The expressive power of complex values in object-based data models*

Jan Van den Bussche† Jan Paredaens

Dept. Math. & Computer Sci., University of Antwerp (UIA)
Universiteitsplein 1, B-2610 Antwerp, Belgium
E-mail: vdbuss@wins.uia.ac.be, pareda@wins.uia.ac.be.

Abstract

In object-based data models, complex values such as tuples or sets have no special status and must therefore be represented by objects. As a consequence, different objects may represent the same value, i.e., duplicates may occur. This paper contains a study of the precise expressive power required for the representation of complex values in typical object-based data models supporting first-order queries, object creation, and while-loops. Such models are sufficiently powerful to express any reasonable collection of complex values, provided duplicates are allowed. It is shown that in general, the presence of such duplicates is unavoidable in the case of set values. In contrast, duplicates of tuple values can easily be eliminated. A fundamental operation for duplicate elimination of set values, called abstraction, is considered and shown to be a tractable alternative to explicit powerset construction. Other means of avoiding duplicates, such as total order, equality axioms, or copy elimination, are also discussed.

*Extended and revised version of a paper presented at the Tenth ACM Symposium on Principles of Database Systems [VP91].

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

1 Introduction

In the past decade, there has been a lot of interest in database systems allowing a more direct representation of complex data structures than possible in standard relational systems. Recent work in this field lead to the definition of two new data models: the *complex value* model and the *object-based* model. (There were also proposals to combine the two approaches [AK89, Bee90, HK87, LRV88].)

The complex value model (also known as the complex object, nested relational, NF^2, or unnormalized model [AFS89]) is an extension of the standard relational model [Ull88]. While the relational model offers collections of tuples, the complex value model offers collections of arbitrary combinations of sets and tuples called *complex values*. In the object-based model [GPVG94, HS89, HY90, KV93, KW93], a database is thought of as a labeled graph of objects, where each set of equally labeled objects comprises a so-called class. The edges between objects in the graph express properties and are labeled by property names. This approach is inspired by the object-oriented philosophy [KL89], but can in fact be traced back to the Functional Data Model [Shi81].

A difference between the complex value approach and the object-based approach is that in the latter, complex values are not explicitly part of the data model. The usual way of representing a complex value in such a model is by an object. More specifically, an n-ary tuple is represented by an object, linked to each of the n tuple components by a differently labeled edge. Similarly, a set of objects is represented by an object with equally labeled edges going to each element of the set. A class of objects then represents a collection of complex values if each object in the class represents a complex value in the collection, and conversely, each complex value in the collection is represented by an object in the class. However, it may occur that two different objects in the class represent the same value, i.e., *duplicates* may occur.

In this paper, we will be concerned with object-based queries whose result is a collection of complex values. Such queries augment the database with the new objects and edges necessary for representing the desired collection. The fundamental query language for relational databases, the relational calculus [Ull88], can be adapted for this purpose. More specifically, a relational calculus query over the database graph can be used for object creation by

creating a new object for each tuple in its result. Moreover, if the result is a binary relation, it can be alternatively used for edge addition. A simple yet powerful object-based query language, which we call OBQL, can thus be obtained by providing object creation and edge addition as basic statements and closing off under composition and while-loops. This language subsumes many object-based query languages proposed in the literature.

We will show, using well-known techniques, that any complex value query satisfying the usual requirements of computability and genericity can be expressed in OBQL. However, we will show also that this completeness property depends on the allowance of duplicates in the result. Indeed, if one insists on duplicate-free representations, even very simple collections of set values become inexpressible in OBQL. For example, we will show that the query asking for all subsets of two elements of a given class is not expressible without duplicates. In contrast to sets, duplicates of tuple values *can* easily be eliminated in OBQL, which is not completely unexpected since the core of OBQL is the relational calculus, which is tuple-based from the outset.

Duplicate-free representations have a number of apparent practical advantages. Obviously, duplicates cause redundancy in the database and are often undesirable from a faithful data modeling point of view. Another advantage concerns the efficient answering of queries involving the equality of complex values. In arbitrary representations, checking for equality of two complex values requires an expensive comparison of all components. However, if every complex value is represented by a unique object, checking equality amounts to one single comparison of the corresponding object identifiers. A third advantage is efficiency of representation. If a client program asks for a collection of complex values, it is useful if the server program can deliver the collection in the form of a unique handle to each element. Different handles to the same structured value, i.e., duplicates, would be very undesirable in this situation.

Hence, it is desirable to enrich OBQL with an additional primitive for the creation of duplicate-free value representations. Since the problem of duplicate elimination exists only for collections of set values, one might simply add an explicit primitive for constructing such collections. The obvious candidate for this, first considered by Kuper and Vardi in the context of the Logical Data Model [KV93], is the *powerset* operation. Alternatively, one might propose a quotient construction for creating a unique representative for each equivalence class of duplicate objects. An operation providing

this functionality is the *abstraction* operation, first considered by Gyssens, Paredaens and Van Gucht in the context of the GOOD model [GPVG94]. We will show that these two options for enriching OBQL are equivalent in expressive power.

However, an important advantage of the abstraction operation over the powerset operation is that the former is more efficient. Indeed, we will show that *on ordered databases*, abstraction is expressible in OBQL without using while-loops. As a consequence, the abstraction operation can be computed in polynomial time on a Turing machine. We also show that any arbitrary computable and generic equivalence relation is reducible within OBQL to the particular equivalence relation "is a duplicate of" as dealt with by the abstraction operation. In other words, every reasonable quotient construction can be reduced to the particular quotient construction provided by the abstraction operation.

We will conclude with a discussion on related issues. In particular, we will observe how the problems of object creation, complex values and duplicates can be understood in the context of other data models and object-based query languages proposed in the literature. We will also explain the connection between the complex value queries as considered in this paper and general object-creating queries, and in particular, explain the distinction between the notion of duplicate elimination studied in the present paper and the notion of *copy elimination* introduced by Abiteboul and Kanellakis [AK89]. We will also briefly mention further research issues.

2 The object-based data model

In this section, we define a general object-based data model, which serves as a formal framework capturing the features (relevant to this paper) of many object-oriented database systems encountered in practice. We will define database schemes and instances as directed, labeled graphs, and introduce a simple yet powerful object-based query language, called OBQL. Our formalism closely follows the earlier proposals LDM [KV93] and GOOD [GPVG94].

It is customary in object-based models to depict a database scheme as a graph. Thereto we assume the existence of pairwise disjoint sets of *class names*, *single-valued property names*, and *multi-valued property names*, and

define:

Definition 2.1 *A* scheme *is a finite, edge-labeled, directed graph. The nodes of the graph are class names, and the edges are triples* (B, e, C)*, where* B *and* C *are nodes and the edge label* e *is a (single- or multi-valued) property name.*

A database instance can now be defined as a graph consisting of objects and property-links, the structure of that graph being constrained by some database scheme. So we assume the existence of an infinite supply of *objects*, and define, for an arbitrary scheme S:

Definition 2.2 *An* instance *over* S *is a finite, labeled, directed graph. The nodes of the graph are objects. Each node* o *is labeled by a class name* $\lambda(o)$ *of* S*. The edges are triples* (o, e, p)*, where* o *and* p *are nodes and the edge label* e *is a property name of* S *such that* $(\lambda(o), e, \lambda(p))$ *is an edge of* S*. If* e *is single-valued, then for each* o *there is at most one* p *such that* (o, e, p) *is in the graph.*

The set of all objects in an instance labeled by the same class name C will be called *the class* C.

Before turning to the object-based query language OBQL, we must first specify what we mean with the notion of query in the object-based data model. In the relational model, a query is typically considered a function, mapping an input database to an output relation [CH80]. This output relation is often materialized as derived information, or used as part of the input to a subsequent query. Hence, it is natural to view a relational query alternatively as a function which augments an input database with a new, derived relation. This view of a query can be readily adopted in the object-based data model: a query is a function which augments an input instance with new objects and edges. Correspondingly, OBQL provides two basic operations, one for object creation and one for edge addition. The language is closed off under composition and while-loops.

Object creation and edge addition are based on the following adaptation of the relational calculus to object databases. With a scheme S, we can associate a standard, first-order, many-sorted logic. The class names of S are the sorts, and for each edge (B, e, C) in the scheme there is a binary predicate name e of sort (B, C). Given an instance I over S, a sort C is

interpreted by the class C in I, and the predicate $e(B, C)$ is interpreted by the set of all e-labeled edges going from objects of class B to objects of class C. Now let $\Phi(x_1, \ldots, x_n)$ be a formula over S, $n \geq 0$, and let C_i be the sort of variable x_i. Evaluating $\{(x_1, \ldots, x_n) \mid \Phi\}$ over I yields an n-ary relation consisting of all tuples $(o_1, \ldots, o_n)$ of objects in I satisfying Φ. Note that o_i will be in class C_i.

The *object creation* operation:

$$\Gamma \equiv C[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \Phi$$

provides a natural way to augment the database with a representation of the above n-ary relation. Here, C is some class name and $e_1, \ldots, e_n$ are property names. The effect of Γ on schemes S and instances I is formally defined as follows:

Definition 2.3

- $\Gamma(S)$ *is the scheme obtained by augmenting* S *with node* C *and edges* (C, e_i, C_i)*, for* $i = 1, \ldots, n$*. (By* augmenting *a graph* G *with a node or edge* x*, we mean adding* x *to* G *provided* x *does not already belong to* G*.)*

- $\Gamma(I)$ *is the instance over* $\Gamma(S)$ *obtained from* I *by adding, for each tuple* $(o_1, \ldots, o_n)$ *of objects in* I *such that* $\Phi(o_1, \ldots, o_n)$ *is true in* I*, one new object* o *with label* C*, together with edges* (o, e_i, o_i)*, for* $i = 1, \ldots, n$*.*

If $n = 2$, then evaluating formula Φ yields a binary relation, which can be used not only for object creation, but also for edge addition. Indeed, each pair in the relation can be interpreted as a set of derived edges. These can be added to the database using the *edge addition* operation

$$\Delta \equiv e(x_1, x_2) \Leftarrow \Phi,$$

where e is some multi-valued property name. The effect of Δ on schemes and instances is formally defined as follows:

Definition 2.4

- $\Delta(S)$ *is the scheme obtained by augmenting* S *with the edge* (C_1, e, C_2)*.*

- $\Delta(I)$ *is the instance over* $\Delta(S)$ *obtained by augmenting* I *with an edge* (o_1, e, o_2) *for each tuple* (o_1, o_2) *of objects in* I *such that* $\Phi(o_1, o_2)$ *is true in* I.

Queries can now be expressed in OBQL by means of arbitrary compositions of object creation and edge addition operations. Furthermore, these compositions can be iterated using a while-loop construct of the form

while change do $op_1; \ldots; op_k$ **od**.

The body of the loop is executed as long as the instance under operation changes (which might be forever). The instance resulting from the execution of an OBQL program P on an instance I will be denoted by $P(I)$.

Remarks. We conclude this section with some remarks on specific features of OBQL.

The definitions of object creation and edge addition allow that the labels of objects and edges that are added to an instance already exist in the scheme of that instance. This provision is necessary for adding deriving information incrementally, e.g., using a while-loop. For example, the following program computes the transitive closure of a database graph whose objects are all in the same class and whose edges all have the same label e. The edges of the transitive closure will get the label e^*.

$e^*(x, y) \Leftarrow e(x, y)$;
while change do $e^*(x, y) \Leftarrow \exists z : e(x, z) \wedge e^*(z, y)$ **od**

A program expressing a query will often create a lot of temporary objects and edges that are only used for storing intermediate results in the course of the computation, and should be omitted from the end result. We can make this formal by dividing object and edge labels (i.e., class and property names) into three kinds: those that are in the input scheme, those that are wanted in the ouput, and those that are only auxiliary and are used to label the intermediate results. It will always be clear from the context which of the labels are auxiliary; we will never explicitly indicate them.

A number of object-based data models considered in the literature [GPVG94, HY90, KW93] use an alternative semantics for object creation, which we will call *weak* semantics, and which is often natural and useful.

Recall Definition 2.3 of the object creation operation. The weak variant of this operation, written

$$C[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow_{\text{weak}} \Phi,$$

only adds a new object o (as specified in the definition) if there is not already a C-labeled object o' with edges (o', e_i, o_i) in the database. Hence, it is equivalent to

$$C[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \Phi \wedge \neg\exists x : e_1(x, x_1) \wedge \cdots \wedge e_n(x, x_n).$$

Thus, the weak semantics can be simulated in our semantics. Actually, the converse is true as well. The converse simulation uses an auxiliary class T, structured as an ever-growing stack. The stack is initialized with a bottom object using the "zero-ary" object addition:

$$T[\,] \Leftarrow_{\text{weak}} \mathbf{true}.$$

The object creation

$$C[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \Phi$$

is then simulated by first pushing a new object on the stack:

$$T[\mathit{previous} : t] \Leftarrow_{\text{weak}} \neg\exists t' : \mathit{previous}(t', t).$$

(Here, t and t' are variables of sort T; the formula states that t is the top of the stack.) The actual object creation is then performed by

$$C[e_1 : x_1, \ldots, e_n : x_n, e : t] \Leftarrow_{\text{weak}} \Phi \wedge \neg\exists t' : \mathit{previous}(t', t).$$

In words, the new object must be connected to the top of the stack with a temporary edge labeled e. This guarantees that it will indeed be created, regardless of whether there already exists an object with the same $e_1, \ldots, e_n$-edges, since such an object will be connected to a lower object in the stack.

A final remark concerns the sorts of the logical variables used in object creation and edge addition operations. In the sequel, when OBQL programs are shown, we will precede these programs by a *variable declaration* which indicates the sorts of the different variables that are used. For example, to declare variables x and y as of sort C and variable z as of sort B, we will write

sorts $x, y : C$; $z : B$

before the beginning of a program using variables x, y and z. Note that variable declarations strictly belong to the "meta language" and are not formally part of the syntax of OBQL.

3 Representation of complex values

In this section, we show how complex values, in the widely adopted style of [AB88, AK89], can be represented in the object-based data model, and establish a completeness result for OBQL in this respect.

We first need the *types*, which are expressions inductively defined as follows. In what follows, S is an arbitrary but fixed scheme.

Definition 3.1

1. *Every class name of S is a type, called an* atomic type.
2. *If $e_1, \ldots, e_n$ are different single-valued property names, and $\tau_1, \ldots, \tau_n$ are types, then $[e_1 : \tau_1, \ldots, e_n : \tau_n]$ is a type, called a* tuple type.
3. *If e is a multi-valued property name, and τ is a type, then $\{e : \tau\}$ is a type, called a* set type.

The order in which the different $e_i : \tau_i$ are listed in a tuple type is unimportant.

The *extent* of a type over a given instance I over S, which is the set of all possible values of that type, is now inductively defined as follows.

Definition 3.2

1. *The extent of atomic type C is the class C in I.*
2. *The extent of tuple type $[e_1 : \tau_1, \ldots, e_n : \tau_n]$ is the set of all tuples $[e_1 : v_1, \ldots, e_n : v_n]$ such that v_i is a value of type τ_i.*
3. *The extent of set type $\{e : \tau\}$ is the set of all sets of values of type τ.*

Values of non-atomic types are called complex values.

In data models combining the complex value and the object-based approach, complex values are "first-class citizens" of the model and are typically associated to objects according to a so-called *type assignment* (see the references in the Introduction). In contrast, the "pure" object-based data model defined in the previous section does not explicitly contain complex values. However, values can be naturally represented using objects. An n-ary tuple value can be represented by an object with n single-valued properties. A set value can be represented by an object with a multi-valued property. To make this idea formal, we need the "dual" notion of a type assignment, which we call a class assignment:

Definition 3.3 *Let τ be a type. A* class assignment for τ *is a mapping α from τ and all types occurring in τ to class names, such that:*

1. *Atomic types (i.e., class names) are mapped to themselves.*
2. *If $\alpha([e_1 : \tau_1, \ldots, e_n : \tau_n]) = C$, then for each i, $(C, e_i, \alpha(\tau_i))$ is an edge of S.*
3. *If $\alpha(\{e : \tau'\}) = C$, then $(C, e, \alpha(\tau'))$ is an edge of S.*

In other words, $\alpha(\tau) = C$ means that C has the right properties for values of type τ to be representable by objects of class C. Let an arbitrary class assignment α be fixed. Let I be an instance, let v be a value over I of type τ, and let o be an object in class $\alpha(\tau)$ in I. We define inductively what it means for o to represent v:

Definition 3.4

1. *If τ is atomic, then o* represents *v if $o = v$.*
2. *If τ is the tuple type $[e_1 : \tau_1, \ldots, e_n : \tau_n]$ and $v = [e_1 : v_1, \ldots, e_n : v_n]$, then o represents v if there are objects $o_1, \ldots, o_n$ of classes $\alpha(\tau_1), \ldots, \alpha(\tau_n)$ respectively such that for each i, (o, e_i, o_i) in I and o_i represents v_i.*

3. *If τ is the set type $\{e : \tau'\}$, then o represents v if for each $v' \in v$ there is an edge (o, e, o') in I such that o' is of class $\alpha(\tau')$ and represents v', and conversely, for each o' of class $\alpha(\tau')$ such that (o, e, o') the value v' represented by o' is in v.*

In the sequel, we will almost never explicitly mention the class assignment in use if this is clear from the context.

Since we are primarily interested not in single values, but in collections V of complex values of a common type, we finally define:

Definition 3.5 *A class* represents *a collection V if each object o in the class represents a value in the collection, and conversely, each value in the collection is represented by some object in the class.*

We are now ready to present a first result which indicates that the query language OBQL defined in the previous section is quite powerful. Let S be a scheme and let τ be a complex value type.

Definition 3.6 *A* complex value query of type $S \to \tau$ *is a function V, mapping each instance I over S to a collection $V(I)$ of complex values over I of type τ.*

As usual in database applications, we consider only mappings which do not "interpret" the objects: objects are abstract entities and only their interrelationships matter [AU79, CH80]. This requirement, nowadays known as *genericity* (e.g., [AV90, HS93]), is formalized by the requirement that the mapping, viewed logically as a binary relationship, is invariant under permutations of the universe of objects. Furthermore, a complex value query V is called *computable* if the problem: given an instance I and a complex value v over I of type τ, does v belong to $V(I)$? is decidable. Finally, V is called *expressible in OBQL* if there is a class name K and an OBQL program P augmenting each instance I with the necessary objects and edges such that class K in $P(I)$ represents $V(I)$. Clearly, every complex value query expressible in OBQL is computable and generic. The converse holds as well:

Theorem 3.7 *Every generic computable complex value query is expressible in OBQL.*

To prove Theorem 3.7, we first observe that we may, without loss of generality, restrict attention to types of *depth 1*, defined as follows:

Definition 3.8 *A type has depth 1 if it has the form* $[e_1 : C_1, \ldots, e_n : C_n]$ *or* $\{e : C\}$*, where* $C, C_1, \ldots, C_n$ *are class names.*

Indeed, if we know that the theorem holds for depth 1 types, then the general theorem follows by an obvious bottom-up construction.

We next present two lemmas.

Lemma 3.9 *The extent of the tuple type* $[e_1 : C_1, \ldots, e_n : C_n]$ *is expressible in OBQL.*

Proof. Our task is essentially to represent the Cartesian product of classes $C_1, \ldots, C_n$ by a new class K. This can be done by the following object addition:

sorts $x_1 : C_1; \ldots; x_n : C_n$

$K[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \mathbf{true}(x_1, \ldots, x_n)$

($\mathbf{true}(x_1, \ldots, x_n)$ denotes an arbitrary tautology with free variables $x_1, \ldots, x_n$.) ■

Lemma 3.10 *The extent of the set type* $\{e : C\}$ *is expressible in OBQL.*

Proof. Our task is essentially to represent the powerset of class C by a new class K. This can be done by the following program. Properties a_1 and a_2 are temporary. Recall that by the definition of set types, e must be a multi-valued property name.

sorts $x : C$; $z, z_1, z_2, z_{12} : K$

$K[e : x] \Leftarrow \mathbf{true}(x)$;
while change do
  $K[a_1 : z_1, a_2 : z_2] \Leftarrow \neg\exists z : \forall x : e(z, x) \leftrightarrow (e(z_1, x) \vee e(z_2, x))$;
  $e(z, x) \Leftarrow \exists z_{12} : (a_1(z, z_{12}) \vee a_2(z, z_{12})) \wedge e(z_{12}, x)$
  od
$K[\,] \Leftarrow \mathbf{true}$

■

In the above proof, the powerset is constructed bottom-up, starting from the singletons and then taking pairs until all sets are represented. The final statement adds a representation for the empty set. The representation K thus obtained will contain duplicates, i.e., different K-objects representing the same set of C-objects. This is allowed by Definition 3.5. We will return to the subject of duplicates in detail in the next section.

We are now ready for:

Proof of Theorem 3.7. Let V be a generic computable complex value query of type $S \to \tau$, where τ has depth 1. We have to show that V is expressible in OBQL. Let I be an instance over S.

Since V is computable, there is an effective algorithm that decides for each element of the extent of τ whether it belongs to $V(I)$. Since V is generic, we can assume that this algorithm is implemented on an *input-order independent domain Turing machine (domTM).* The domTM model, introduced in [HS93], is well-suited for describing database computations. A domTM is like a normal Turing machine, but has extra capabilities. Apart from the letters of the finite alphabet, the tape can also contain objects. Hence, there is no need to encode objects as strings over the alphabet, as would be necessary if we would use a conventional Turing machine. The transition function of the machine can refer to the objects in a generic manner only, using two generic variables η and κ which can only distinguish whether two objects are equal or not. The machine has also a register.

Formally, a domTM is a five-tuple $(Q, \Sigma, \delta, q_0, q_a)$, where Q is a finite set of states, Σ is a finite alphabet, q_0 is the initial state and q_a is the accepting state. The transition function $\delta : Q \times \Sigma_1 \times \Sigma_2 \to Q \times \Sigma_2 \times \Sigma_2 \times \{L, R\}$, where $\Sigma_1 = \Sigma \cup \{\eta\}$ and $\Sigma_2 = \Sigma \cup \{\eta, \kappa\}$, maps the current state, the current contents of the register and the contents of the current tape cell to a new state, a new contents of the register, a new contents of the current tape cell, and a movement on the tape. The following technical restrictions apply for transitions $\delta(q, x, y) = (q', x', y', m)$: $y = \kappa$ only if $x = \eta$; $\eta \in \{x', y'\}$ only if $\eta \in \{x, y\}$; and $\kappa \in \{x', y'\}$ only if $\kappa \in \{x, y\}$. A transition is called *generic* if $\eta \in \{x, y\}$. A generic transition serves as a template for the infinite set of transitions formed by letting η (and κ if it occurs) range over (different) objects. Computations of a domTM are now defined in the obvious way.

The input-order independent domTM M_V that computes V gets as input on its tape a description of an instance I, followed by a description of a value

v of type τ over I. I is written on the tape as a string of objects and edges, listed in some arbitrary order. An object o of class C is written as oC. An edge from o to p with label e is written as oep. The value v is written as $e_1o_1 \ldots e_no_n$ if τ is a tuple type, or simply as a string of its elements if τ is a set type. Note that we assume that the finite alphabet of M_V includes the class and property names of the scheme S.

We can construct an OBQL program P_V that expresses V as follows. On input I, P_V first creates a representation of the extent of τ in a class K. Then, for each object o of class K, the computation of M_V on input I, v, where v is the complex value of type τ represented by o, is simulated. If the computation is accepting, then o is marked with a label *accept*. Finally, for each accepted K-object o, an object o' of another class K' is created and the necessary property edges are brought over so that o' represents the same value (of type τ) as o. As a result, class K' represents the collection $V(I)$, as desired.

The first part of program P_V, expressing the extent of τ, was already shown in Lemma 3.9 (if τ is a tuple type) or Lemma 3.10 (if τ is a set type). The final part, the creation of class K', is a straightforward application of object addition and property addition. Hence, we will focus on the middle and most important part, the simulation of M_V. The techniques used thereto are inspired by techniques used in earlier completeness proofs [AV90, CH80, HS93]. In order not to loose focus, in the following exposition we will often refer to the Appendix of this paper for the precise implementation details of the simulation. The reader interested in concrete examples of programs written in OBQL is encouraged to examine this Appendix.

To simulate M_V on input I, v, we must create a tape on which I and v are listed. The nodes and edges of I and, if τ is a set type, the elements of v, can be listed in an arbitrary order. However, it is impossible in OBQL to choose one such order. Therefore, all possible orderings must be generated and a simulation will be run for each of them. So, for each class C, a class $List_C$ is created containing all possible lists of all C-objects, using the following procedure which is similar in spirit to the powerset construction of Lemma 3.10. List objects have the standard single-valued properties *head* and *tail*, and an auxiliary multi-valued property *contains* used to keep track of the generation.

sorts $x, y : C$; $z, z_1, z_2 : List_C$

$List_C[\,] \Leftarrow \mathbf{true}$;
$List_C[head : x, contains : y, tail : z] \Leftarrow y = x$;
while change do
$List_C[head : x, tail : z_1, contains : y] \Leftarrow y = x \wedge$
$\neg contains(z_1, x) \wedge \neg\exists z_2 : head(z_2, x) \wedge tail(z_2, z_1)$;
$contains(z_2, x) \Leftarrow \exists z_1 : tail(z_2, z_1) \wedge contains(z_1, x)$
od
$Order_C(z_1, z_1) \Leftarrow \neg\exists z_2 : tail(z_2, z_1)$

The last statement marks those lists that contain all C-objects, and which can thus be viewed as an ordering of the class C, with a loop-edge labeled $Order_C$.

We also have to order the edges of input instance I. Once this is done, we can deduce the orderings of the whole instance as aggregates of the orderings of the individual classes and properties. This leads to a class *Order* holding an object for each possible ordering of the inputs instance. The implementation details of the ordering of the edges and the construction of the class *Order* are given in Appendix A1.

The two-way infinite tape of M_V will be simulated by the class *Tape*. Every *Tape*-object simulates a tape cell, and has a single-valued property *left*, pointing to its left neighbor cell. Of course, at any instant of time, *Tape* contains only a finite number of cells, but can grow as needed during the computation. *Tape* is initialized to a single cell as follows:

$$Tape[\,] \Leftarrow \mathbf{true}.$$

Tape cells can contain objects in the input instance I as well as letters from the finite alphabet. Therefore, to represent the possible tape symbols in a uniform way, we populate a "generic" class of *Symbol* objects. For each class name C of S, we create a *Symbol* object for each object of class C in the input instance I as follows:

sorts $x : C$; $z : Symbol$

$Symbol[is : x] \Leftarrow \mathbf{true}(x)$

Moreover, we create a *Symbol* object for each alphabet letter ℓ as follows:

sorts $l : \tilde{\ell}$

$\tilde{\ell}[\ ] \Leftarrow \mathbf{true}$;
$Symbol[is : l] \Leftarrow \mathbf{true}(l)$

In order to be able to distinguish easily between *Symbol* objects representing input objects and *Symbol* objects representing alphabet letters, we mark the former using for each class name C of S the following edge addition:

sorts $z : Symbol$; $x : C$

$object(z, z) \Leftarrow \exists x : is(z, x)$

(Note that this construction of the class *Symbol* illustrates how *variant types*, also known as *union types*, can be represented using OBQL.)

The actual contents of the tape cells are represented by the class *Content*, which has single-valued properties labeled *cell* and *contains*. Every *Content*-object represents a pair $[cell : t, contains : x]$, meaning that cell t contains x. Here, t is a *Tape*-object and x is a *Symbol* object. Since a simulation of M_V is to be performed for every ordering of I and every value v, every *Content*-object has two more properties, named *order* and *val*. The *order*-edge points to an *Order*-object and the *val*-edge points to the K-object representing the value.

The construction of the initial configuration of the tape, containing the input (I, v) for each simulation is shown in detail in Appendix A2.

Before we are ready to start the simulation of the domTM computations on the input tapes just prepared, we need to initialize some bookkeeping information. Each computation step will be "timestamped" using a class *Clock*, organized as an ever-growing stack. All *Content* objects that represent the input configuration of the tape are linked to time zero. We also need a class *Status* with properties *order*, *val*, *state*, *register*, *cell*, and *time*, with the obvious semantics, to keep track of the status of the different computations at each step.

The simulation of the actual domTM transitions now happens in one big while-loop. At the beginning of each iteration, the computations that are still active are marked with their state information. Next, the computations that come to an end are marked. After that, a new *Clock* object is created, provided there are still active computations left. The loop body is completed with a number of statements performing the actual configuration changes for each of the state triples on which the domTM transition function is defined. All the implementation details are provided in Appendix A3.

This concludes the proof of Theorem 3.7. ■

4 Duplicate-free representations

In this section, we complement the completeness property of OBQL established in the previous section with a result showing that OBQL is no longer complete if duplicate-free representations are required.

Let τ be a complex value type. Under a class assignment α with $\alpha(\tau) = C$, objects of class C in a given instance represent values of type τ. If two objects represent the same value, they are called *duplicates with respect to* τ. Definition 3.5 allows duplicates; however, duplicate-free representations are often useful, as argued in the Introduction and further illustrated by the following example.

Consider a scheme containing class names *Student* and *Course*, with an edge from *Student* to *Course* labeled by the multi-valued property name *takes*. In an instance over this scheme, there will be objects labeled *Student*, i.e., students, and objects labeled *Course*, i.e., courses. Each student is connected to the courses he takes by edges labeled *takes*. Hence, the class *Student* represents the collection V of the sets of all courses taken by some student. However, this representation is not duplicate-free since different students might well take exactly the same courses.

Of course, we do not want to disallow duplicates in this situation. Nevertheless, it might be desirable to also have a representation of V which is duplicate-free. Assume we have such an additional class, *Set*, with in the scheme an edge from *Set* to *Course* labeled *contains*. In the instance, each *Set*-object is linked via *contains*-edges to precisely the courses of a set in V. Furthermore, class *Set* does not contain duplicate objects w.r.t. $\{contains : Course\}$. We can then derive a new single-valued property of students, named *takes_set*, by the following edge addition operation:

sorts $s : Student$; $z : Set$; $c : Course$

$takes\_set(s, z) \Leftarrow \forall c : takes(s, c) \leftrightarrow contains(z, c)$.

After this operation, each student is linked to the unique *Set*-object representing the set of courses which that student takes. Queries concerning the equality of sets of courses can now be answered very efficiently. To test

whether students take exactly the same courses, it suffices to check whether they are linked to the same *Set*-object.

The preceding discussions motivates the following question: is it possible to produce this duplicate-free class *Set* by means of an OBQL program? More formally, is the complex value query, mapping a given student-course database to the complex value collection of type $\{contains : Course\}$ consisting of the sets of all courses taken by some student, expressible in OBQL *without duplicates?* We will prove in the following that the answer is negative. This will also show that our proof of Theorem 3.7, which relies on the allowance of duplicates if set values are involved, cannot be improved in this respect.

Actually, we will prove that even a much simpler complex value query is already inexpressible without duplicates. Let S be a scheme and C a class name in S. Let e be a multi-valued property name. For a fixed $m \geq 2$, the *m-combinations query* of type $S \to \{e : C\}$ maps a given instance I over S to the subset of the powerset of class C in I consisting of all sets of cardinality m.

The proof will exploit the observation made at the end of Section 2 that every OBQL program can be simulated by another program operating under the weak semantics for object creation. Hence, without loss of generality, *we will assume weak semantics for object creation for the remainder of this section.* Unlike as we did in Section 2, we will no longer explicitly indicate weak object creation with '$\Leftarrow_{\text{weak}}$'.

This assumption allows us to associate untyped, nested tuple values to created objects. More concretely, let P be an OBQL program, let I be an instance to which P is applied, and let J be the result of this application. We associate to each object $o \in J$ a value $val(o)$ in an inductive manner, as follows:

Definition 4.1

- *If o is in I, then $val(o) := o$.*
- *If o is not in I, then o must have been created during the application of P on I, by an object creation operation of the form:*

  $$K[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \Phi(x_1, \ldots, x_n)$$

  Then o is created in function of a tuple $(o_1, \ldots, o_n)$ of objects in J, and we define $val(o) := [e_1 : val(o_1), \ldots, e_n : val(o_n)]$.

The following important property follows readily from the weak semantics for object creation:

Lemma 4.2 *Two different objects in the same class have a different associated value.*

In the sequel, it will be convenient to identify an object o in J with the pair $[B, val(o)]$, where B is the name of the class of o. (Interestingly, this identification is analogous to logic programming-based approaches to object creation [KW93]. We will illustrate this analogy further in Section 6.2.)

A permutation of the objects of an instance I can be canonically extended to a permutation of the complex values over I. With I and J as above, the identification just made then allows us to observe that OBQL is *BP-bounded* in the sense of [CH80].

Lemma 4.3 *If f is an automorphism of I, then f is also an automorphism of J.*

The proof proceeds by a straightforward induction and is left to the reader.

We are now ready for:

Theorem 4.4 *The m-combinations query is not expressible in OBQL without duplicates.*

Proof. Let S be the scheme consisting of one single class name C and no edges. Setting $n := m + 2$, let I_n be the instance over S with objects $1, \ldots, n$. So, I_n is a discrete graph of n isolated nodes. As a consequence, every permutation $\{1, \ldots, n\}$ is an automorphism of I_n.

For the sake of contradiction, suppose there is an OBQL program P expressing the m-combinations query. So, for some class name K, class K in $P(J_n)$—which we will denote by K_n—is a duplicate-free representation of the collection of m-combinations over $\{1, \ldots, n\}$. For each $o \in K_n$, there are exactly m edges labeled e leaving o, arriving in objects of I_n. We will denote the set of these m objects by $set(o)$. The set of all objects (of I_n) appearing in $val(o)$ is denoted by $base(o)$.

We next observe that for any object o in K_n, the permutations of I_n leaving $set(o)$ invariant are precisely those leaving $val(o)$ invariant. Indeed, assume f is a permutation of I_n such that $f(set(o)) = set(o)$. Since f is an

automorphism of I_n, f is also an automorphism of J_n, by Lemma 4.3. Hence, $f(o)$ is an object in K_n with $set(f(o)) = f(set(o))$. But $f(set(o)) = set(o)$, so, since K_n does not contain duplicates, $f(o) = o$. By the identification $o = [K, val(o)]$, we thus conclude that $f(val(o)) = val(o)$. Conversely, assume f is a permutation of I_n such that $f(val(o)) = val(o)$. But then $f(o) = o$ and, since again f is an automorphism of J_n, $f(set(o)) = set(f(o)) = set(o)$.

On the other hand, the permutations leaving $val(o)$ invariant are clearly those that are the identity on $base(o)$. As a result, the permutations that are the identity on $base(o)$ are precisely those leaving $set(o)$ invariant. The number of the former is $(n-b)! = (m+2-b)!$, where b is the cardinality of $base(o)$, while the number of the latter is $(n-m)!m! = 2m!$. We thus have $(m+2-b)! = 2m!$ and distinguish between the following cases:

- If $b \geq 2$ then $m+2-b \leq m$ and thus $(m+2-b)! \leq m! < 2m!$ which is false.
- If $b = 1$ then $m+2-b = m+1$ and thus $(m+1)! = 2m!$ or $m = 1$ which is false.
- If $b = 0$ then $m+2-b = m+2$ and thus $(m+2)! = 2m!$ or $(m+1)(m+2) = 2$ which is only possible if $m = 0$ which is false.

We have thus arrived at the desired contradiction. ■

It now follows that the student-courses query, which we originally considered, is not expressible in OBQL without duplicates either. For, if it were, then the m-combinations query would be expressible without duplicates as well, contradicting the above theorem. To see the latter claim, consider the simple m-ary object creation

$$K[e : x_1, \ldots, e : x_m] \Leftarrow \bigwedge_{1 \leq i < j \leq m} x_i \neq x_j,$$

where variables x_i are variables of sort C. This operation expresses the m-combinations query *with* duplicates, namely, $m!$ duplicates per m-combination. Since we can view the C-objects as courses and the K-objects as students, the ability to create a unique object for each set of all courses taken by some student would allow us to represent the m-combinations without duplicates.

5 The abstraction operation

Knowing from the previous section that certain complex value queries are not expressible in OBQL without duplicates, in the present section we investigate means to extend OBQL so that the problem goes away.

The two examples we gave in the previous section—the student-courses query and the m-combinations query—are complex value queries whose result type is a *set* type of depth 1. Actually, the duplicate-free representation problem does not exist for *tuple* types of depth one. Indeed, it is easy to express the extent of a depth 1 tuple type without duplicates, as shown in Lemma 3.9, so that the proof of Theorem 3.7, specialized to complex value queries resulting in depth 1 tuples, yields expressibility without duplicates. Furthermore, suppose we would define an extension of OBQL for which a duplicate-free version of Theorem 3.7 would hold for complex value queries whose result type is a set type of depth 1. Then by an obvious bottom-up construction this result would generalize to types of higher depth.

Hence, the "duplicate elimination" problem exists only for collections of depth 1 set values. There are at least two natural approaches to extending OBQL to deal with this problem. A first approach, pursuing an analogy with the tuple case for which we just observed that the problem does not exist, is to add an operation for expressing the extent of a (depth 1) set type without duplicates. We will call this operation the *powerset* operation. A second approach, inspired by the fact that OBQL is already able to express all reasonable complex value queries *with* duplicates, is to add an operation for producing a duplicate-free representation from another (not necessarily duplicate-free) representation. We will call this operation the *abstraction* operation. We will now develop these two approaches in more detail.

Let S be a scheme, C a class name in S, and I an instance over S. We define:

Definition 5.1 *The* powerset *operation*

$$\textbf{powerset } K\{e : C\}$$

applied to I results in the addition, for each subset Z of class C, of one new object o with label K together with edges (o, e, o') to all $o' \in Z$.

Now assume furthermore that (C, a, B) is a multi-valued edge in S. For simplicity, we assume that there is only one edge labeled a leaving C in the scheme. The relation "... and ... are duplicates w.r.t. $\{a : B\}$" is an equivalence relation which induces a partition in equivalence classes on the objects in class C of I. We call these the *equivalence classes with respect to* $\{a : B\}$. We then define:

Definition 5.2 *The* abstraction *operation*

$$\textbf{abstr}\ K[b] \Leftarrow C/a$$

applied to I results in the addition, for each equivalence class Z w.r.t. $\{a : B\}$, of one new object o with label K together with edges (o, b, o') to all $o' \in Z$.

We can now extend OBQL by allowing powerset operations as basic statements in the language, besides object creation and edge addition. The thus extended language will be called OBQL + **powerset**. Similarly, we obtain the language OBQL + **abstr**.

As an example, the student-courses query can be expressed without duplicates in OBQL + **powerset** as follows.

sorts $k : K$; $s : \mathit{Student}$; $c : \mathit{Course}$; $z : \mathit{Set}$

powerset $K\{e : \mathit{Course}\}$;
$\mathit{Set}[a : k] \Leftarrow \exists s : \forall c : \mathit{takes}(s, c) \leftrightarrow e(k, c)$;
$\mathit{contains}(z, c) \Leftarrow \exists k : a(z, k) \wedge e(k, c)$

As another example, the m-combinations query can be expressed without duplicates in OBQL + **abstr** as follows.

$K'[a : x_1, \ldots, a : x_m] \Leftarrow \bigwedge_{1 \leq i < j \leq m} x_i \neq x_j$;
abstr $K[e] \Leftarrow K'/a$

From the two above examples and the inexpressibility results of the previous section, we immediately obtain:

Proposition 5.3 *The languages OBQL+***powerset** *and OBQL+***abstr** *are strictly more expressive than OBQL.*

As already implicitly suggested in the preceding discussion, the expressive power of duplicate elimination as provided by the abstraction operation equals precisely the expressive power of duplicate-free powerset construction. Before we prove this formally, we introduce a technique and a lemma which will be used in the proof and which seem sufficiently interesting in their own right to explain them independently.

Recall from Section 2 that objects and edges with "auxiliary" labels are only temporary and are omitted from the end result of an application of a program to an instance. However, it is often convenient to be able to ignore certain temporary objects not only at the end, but also during the computation. More specifically, let T be an auxiliary class name, and let $\Phi(t)$ be a formula determining which objects of class T are no longer needed (t is a variable of sort T). We mark these objects by attaching to them a loop-edge labeled *old* using the edge addition:

$$old(t, t) \Leftarrow \Phi(t).$$

The thus marked objects can then be ignored by using formulas of the form $\neg old(t, t) \wedge \Phi'(t, \ldots)$. (This technique has been previously used in the literature [AV90, Theorem 2.4.1].)

The abstraction operation as defined above works on all the objects of a class C. It is often convenient however to work only on a subset of the class, determined by some formula $\Phi(x)$, with x a variable of sort C. We will write this generalized version of the abstraction operation, which we will call the *qualifying* version, as

$$\textbf{abstr}\ K[b] \Leftarrow C/a \mid \Phi.$$

Qualifying abstraction is a useful shorthand but does not increase the expressive power, as is shown next:

Lemma 5.4 *OBQL* + **abstr** *is equivalent to OBQL + qualifying* **abstr**.

Proof. Given a program in OBQL+qualifying **abstr**, an equivalent program in OBQL + **abstr** can be obtained by replacing each statement of the form

$$\textbf{abstr}\ K[b] \Leftarrow C/a \mid \Phi$$

with the following statements.

sorts $z : K$; $z' : K'$; $x : C$

abstr $K'[b'] \Leftarrow C/a$;
$K[b'' : z'] \Leftarrow \neg old(z', z') \wedge \exists x : b'(z', x) \wedge \Phi(x)$;
$b(z, x) \Leftarrow \exists z' : \neg old(z', z') \wedge b''(z, z') \wedge b'(z', x) \wedge \Phi(x)$;
$old(z', z') \Leftarrow \mathbf{true}(z')$

■

As illustrated in the above proof, the *old*-marking technique is useful to ensure that a simulation works in case the simulated statement would occur in the body of while-loop, since in this case the auxiliary class names (K' in the above) need to be reused. We point out that one can also define a qualifying version of the powerset operation, constructing the powerset only of the subset of class C consisting of those objects x satisfying a formula $\Phi(x)$. The analogue of Lemma 5.4 for powerset can then be proved analogously.

We are now ready for:

Theorem 5.5 *OBQL* + **powerset** *is equivalent to OBQL* + **abstr**.

Proof. Given a program P in OBQL + **powerset**, an equivalent program in OBQL + **abstr** can be obtained by the following replacement procedure. Consider a statement in P of the form

$$\mathbf{powerset}\ K'\{e' : C\}.$$

Recall the OBQL program given in the proof of Lemma 3.10 for representing the extent of $\{e : C\}$ by a class K (albeit with duplicates); we will call this program P_{dup}. Assuming, without loss of generality, that K is not used in P, we can thus simulate a powerset operation by an application of P_{dup} followed by an abstraction operation to eliminate the duplicates. Formally, we replace the above powerset operation with the following statements.

sorts $z : K$; $z' : K'$; $z'' : K''$; $x : C$

(insert program P_{dup} here);
abstr $K'[b] \Leftarrow K/e \mid \neg old(z, z)$;
$e'(z', x) \Leftarrow \exists z'' : \neg old(z'', z'') \wedge b(z', z'') \wedge e''(z'', x)$;
$old(z, z) \Leftarrow \mathbf{true}(z)$;

Notice the use of qualifying abstraction.

Conversely, let P be a program in OBQL + **abstr**. An equivalent program in OBQL + **powerset** can be obtained by replacing each abstraction statement as defined in Definition 5.2, of the form

$$\textbf{abstr}\ K[b] \Leftarrow C/a,$$

with the following statements.

sorts $z : K$; $z' : K'$; $x, y : C$; $v : B$

powerset $K'\{e : C\}$;
$K[b' : z'] \Leftarrow \neg old(z', z') \wedge \exists x : \forall y : e(z', y) \leftrightarrow (\forall v : a(x, v) \leftrightarrow a(y, v))$;
$b(z, x) \Leftarrow \exists z' : \neg old(z', z') \wedge b'(z, z') \wedge e(z', x)$;
$old(z', z') \Leftarrow \textbf{true}(z')$

The first statement creates a unique object for each subset of class C; the second statement then selects those that actually represent an equivalence class w.r.t. $\{a : B\}$, as required for the abstraction operation. ■

Corollary 5.6 *Every generic computable complex value query can be expressed in OBQL +* **abstr** *without duplicates.*

Although abstraction and powerset are equivalent from an expressiveness point of view, they are not from a computational point of view. The result of an application of the powerset operation can have exponential size; in contrast, the result of an application of the abstraction operation has linear size, and can in fact be computed efficiently, as we will show next.

Let S be a scheme and C a class name in S such that there is an edge $(C, \leq, C)$ in S. An instance I over S is called *C-ordered* if the set of pairs (o, p) of objects of class C for which there is an edge $(o, \leq, p)$ in I is a total order on class C. Although we know from Proposition 5.3 that abstraction is in general not expressible in OBQL, it *is* easily expressible in the special case of ordered databases, since in this case we can designate a unique representative for each equivalence class of duplicates by taking the smallest element. Formally, we have:

Theorem 5.7 *On C-ordered instances, abstraction over C is expressible in OBQL without using while-loops.*

Proof. The abstraction

$$\textbf{abstr}\ K[b] \Leftarrow C/a$$

is expressible as follows.

sorts $x, y : C$; $v : B$; $z : K$

$duplicates(x, y) \Leftarrow \forall v : a(x, v) \leftrightarrow a(y, v)$;
$K[b : x] \Leftarrow \forall y : duplicates(x, y) \rightarrow x \leq y$;
$b(z, y) \Leftarrow \exists x : b(z, x) \wedge duplicates(x, y)$

■

Corollary 5.8 *Abstraction can be computed in polynomial time on a Turing machine.*

Proof. A Turing machine, when presented an instance on its input tape, effectively has access to the order on the objects of the instance in which they are written on the tape. Since every OBQL program that does not use while-loops can be implemented on a polynomial-time Turing machine in a straightforward manner, the claim follows from Theorem 5.7. ■

We end this section with a result indicating the generality of the abstraction operation as a means to perform duplicate elimination. As defined in Definition 5.2, abstraction can be viewed as a quotient construction w.r.t. the equivalence relation "... and ... are duplicates." However, the definition does not rely in any way on this particular equivalence relation. This inspires us to define a generalization of the abstraction operation as follows.

Let S be a scheme, C be a class name in S, and $V_{\simeq}$ be a total generic computable complex value query of type $S \rightarrow [e_1 : C, e_2 : C]$ such that for each instance I over S, $V_{\simeq}(I)$—being a collection of binary tuples (ordered pairs) of objects of class C—is an equivalence relation on class C. The *generalized abstraction*

$$\textbf{abstr}\ K[b] \Leftarrow C/V_{\simeq}$$

applied to an instance I creates a unique new object for each equivalence class w.r.t. $V_{\simeq}(I)$, similarly to ordinary abstraction.

We next show that generalized abstraction can be easily expressed in terms of ordinary abstraction:

Theorem 5.9 *Generalized abstraction is expressible in OBQL +* **abstr**.

Proof. Consider a generalized abstraction operation of the form just described. By Theorem 3.7, there is an OBQL program $P_{\simeq}$ expressing $V_{\simeq}$. Let *Equiv* be the resulting class of this program. The generalized abstraction is now expressed as follows.

sorts $x, y : C$; $w : Equiv$

(insert program $P_{\simeq}$ here);
$a(x, y) \Leftarrow \exists w : e_1(w, x) \wedge e_2(w, y)$;
abstr $K[b] \Leftarrow C/a$

■

6 Discussion

We conclude this paper with a discussion on some of the ramifications of the work presented here.

6.1 Complex values as first-class citizens

A variety of formal object-based query languages with object creation capabilities have recently been proposed in the literature. Some of these languages are based on a data model were complex values are first-class citizens (e.g., [AK89]) so that they do not need the abstraction or the powerset operation. Unfortunately, whether or not a certain concept is "first-class" in one data model or another is not formally defined, and in fact often subject of considerable debate. We can clearly illustrate the point however by looking at two specific models: the relational model, and the nested relational model.

The nested relational data model extends the relational model in that tuple components need not be atomic, but can be relations in turn [PDGG89, Chapter 7]. In this sense, complex values are first-class citizens in the nested relational model. There is also the nested relational algebra, which augments the relational algebra with two operators, **nest** and **unnest**, to manipulate these complex values. Although we have presented our results in the context of a general object-base data model (since the problems which we have considered were motivated by object-oriented applications) the whole theory

has an equivalent relational counterpart. We will not elaborate this claim, since data model mappings from object-based to relational and vice versa are well understood (e.g., [HY90]). We will however illustrate the relational counterparts of object creation and abstraction.

Object creation can be achieved in the relational model by augmenting the relational algebra with the **new** operator. When applied to a relation R, this operator extends R with a new column containing a unique new identifier for each tuple in R. For example,

$$\mathbf{new}\left(\begin{array}{|cc|}\hline a & b \\ a & c \\ b & c \\ \hline\end{array}\right) = \begin{array}{|ccc|}\hline a & b & \alpha \\ a & c & \beta \\ b & c & \gamma \\ \hline\end{array}.$$

If we add the **new** operator to an extension of the relational algebra with while-loops (such as RQL [CH82]), we obtain a relational equivalent of the language OBQL considered in the present paper.

What is the relational counterpart of the abstraction operation? One way to define it is in the spirit of the student-courses query of Section 4. When applied to a binary relation R, interpreted as a student-course relation, the result is another binary relation, interpreted as a set-membership relation, where each set of all courses taken by some student is represented by a unique new identifier. For example,

$$\mathbf{abstr}\left(\begin{array}{|cc|}\hline \text{john} & \text{math} \\ \text{john} & \text{cs} \\ \text{mary} & \text{math} \\ \text{mary} & \text{cs} \\ \text{ellen} & \text{arts} \\ \hline\end{array}\right) = \begin{array}{|cc|}\hline \text{math} & \alpha \\ \text{cs} & \alpha \\ \text{arts} & \beta \\ \hline\end{array}.$$

By Proposition 5.3 and the equivalence of RQL + **new** and OBQL, relational **abstr** is not expressible in RQL + **new**.

However, if we now pick up the original motivation for the current few paragraphs, and move from the relational model to the *nested* relational model, it turns out that **abstr** becomes expressible. To do this, we add the **nest** and **unnest** to RQL, obtaining NRQL, and next NRQL + **new**, by extending the **new** operator canonically to work on nested relations. Relational abstraction is expressible in NRQL + **new** simply as

$$\mathbf{abstr}(R) = \mathbf{unnest} \cdot \mathbf{new} \cdot \pi_2 \cdot \mathbf{nest}(R),$$

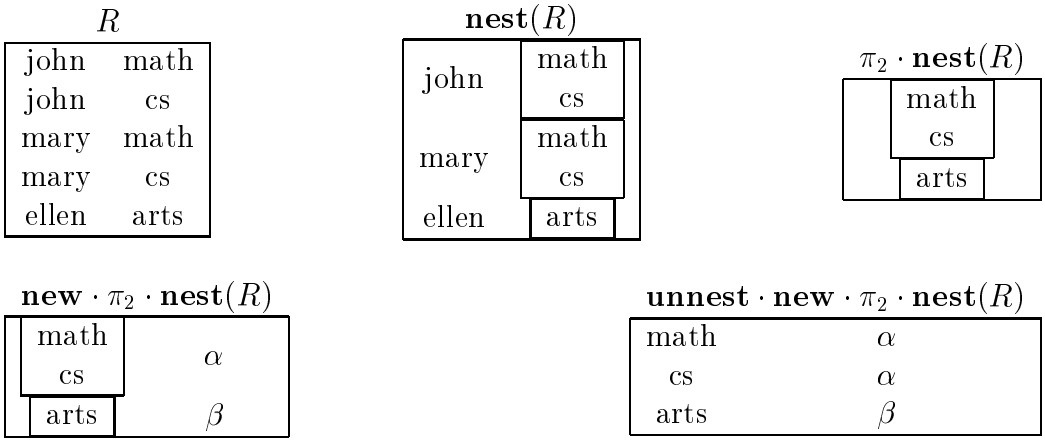

Figure 1: Computing **abstr**(R) in NRQL + **new**.

as illustrated in Figure 1. To summarize, we hope it has become clear that *abstraction is the operational realization of the ability to represent complex values (most notably set values) as first-class citizens in the data model.*

6.2 Related work

Typically, object-oriented data models of the "pure" variety do not support complex values on a first-rate basis (see the Introduction). As a result, many object-based query languages proposed in the literature are essentially equivalent to OBQL and hence have to deal with the problem of duplicates. Thereto, they often provide an extra feature in much the same way as the abstraction operation can be added as an extra feature to OBQL.

Two languages which do *not* provide such an extra feature are ILOG [HY90] and O-Logic [KW93]. Both are essentially extensions of Datalog [Ull88] with Skolem functions for object creation. For example, the analogue of the OBQL object creation

$$K[e_1 : x_1, \ldots, e_n : x_n] \Leftarrow \Phi(x_1, \ldots, x_n)$$

in these languages is a rule of the form

$$K(f^K_{e_1,\ldots,e_n}(x_1, \ldots, x_n), x_1, \ldots, x_n) \leftarrow \Phi(x_1, \ldots, x_n)$$

where $f^K_{e_1,\ldots,e_n}$ is an uninterpreted function symbol. Under this analogy, each thus created object o is identified with a term of the form $f^K_{e_1,\ldots,e_n}(t_1,\ldots,t_n)$, where the t_i are other terms which further identify other objects o_i. Note that this identification corresponds exactly to Lemma 4.2, where such an object o was shown to be identifiable with the pair $[K, val(o)]$. By Definition 4.1, $val(o)$ will equal $[e_1 : val(o_1), \ldots, e_n : val(o_n)]$, whence we can conclude that the two identifications $f^K_{e_1,\ldots,e_n}(t_1,\ldots,t_n)$ and $[K, val(o)]$ have precisely the same information content and differ only in syntax. The approaches to object creation in OBQL and in ILOG or O-Logic are therefore entirely equivalent. In particular, the latter languages, like OBQL, cannot eliminate duplicates.

However, F-Logic [KLW93], the successor of O-Logic, does provide the possibility to specify additional equality axioms to enforce two terms to be equal in the underlying logical interpretation. For example, Kifer, Lausen and Wu [KLW93] present an F-Logic program for expressing the powerset, using an interated pairing construction similar in spirit to the OBQL program used in the proof of Lemma 3.10. By adding to this program an axiom of the form $pair(x, pair(y, z)) = pair(y, pair(x, z))$, the duplicates are equated and thus eliminated. In this way, F-Logic can express duplicate-free powerset and hence (by Theorem 5.5) also abstraction.

Two languages which provide an explicit abstraction or duplicate elimination operation are GOOD [GPVG94] and the algebra of Shaw and Zdonik [SZ90]. The LDM algebra [KV93] provides an explicit powerset operation.

Languages which provide the ability to non-deterministically choose an element from a set can also express abstraction, as they can choose a unique representative for each equivalence class of duplicates. An example is the language DL [AV91]. Another way to see this is to observe that by repeated non-deterministic choices, a total order can be constructed so that (by Theorem 5.7) abstraction can be expressed. Consequently, languages providing the possibility to perform an action on each element of a set in some non-deterministic order can express abstraction. Examples are the language TL [AV90] and some of the languages considered in [HS89].

6.3 Duplicate elimination versus copy elimination

In this paper, we have concentrated on *complex value* queries, as defined in Section 3. However, as we observed in Section 2, programs in OBQL (and

OBQL + **abstr** for that matter) in general have the effect of augmenting a database with derived information in the form of new objects and edges, not necessarily representing a collection of complex values of some fixed type as is the case for complex value queries. This effect is, strictly speaking, non-deterministic, since the identity of the newly created objects is unimportant (of course this non-determinism is of an infinitely more restricted nature than the arbitrary non-determinism mentioned at the end of Section 6.2). Furthermore, just as with complex value queries, we want the effect to be *generic*, i.e., to be invariant under permutations of the universe of objects.

The formal concept of general object-based query alluded upon in the previous paragraph was introduced by Abiteboul and Kanellakis [AK89] as follows. Let S_{in} and S_{out} be two schemes such that S_{out} properly contains S_{in}. Denote the set of all instances over a scheme S by $\text{inst}(S)$.

Definition 6.1 *An* object-based query of type $S_{\text{in}} \to S_{\text{out}}$ *is a binary relationship* $Q \subseteq \text{inst}(S_{\text{in}}) \times \text{inst}(S_{\text{out}})$ *satisfying:*

1. *Q is recursively enumerable;*
2. *If $Q(I, J)$ then J restricted to S_{in} equals I;*
3. *Q is invariant under permutations of the universe of objects;*
4. *If $Q(I, J_1)$ and $Q(I, J_2)$ then there exists a permutation f of the universe of objects which is the identity on the objects in I such that $f(J_1) = J_2$.*

Since the abstraction operation is a query in the above sense, not every query can be expressed by an OBQL program (by Corollary 5.8). In other words, OBQL is not *complete* w.r.t. the object-based queries. Is OBQL + **abstr** complete? This question was studied by Abiteboul and Kanallakis [AK89] in the context of the IQL language, which is essentially equivalent to OBQL + **abstr**. They showed that IQL is only complete *up to copies*.

Formally, let $\bar{J}$ and J be instances over S_{out}, and let I be the restriction of J to S_{in}. We define:

Definition 6.2 *$\bar{J}$ is an instance* with copies of *J if there are n instances $J_1, \ldots, J_n$ over S_{out} such that the following holds:*

1. *For each i, there is a permutation f of the universe of objects which is the identity on the objects in I such that $f(J_i) = J$;*

2. *For each $i \neq j$, J_i and J_j are disjoint outside S_{in};*

3. *$\bar{J} = J_1 \cup \cdots \cup J_n$.*

Now let $\bar{Q}$ and Q be object-based queries of type $S_{\text{in}} \to S_{\text{out}}$. We further define:

Definition 6.3 *$\bar{Q}$ equals Q* up to copies *if $Q(I, J)$ implies $\bar{Q}(I, \bar{J})$ for some instance $\bar{J}$ with copies of J, and conversely, $\bar{Q}(I, \bar{J})$ implies $Q(I, J)$ for some J such that $\overline{J}$ is an instance with copies of J.*

Abiteboul and Kanellakis proved:

Theorem 6.4 ([AK]) *Every object-based query can be expressed up to copies in IQL.*

The notion of copies just defined should of course not be confused with the notion of duplicates introduced in Section 4. Copies are complete instances which are isomorphic with respect to some reference instance I, while duplicates are individual objects which are logically indistinguishable on the basis of their local properties. Nevertheless, there turns out to be a connection between the two notions. Recall from Theorem 3.7 that every complex value query can be expressed in OBQL, albeit possibly with duplicates. Using this result, we can show that not only OBQL + **abstr**, being equivalent to IQL, but already OBQL alone is complete up to copies:

Theorem 6.5 *Every object-based query can be expressed up to copies in OBQL.*

Proof. We will outline the argument for the particular case that S_{in} consists of a single class name A and no edges, and S_{out} additionally has class name B and edges (B, a, A) and (B, b, B). The general case is analogous. Let Q be a query of type $S_{\text{in}} \to S_{\text{out}}$. Hence, the effect of Q on an instance over S_{in} (consisting of a set of A-labeled objects) is to add a number of new B-labeled objects together with a-labeled edges from the new to the old objects and b-labeled edges between the new objects. We want to show that Q is expressible up to copies in OBQL.

Thereto, we associate a complex value query Q^{val} to Q as follows. Let S'_{in} be the scheme obtained from S_{in} by adding two classes, B and *Count*, and an edge (*Count*, *next*, *Count*). Given an instance I over S_{in} and two natural numbers n and m, denote by $I'_{n,m}$ the instance over S'_{in} obtained from I by adding n new objects labeled B, and adding m new objects labeled *Count* organized as a linear list by means of *next* edges. Furthermore, let τ be the complex value type

$$[a : \{[from : B, to : A]\}\ ,\ b : \{[from : B, to : B]\}].$$

Given an instance I over S_{in}, $I'_{n,m}$ (for some n and m) together with a value v of type τ over $I'_{n,m}$ describes an instance over S_{out} in the obvious way, which we denote as $J(I'_{n,m}, v)$. Finally, let M some fixed Turing machine which enumerates Q. Now the complex value query Q^{val} of type $S'_{\text{in}} \to \tau$ is defined as follows: for each I, n and m, $Q^{\text{val}}(I'_{n,m})$ results in the collection of values v for which M on input I outputs an instance J isomorphic to $J(I'_{n,m}, v)$ after m steps.

By Theorem 3.7, Q^{val} is expressible in OBQL. We can now write an OBQL program expressing Q up to copies as follows. On input I, the program visits all pairs of natural numbers (n, m) in succession (encoded using temporary objects in the well-known way) and tests at every stage whether $Q^{\text{val}}(I'_{n,m})$ is non-empty. If the test succeeds, we have in effect a collection of descriptions of instances J for which $Q(I, J)$. These descriptions can be collected in an instances with copies. The details are tedious but straightforward and left to the reader. ■

In retrospect, this proof of completeness up to copies is very close to the one Abiteboul and Kanallakis gave for IQL [AK]. However, we have arrived at it independently and from a somewhat different perspective. Furthermore, we have proved the result for OBQL, which shows that the abstraction operation (or, equivalently, duplicate-free set representation) is not needed to achieve completeness up to copies. This has two consequences as to the connection between the notions of copies and duplicates announced above.

First, it shows that elimination of copies is at least as strong than elimination of duplicates. Indeed, Theorem 6.5 implies that the abstraction operation, viewed as an object-based query, is expressible in OBQL up to copies. Actually, elimination of copies is known to be *strictly* stronger than elimination of duplicates; we will not elaborate on this but refer the interested

reader to [VVAG94] for an overview of the different notions of completeness for object-creating query languages.

Second, inspection of the Proof of Theorem 6.5 reveals that if OBQL + **abstr** is used, then Q^{val} can be expressed without duplicates so that the instance with copies will contain *less copies* than would be possible in OBQL. Hull and Yosikawa [HY90] have made similar observations in the context of the ILOG language, although they did not recognize duplicates and copies as distinct notions. Specifically, they claimed that ILOG is complete up to copies but cannot express the 2-combinations query (using the terminology of our Section 4) without duplicates. Comparing these findings with the results of Abiteboul and Kanellakis on IQL, they concluded that "IQL can eliminate more duplicates than ILOG." No rigorous proofs were presented.

6.4 Further research

To conclude this paper, we mention some further results which have been obtained in connection with the present work. Van den Bussche and Van Gucht [VV] have reexamined the problem of duplicate set values in a setting where a fixed cardinality bound on set values is known. One of their results is the following generalization of Theorem 4.4. The m-combinations query without duplicates, considered in Section 4, can be viewed as a cardinality-restricted version of the powerset operation defined in Section 5. We can add this operation to OBQL, obtaining OBQL + $\mathbf{powerset}_m$. Then except for $m = 0$ and $m = 3$, OBQL + $\mathbf{powerset}_{m+1}$ is strictly more expressive than OBQL + $\mathbf{powerset}_m$.

Van den Bussche et al. [VVAG94] have characterized the classes of all object-based queries that can be expressed in OBQL and OBQL + **abstr** *without copies*. It turns out that these classes occur naturally as consisting of those queries for which creation of new objects can be achieved by construction of untyped, nested tuple values (for OBQL + **abstr**, nested set values) over the objects in the input. We have seen that this is indeed possible in OBQL (see Definition 4.1 and following). Alternative algebraic characterizations, in the form of extra conditions imposed on object-based queries, have also been obtained.

Finally, there seems to be a connection between the notion of duplicates in object-based data models, as considered in this paper, and recent work on *bag databases* (bags generalize sets with duplicates). For example, Grumbach

and Milo [GM93] present results similar to ours concerning the equivalence between duplicate elimination and powerset construction in the context of bags. We conjecture that bag representation and manipulation can be naturally studied in the object-based framework set up in the present paper.

Acknowledgment

Dirk Van Gucht made valuable suggestions concerning the expressive power of abstraction. Marc Gyssens offered most helpful comments on previous drafts of this paper. We would also like to thank the referees for their suggestions concerning the presentation of our paper.

References

[AB88] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Rapport de recherche 846, INRIA-Rocquencourt, 1988.

[AFS89] S. Abiteboul, P. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In Clifford et al. [CLM89], pages 159–173.

[AK] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. Rapport de recherche 1022, INRIA-Rocquencourt, 1989. Full version of [AK89], to appear in *Journal of the ACM*.

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

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[Bee90] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990.

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

[CH82] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[CLM89] 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*. ACM Press, 1989.

[GM93] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Proceedings 12th ACM Symposium on Principles of Database Systems*, pages 49–58. ACM Press, 1993.

[GPVG94] 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), 1994.

[HK87] R. Hull and R. King. Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HS89] R. Hull and J. Su. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In Clifford et al. [CLM89], pages 147–158.

[HS93] R. Hull and Y. Su. Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, 47(1):121–156, 1993.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 455–468. Morgan Kaufmann, 1990.

[KL89] W. Kim and F.H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Frontier Series. ACM Press, Addison-Wesley, 1989.

[KLW93] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 93/06, Dept. Comp. Science, SUNY Stony Brook, 1993. To appear in *Journal of the ACM*.

[KV93] G. Kuper and M. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

[KW93] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, 1993.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In H. Boral and P.A. Larson, editors, *1988 Proceedings SIGMOD International Conference on Management of Data*, pages 424–433. ACM Press, 1988.

[PDGG89] J. Paredaens, P. De Bra, M. Gyssens, and D. Van Gucht. *The Structure of the Relational Database Model*, volume 17 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1989.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 16(10):140–173, 1981.

[SZ90] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proceedings Seventh International Conference on Data Engineering*, pages 154–162. IEEE Computer Society Press, 1990.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[VP91] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 291–299. ACM Press, 1991.

[VV] J. Van den Bussche and D. Van Gucht. The expressive power of cardinality-bounded set values in object-based data models. *Theoretical Computer Science*. Extended and revised version of [VV92]. To appear.

[VV92] J. Van den Bussche and D. Van Gucht. A hierarchy of faithful set creation in pure OODBs. In J. Biskup and R. Hull, editors, *Database Theorey—ICDT'92*, volume 646 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, 1992.

[VVAG92] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *Proceedings 33rd Symposium on Foundations of Computer Science*, pages 372–379. IEEE Computer Society Press, 1992.

[VVAG94] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creation database transformation languages. Manuscript, extended and revised version of [VVAG92], 1994.

Appendix: Proof of Theorem 3.7

A1. Ordering edges and the whole instance

Having created orderings of the different classes of objects in the input instance I, we also have to order the edges in I. This can be done by creating for each edge (B, e, C) in the scheme S a new class B_e_C, which we will call an *edge class*, using the following object addition:

sorts $x : B$; $x' : C$

$B\_e\_C[from : x, to : x'] \Leftarrow e(x, x')$.

In this way, for each e-edge (o, e, o') in the instance, where o is a B-object and o' is a C-object, an object representing the pair $[from : o, to : o']$ is created. The class B_e_C can now be ordered in the same way as the "ordinary" classes.

Using the created orderings of the classes and properties, we can now deduce the orderings of the whole instance as aggregates of the individual orderings. Let $\{C_1, \ldots, C_n\}$ be the set of all class names and edge-class names of S. Then we perform the object creation:

sorts $x_1 : List_{C_1}; \ldots; x_n : List_{C_n}$

$Order[order_{C_1} : x_1, \ldots, order_{C_n} : x_n] \Leftarrow Order_{C_1}(x_1, x_1) \wedge \cdots \wedge Order_{C_n}(x_n, x_n)$

A2. Constructing the initial configuration

The initial configuration of the tape contains the input (I, v) for each simulation. We first write the objects of I on the tape. For each class name C of S, we perform the following procedure. The procedure uses an auxiliary property *list* of *Content*-objects which will point to $List_C$-objects.

sorts $t, t' : Tape$; $c : Content$; $x : Symbol$;
$o : Order$; $v : K$; $y : C$;
$l : List_C$; $stop : Stop_C$; $z : \tilde{C}$

$Content[cell : t, order : o, val : v, list : l] \Leftarrow \neg\exists t' : left(t', t) \wedge order_C(o, l)$;
while change do
$Stop_C[\,] \Leftarrow \exists c : cell(c, t) \wedge \neg\exists t' : left(t', t) \wedge \exists l : list(c, l) \wedge \neg\exists x : head(l, x)$;
$contains(c, x) \Leftarrow \neg\exists stop \wedge \exists t : cell(c, t) \wedge \neg\exists t' : left(t', t) \wedge$
$\exists y \exists l : list(c, l) \wedge head(l, y) \wedge is(x, y)$;
$Tape[left : t] \Leftarrow \neg\exists stop \wedge \neg\exists t' : left(t', t)$;
$Content[cell : t, order : o, val : v, list : l, contains : x] \Leftarrow$
$\exists z : is(x, z) \wedge \neg\exists stop \wedge \exists t' : left(t, t') \wedge$
$\neg\exists t'' : left(t'', t) \wedge \exists c : cell(c, t') \wedge order(c, o) \wedge val(c, v) \wedge list(c, l)$;
$Tape[left : t] \Leftarrow \neg\exists stop \wedge \neg\exists t' : left(t', t)$;
$Content[cell : t, order : o, val : v, list : l] \Leftarrow$
$\neg\exists stop \wedge \exists t' : left(t, t') \wedge \neg\exists t'' : left(t'', t) \wedge$
$\exists c : cell(c, t') \wedge order(c, o) \wedge val(c, v) \wedge \exists l' : list(c, l') \wedge tail(l', l)$
od

Note that the last *Content*-objects created by the above procedure do not have a *contains*-property. These "void" objects are harmless and will not play a role in the further simulation.

To write the edges of I on the tape, we perform, for each edge-class name, an analogous procedure as for the ordinary class names. At last we write value v on the tape. If τ is the set type $\{e : C\}$, then the ordering of C is used. We omit the details.

A3. Bookkeeping and actual simulation

Before starting the simulation of the computations on the inputs just prepared, some bookkeeping information must be initialized. Each step of the simulation will be "timestamped" using a class *Clock*, organized as an ever-growing stack and initialized to a single object, standing for time zero. All *Content* objects that represent the input configuration of the tape are linked to time zero. So, we have the following object en property additions:

sorts $c : \mathit{Content}$; $cl : \mathit{Clock}$

$\mathit{Clock}[\,] \Leftarrow \mathbf{true}$;
$\mathit{time}(c, cl) \Leftarrow \mathbf{true}(c, cl)$

Furthermore, for each state q of M_V, we create a class $\tilde{q}$, consisting of a single object, using the operation $\tilde{q}[\,] \Leftarrow \mathbf{true}$.

The class *Status* with properties *order*, *val*, *state*, *register*, *cell* and *time* keeps track of the status of the different computations at each step. Initially, the state is the initial state q_0, and the cell is the first tape cell. The initial contents of the register does not matter so we leave it unspecified. We thus have the following initialization:

sorts $o : \mathit{Order}$; $v : K$; $z : \tilde{q}_0$; $t : \mathit{Tape}$; $cl : \mathit{Clock}$

$\mathit{Status}[\mathit{order} : o, \mathit{val} : v, \mathit{state} : z, \mathit{cell} : t, \mathit{time} : cl] \Leftarrow$
$\quad \mathbf{true}(o, v, z, cl) \wedge \neg\exists t' : \mathit{left}(t', t)$

The simulation of the actual domTM transitions now happens in one big while-loop. At the beginning of each iteration, the state information of the computations that are still active can be described by a triple (q, x, y), where $q \in Q$, $x \in \Sigma \cup \{\eta\}$, and $y \in \Sigma \cup \{\eta, \kappa\}$. This information, which is implicit

in classes *Status* and *Content*, is made explicit by marking each order-value pair whose associated computation is still active with a $\widetilde{qxy}$-object as follows. If $x, y \in \Sigma$ then we have the statement: respectively)

sorts $o : Order$; $v : K$; $cl, cl' : Clock$;
$s : Status$; $c : Content$; $z_x, z_y : Symbol$;
$z_q : \tilde{q}$; $xx : \tilde{x}$; $yy : \tilde{y}$

$\widetilde{qxy}[order : o, val : v, time : cl] \Leftarrow$
$\neg\exists cl' : previous(cl', cl) \wedge \exists s, c, z_q, z_x, xx, z_y, yy :$
$order(s, o) \wedge val(s, v) \wedge time(s, cl) \wedge$
$order(c, o) \wedge val(c, v) \wedge time(c, cl) \wedge$
$state(s, z_q) \wedge register(s, z_x) \wedge is(z_x, xx) \wedge$
$\exists t : cell(s, t) \wedge cell(c, t) \wedge contains(c, z_y) \wedge is(z_y, yy)$

If $x = \eta$ and $y \in \Sigma$, then x ranges over all objects in the input instance I. In this case, the clause $register(s, z_x)$ in the above statement must be replaced by $register(s, z_x) \wedge object(z_x, z_x)$. The cases $x = y = \eta$ and $x = \eta, y = \kappa$ are similar and left to the reader.

Next, the computations that come to an end are marked with a *Done*-object by including the following statement for each triple (q, x, y) on which the transition function is undefined:

sorts $s : Status$; $v : K$; $cl, cl' : Clock$;
$o : Order$; $z_q : \tilde{q}$; $z : \widetilde{qxy}$

$Done[order : o, val : v, state : z_q] \Leftarrow \exists cl : \neg\exists cl' : previous(cl', cl) \wedge$
$\exists z : order(z, o) \wedge val(z, v) \wedge time(z, cl)$

After that, a new *Clock*-object is created, provided there are still active computations left:

sorts $cl, cl' : Clock$; $o : Order$; $v : K$; $d : Done$

$Clock[previous : cl] \Leftarrow \neg\exists cl' : prev(cl', cl) \wedge$
$\exists o, v : \neg\exists d : order(d, o) \wedge val(d, v)$

The loop body is completed with, for each triple (q, x, y) on which the transition function *is* defined, three statements performing the actual configuration change. For example, if $\delta(q, \eta, y) = (q', \eta, \eta, L)$, we have:

sorts $o : Order$; $t, t' : Tape$; $cl, cl', cl'' : Clock$;
$s : Status$; $x : Symbol$; $v : K$;
$z : \widetilde{q\eta y}$; $z_{q'} : \widetilde{q'}$

$Content[order : o, val : v, cell : t, contains : x, time : cl] \Leftarrow$
$\exists cl' : previous(cl, cl') \wedge \neg\exists cl'' : previous(cl'', cl) \wedge \exists z, s :$
$order(z, o) \wedge val(z, v) \wedge time(z, cl') \wedge$
$order(s, o) \wedge val(s, v) \wedge time(s, cl') \wedge cell(s, t) \wedge register(s, x)$;
$Content[order : o, val : v, cell : t, contains : x, time : cl] \Leftarrow$
$\exists cl' : previous(cl, cl') \wedge \neg\exists cl'' : previous(cl'', cl) \wedge \exists z, s :$
$order(z, o) \wedge val(z, v) \wedge time(z, cl') \wedge$
$order(s, o) \wedge val(s, v) \wedge time(s, cl') \wedge \neg cell(s, t) \wedge$
$\exists c : order(c, o) \wedge val(c, v) \wedge time(c, cl) \wedge cell(c, t) \wedge contains(c, x)$;
$Status[order : o, val : v, state : z_{q'}, register : x, cell : t, time : cl] \Leftarrow$
$\exists cl' : previous(cl, cl') \wedge \neg\exists cl'' : previous(cl'', cl) \wedge \exists z, s :$
$order(z, o) \wedge val(z, v) \wedge time(z, cl') \wedge$
$order(s, o) \wedge val(s, v) \wedge time(s, cl') \wedge register(s, x) \wedge \exists t' : cell(s, t') \wedge left(t', t)$

The other cases are similar.

Finally, after the end of the loop, the accepted values v, i.e., those corresponding to computations that have halted in state q_a, are marked:

sorts $v : K$; $d : Done$; $z : \widetilde{q_a}$

$accept(v, v) \Leftarrow \exists d : val(d, v) \wedge state(d, z)$.