is simulated as follows. We denote $(K, \alpha_i, \lambda(\mathbf{m}_i))$ by p_i.

$$\begin{aligned} NA &:= \textbf{tuple-new}\ \big\{(\mathbf{m}_1, \ldots, \mathbf{m}_k) \mid \mathrm{match}(J; \mathbf{m}_1, \ldots, \mathbf{m}_k) \wedge \\ &\qquad \neg(\exists x)\big(K(x) \wedge \textstyle\bigwedge_{i=1}^{k} p_i(x, \mathbf{m}_i)\big)\big\}; \\ p_1 &:= p_1 \cup \{(x, x_1) \mid (\exists x_2) \ldots (\exists x_k) NA(x_1, \ldots, x_k, x)\}; \end{aligned}$$

[8]The notation in [20] also includes parameters for the scheme and the instance on which the node addition operates, but for simplicity we leave these parameters understood.

$\vdots$
$p_k := p_k \cup \{(x, x_k) \mid (\exists x_1) \ldots (\exists x_{k-1}) NA(x_1, \ldots, x_k, x)\}$.

2. An edge addition $\mathrm{EA}[J, \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_k, \alpha_k, \mathbf{m}'_k)\}]$ is simulated as follows. We denote $(\lambda(\mathbf{m}_i), \alpha_i, \lambda(\mathbf{m}'_i))$ by p_i.

   $EA := \{(\mathbf{m}_1, \ldots, \mathbf{m}_k, \mathbf{m}'_1, \ldots, \mathbf{m}'_k) \mid \mathrm{match}(J; \mathbf{m}_1, \ldots, \mathbf{m}_k, \mathbf{m}'_1, \ldots, \mathbf{m}'_k)\}$;
   $p_1 := p_1 \cup \{(x_1, x'_1) \mid (\exists x_2)(\exists x'_2) \ldots (\exists x_k)(\exists x'_k) EA(x_1, x'_1, \ldots, x_k, x'_k)\}$;
   $\vdots$
   $p_k := p_k \cup \{(x_k, x'_k) \mid (\exists x_1)(\exists x'_1) \ldots (\exists x_{k-1})(\exists x'_{k-1}) EA(x_1, x'_1, \ldots, x_k, x'_k)\}$.

3. An edge deletion $\mathrm{ED}[J, \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_k, \alpha_k, \mathbf{m}'_k)\}]$ is simulated in an analogous manner as the corresponding edge addition. It suffices to replace each union in the simulation by a difference.

4. A node deletion $\mathrm{ND}[J, \mathbf{m}]$ is simulated as follows:

   $ND := \{\mathbf{m} \mid \mathrm{match}(J; \mathbf{m})\}$;
   $\lambda(\mathbf{m}) := \lambda(\mathbf{m}) - ND$;

   followed by, for each production $p = (\lambda(\mathbf{m}), \alpha, L) \in \mathcal{P}$:

   $p := \{(x, y) \mid p(x, y) \wedge \neg ND(x)\}$;

   followed by, for each production $p = (K, \alpha, \lambda(\mathbf{m})) \in \mathcal{P}$:

   $p := \{(x, y) \mid p(x, y) \wedge \neg ND(y)\}$.

5. An abstraction $\mathrm{AB}[J, \mathbf{n}, K, \alpha, \beta]$ is simulated as follows. We denote the set of productions in $\mathcal{P}$ of the form $(\lambda(\mathbf{n}), \alpha, L)$ by $\mathcal{Q}$, and we denote $(K, \beta, \lambda(\mathbf{n}))$ by p.

   $AB_1 := \{(\mathbf{n}, y) \mid \mathrm{match}(J; \mathbf{n}) \wedge \bigvee_{q \in \mathcal{Q}} q(\mathbf{n}, y)\}$;
   $Equiv := \{(x, x') \mid \lambda(\mathbf{n})(x) \wedge \lambda(\mathbf{n})(x') \wedge (\forall y)(AB_1(x, y) \leftrightarrow AB_1(x', y))\}$;
   $AB_2 := \textbf{set-new} \big\{(x, y) \mid AB_1(x, y) \wedge$
   $\neg(\exists z)\big(K(z) \wedge p(z, x) \wedge (\forall x')(p(z, x') \leftrightarrow Equiv(x, x'))\big)\big\}$;
   $K := K \cup \{z \mid (\exists x) AB_2(x, z)\}$;
   $p := p \cup \{(z, x) \mid AB_2(x, z)\}$.

To complete the simulation of GOOD by FO + **new** + **while** it remains to simulate the method construct. Methods in GOOD are procedures; it is general knowledge in programming that non-recursive procedure calls do not add expressive power and that recursive procedure calls can be simulated using while loops. Van Rossum [33] has done this exercise in the context of GOOD.

A.2 From FO + new + while to IQL

Relational database schemes and instances can be directly represented in the IQL data model. The domain of the instance is kept in some fixed class *Dom*. The type $\mathbf{T}(Dom)$ is not important and can be set to the empty tuple type. Each relation R of arity α is directly represented as an IQL relation R of type $\mathbf{T}(R) = [Dom, \ldots, Dom]$ (α times).

IQL is a rule-based language with inflationary semantics. The language FO + **new** + **while**, in contrast, is based on first-order logic, composition, and while-loops, and assignment to relation variables in its programs need not be inflationary. It is known however [3, 4, 6] that all these features can be simulated in a sufficiently powerful inflationary rule language such as IQL.

We can therefore concentrate on the operations **tuple-new** and **set-new**. A tuple-new statement

$$R := \textbf{tuple-new}\ \{(x_1, \ldots, x_k) \mid \varphi\}$$

is simulated by a rule

$$R(x_1, \ldots, x_k, z) \longleftarrow \varphi.$$

A set-new statement

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

is simulated as

$$\begin{array}{l} Proj_1(x) \longleftarrow \varphi(x, y) \\ Aux_1(x, z) \longleftarrow Proj(x) \\ \widehat{z}(y) \longleftarrow Aux_1(x, z),\ \varphi(x, y) \\ Proj_2(\widehat{z}) \longleftarrow Aux_1(x, z) \\ Aux_2(s, w) \longleftarrow Proj_2(s) \\ R(x, w) \longleftarrow Aux_1(x, z),\ Aux_2(\widehat{z}, w). \end{array}$$

A.3 From IQL to GOOD

We assume the reader is familiar with IQL as described in [4]; we will use the terminology and notation of that paper.

Schemes and instances. We first need to represent IQL schemes and instances by GOOD schemes and instances.

Let $\mathcal{S} = (\mathbf{P}, \mathbf{R}, \mathbf{T})$ be an IQL scheme. We define a GOOD scheme good$(\mathcal{S})$ representing $\mathcal{S}$ as follows. The set of object labels is $\{D, V\} \cup \mathbf{P} \cup \mathbf{R}$. The set of productions consists of the following:

1. For each attribute A occuring in $\mathcal{S}$, the production (V, A, V), where A is used as a functional edge label;
2. The production $(V, \ni, V)$, where '$\ni$' is used as a multivalued edge label;
3. The production (V, ν, V), where 'ν' is used as a multivalued edge label;
4. The production $(D, \ni, V)$;
5. For each class name $P \in \mathbf{P}$, the production $(P, \ni, v)$;
6. For each relation name $R \in \mathbf{R}$, the production $(R, \ni, V)$.

Let $I = (\rho, \pi, \nu)$ be an instance of the IQL scheme $\mathcal{S}$. We define an instance of good$(\mathcal{S})$ representing I as follows. The set of nodes consists of the following:

- All o-values appearing in I, labeled by V;
- A node D, labeled D;
- For each class name P, a node P labeled P;
- For each relation name R, a node R labeled R.

The set of edges consists of the following:

1. For each tuple value $v = [A_1: v_1, \ldots, A_k: v_k]$, the edges (v, A_i, v_i) for each $i = 1, \ldots, k$;
2. For each set value v, the edges $(v, \ni, v')$ for each $v' \in v$;

3. For each oid o such that $\nu(o)$ is defined, the edge $(o, \nu, \nu(o))$;
4. For each constant d, the edge $(D, \ni, d)$;
5. For each class name P, the edges $(P, \ni, o)$ for each $o \in \pi(P)$;
6. For each relation name $R \in \mathbf{R}$, the edges $(R, \ni, v)$ for each $v \in \rho(R)$.

We note two useful facts about the above representation of schemes and instances:

1. The function $\mathbf{T}$ in the IQL schema, which assigns types to class and relation names, is not represented in the corresponding GOOD schema. However, given a type τ, there is a GOOD program FIND_τ that works on the GOOD representation of any IQL instance, and marks all o-values that are of type τ. More specifically, this program creates a node τ labeled τ and adds edges $(\tau, \ni, \mathbf{n})$ for each V-labeled node $\mathbf{n}$ representing an o-value of type τ.

   Note that when τ is an "atomic type", i.e., $\tau = D$ or $\tau = P$ with P a class name, this information is already present in the instance. So for atomic types τ the program FIND_τ is trivial. The construction of FIND_τ for more complex types τ then is an obvious structural induction.

2. In the GOOD representation of an IQL instance, each o-value is represented by a unique node. However, in our simulation of IQL programs by GOOD programs, it will be possible that intermediate results contain different nodes representing the same o-value (this will not be the case for atomic o-values, i.e., constants and oids). Such nodes will be called *value-equal*.

   More formally, let $\mathbf{n}$ and $\mathbf{m}$ be nodes representing o-values of type τ. Value-equality with respect to τ is defined inductively as follows:

   (a) If τ is D or a class name, $\mathbf{n}$ and $\mathbf{m}$ are value-equal w.r.t. τ if and only if they are identical.

   (b) If τ is $[A_1{:}\,\tau_1, \ldots, A_k{:}\,\tau_k]$, $\mathbf{n}$ and $\mathbf{m}$ are value-equal w.r.t. τ if for each $i = 1, \ldots, k$ there are edges $(\mathbf{n}, A_i, \mathbf{n}_i)$ and $(\mathbf{m}, A_i, \mathbf{m}_i)$ such that $\mathbf{n}_i$ and $\mathbf{m}_i$ are value-equal w.r.t. τ_i.

(c) If τ is $\{\tau'\}$, $\mathbf{n}$ and $\mathbf{m}$ are value-equal w.r.t. τ if for each edge $(\mathbf{n}, \ni, \mathbf{n}')$ there is an edge $(\mathbf{m}, \ni, \mathbf{m}')$ such that $\mathbf{n}'$ and $\mathbf{m}'$ are value-equal w.r.t. τ', and vice versa, for each edge $(\mathbf{m}, \ni, \mathbf{m}')$ there is an edge $(\mathbf{n}, \ni, \mathbf{n}')$ such that $\mathbf{m}'$ and $\mathbf{n}'$ are value-equal w.r.t. τ'.

(d) If τ is $\tau_1 \vee \tau_2$, then $\mathbf{n}$ and $\mathbf{m}$ are value-equal w.r.t. τ if for either $i = 1$ or $i = 2$, $\mathbf{n}$ and $\mathbf{m}$ are value-equal w.r.t. τ_i.

By structural induction, for every type τ a GOOD program EQUAL_τ can be constructed that adds edges labeled '$equal_\tau$' between exactly those pairs of nodes that are value-equal w.r.t. τ.

Valuation-domains. We now turn to the expression of the valuation-domain of an IQL program in GOOD.

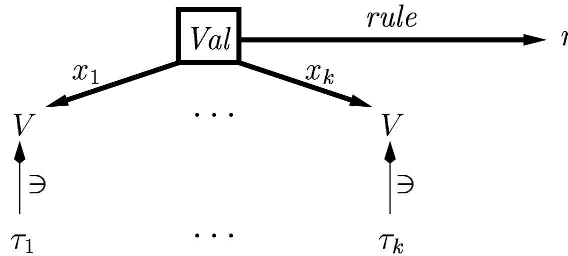
Since valuations are linked to rules, we represent each rule r as a node in the instance by a sequence of node additions. The label of each r is r itself.

Valuations are represented as follows. Let r be a rule and let θ be a valuation of $body(r)$. The pair (r, θ) is represented by a node $\mathbf{n}$ labeled '*Val*', with for each variable x in $body(r)$ an edge $(\mathbf{n}, x, \theta(x))$, and with an edge $(\mathbf{n}, rule, r)$. Here, each variable as well as '*rule*' is used as a functional edge label.

Let r be a rule, and let $\{x_1, \ldots, x_k\}$ be the set of variables in $body(r)$. Let the declared type of x_i be τ_i, for each $i = 1, \ldots, k$. The set of all pairs (r, θ) with θ a valuation of $\{x_1, \ldots, x_k\}$ can be constructed as follows:

1. For each $i = 1, \ldots, k$, apply FIND_{τ_i};

2. Apply the node addition



We have to delete those pairs (r, θ) that are not in the valuation-domain. For these pairs, θ makes $body(r)$ true, i.e., it makes each positive literal in

body(r) true and makes each negated literal in *body*(r) false. Marking the valuations that make a literal false amounts to marking all valuations and then deleting those that make the literal true.

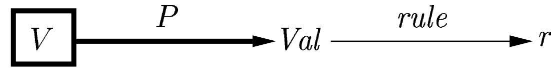
So let us see how to test whether a valuation θ makes a literal of the form $t_1 = t_2$ or $t_1(t_2)$ true. This can be done by deriving representations $\mathbf{n}_1$ and $\mathbf{n}_2$ of the o-values $\theta(t_1)$ and $\theta(t_2)$, after which

- in the case $t_1 = t_2$, we test whether $\mathbf{n}_1$ and $\mathbf{n}_2$ are value-equal w.r.t. the appropriate type, and
- in the case $t_1(t_2)$, we test whether there is a node $\mathbf{n}$ such that $\mathbf{n}$ is value-equal to $\mathbf{n}_2$ w.r.t. the type of t_2, and such that there is an edge $(\mathbf{n}_2, \ni, \mathbf{n})$.

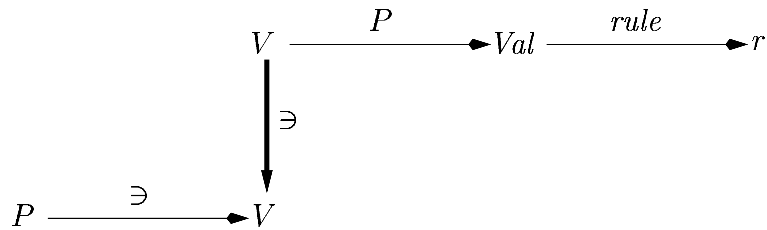
These tests can be performed by an application of EQUAL$_\tau$ for the appropriate types τ, followed by a simple pattern match.

The just-mentioned representation of $\theta(t)$ for some term t can be derived as follows. The representation will be a node, linked to (r, θ) by an edge labeled t. The derivation is by induction on t:

- If t is a variable x, the representation is already there.
- If t is a class name P, perform the node addition

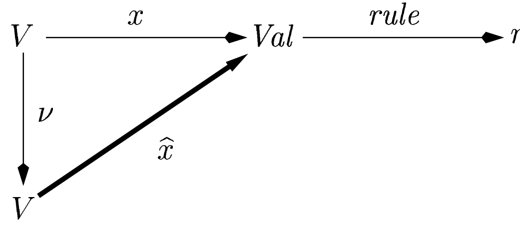


followed by the edge addition

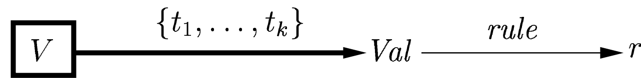


- If t is a relation name R, do the same as in the previous item with R substituted for P.

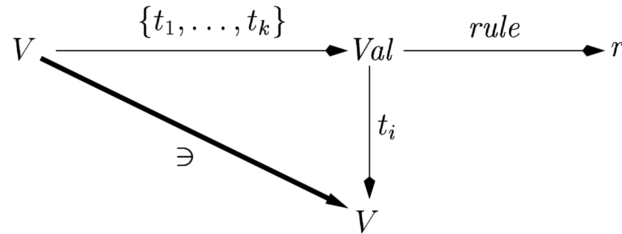
- If t is of the form $\widehat{x}$, perform the node addition



- If t is of the form $\{t_1, \ldots, t_k\}$, derive the representations of $\theta(t_1)$, ..., and $\theta(t_k)$, and then perform the node addition



followed by the edge additions



for each $i = 1, \ldots, k$.

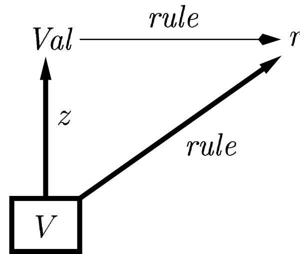
- If t is of the form $[A_1 \colon t_1, \ldots, A_k \colon t_k]$, do the same as in the previous item with $[A_1 \colon t_1, \ldots, A_k \colon t_k]$ substituted for $\{t_1, \ldots, t_k\}$ and, for each $i = 1, \ldots, k$, A_i substituted for $\ni$.

We can now assume that all pairs (r, θ) for which θ does not make $body(r)$ true have been deleted. To arrive at the valuation-domain, it remains to also delete those pairs where θ can be extended to the variables in $head(r)$ so as to make $head(r)$ true. This can be achieved by the same techniques just explained.

Valuation-maps. We next turn to the computation in GOOD of a valuation-map on the valuation-domain. This amounts to extending, for each (r, θ) in the valuation-domain, θ to the variables in $head(r)$ by creating new oids.

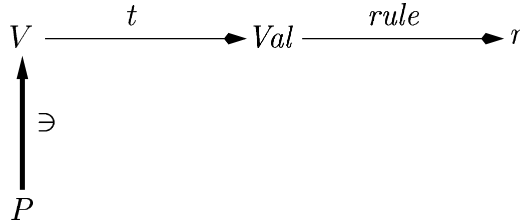
By the previous paragraphs, we can assume that of all nodes (r, θ) only those in the valuation-domain remain. In addition to this, we also delete the pairs (r, θ) where $head(r)$ is of the form $\widehat{x} = t$, and $\nu(\theta(x))$ is already defined. This can be tested by a simple pattern.

The above-mentioned extension of the valuations in the valuation-domain is now done by the following node addition for each rule r and each variable z in $head(r)$ not in $body(r)$:



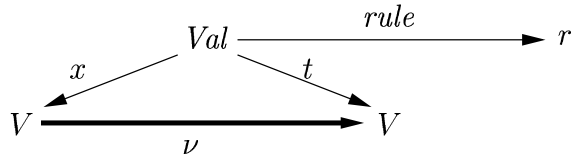
The simulation. The application of a rule r, given that the valuation-map has already been computed, can now be performed as follows. First, as explained earlier, derive a representation of $\theta(t)$, where t is the term in $head(r)$, for each (r, θ). Then:

- If $head(r)$ is of the form $P(t)$ with P a class name, perform the edge addition



- If $head(r)$ is of the form $R(t)$ with R a relation name, do the same as in the previous item with R substituted for P.

- If $head(r)$ is of the form $\widehat{x} = t$, perform the edge addition



We are finally ready to describe the complete simulation of an IQL program by a GOOD program:

compute the valuation-domain;
while valuation-domain not empty **do**
  compute the valuation-map;
  apply each rule;
  undo ambiguous assignments;
  delete auxiliary nodes and edges;
  compute valuation-domain
od.

The clause 'undo ambiguous assignments' corresponds to a semantic check that is built in in the inflationary fixpoint computation of IQL programs. The check detects nodes having two or more outgoing edges labeled ν and deletes these edges. This can be easily programmed in GOOD.

Duplicate elimination. When applying the GOOD simulation P of an IQL program Γ on the GOOD representation of an IQL instance I, we obtain a correct GOOD representation of $\Gamma(I)$, with the exception that value-equal nodes will be present. It remains to perform a duplicate elimination, replacing each equivalence class of value-equal nodes by a single representative. This is accomplished by using the abstraction operation of GOOD (which has not yet been used so far).

To perform duplicate elimination on all nodes representing o-values of type τ, we execute the GOOD program DUPELIM$_\tau$ described in detail below. The complete duplicate elimination process consists of an application of DUPELIM$_\tau$ for each non-atomic type τ occurring in the output schema of the IQL program to be simulated, such that whenever some type τ' occurs in another type τ, DUPELIM$_{\tau'}$ is performed before DUPELIM$_\tau$.

The program DUPELIM$_\tau$ first performs FIND$_\tau$, then performs EQUAL$_\tau$, and then performs an abstraction, a series of edge additions, and a node deletion in the order as shown in Figure 17.

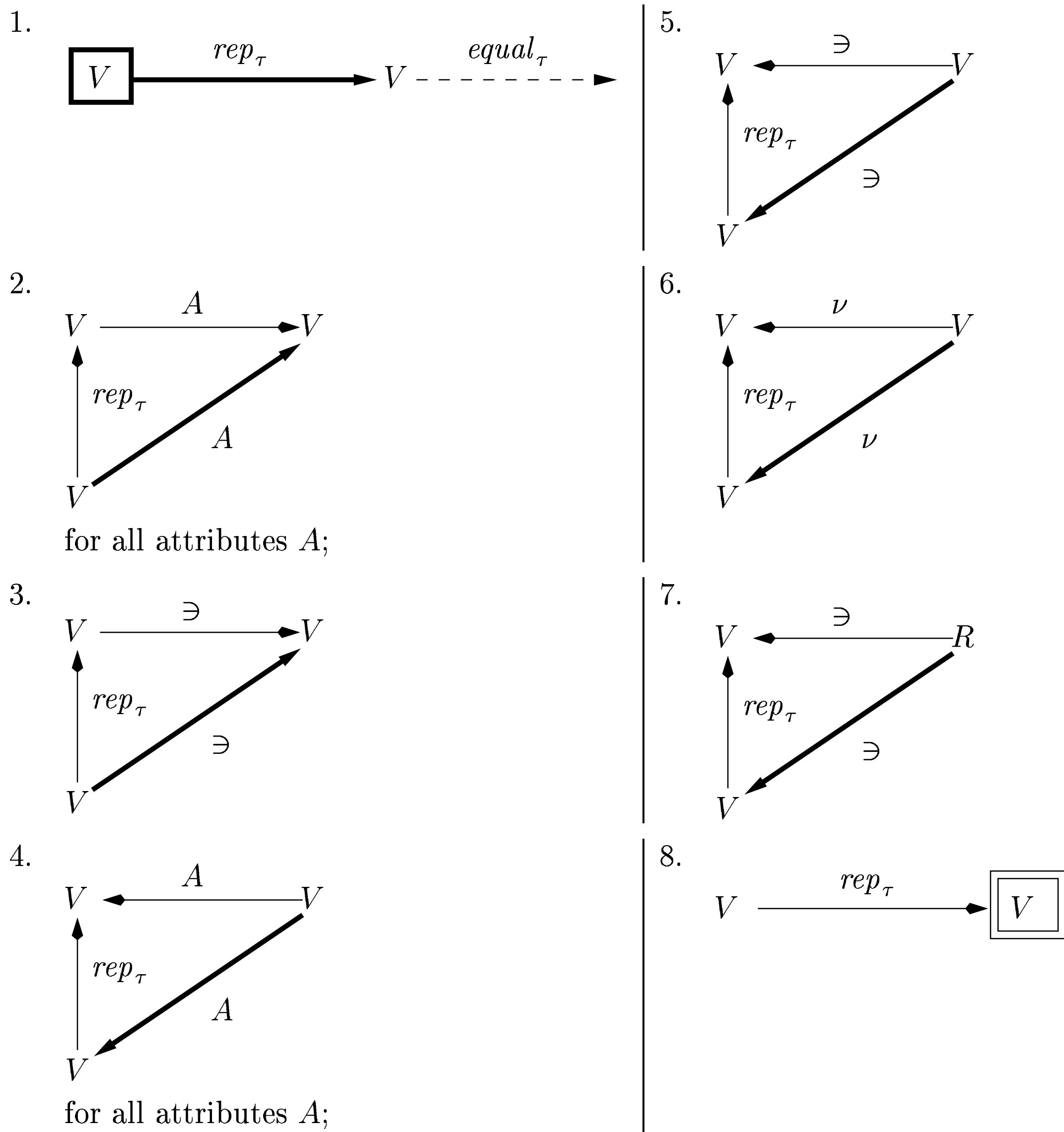


Figure 17: The final duplicate elimination step.

This concludes the Appendix.