Deep Equality Revisited[*]

Serge Abiteboul Jan Van den Bussche

INRIA Rocquencourt[†]

Abstract

We revisit the notion of deep equality among objects in an object database from a formal point of view. We present three natural formalizations of deep equality: one based on the infinite value-trees associated with objects, one based on the greatest fixpoint of an operator on equivalence relations among objects, and one based on indistinguishability of objects using observations of atomic values reachable from the objects. These three definitions are then shown to be equivalent. The characterization in terms of greatest fixpoints also yields a polynomial-time algorithm for checking deep equality. We also study the expressibility of deep equality in deductive database languages.

1 Introduction

In object databases, objects consist of an *object identifier (oid)* and a *value*, typically having a complex structure built using the set and tuple constructor, in which both basic values and further oids appear. An intuitive way to think about an oid is thus as a reference to a complex value, so that such values can be shared. As a consequence, the actual "value" of an oid (be it a physical memory address, or a logical pointer) is of lesser importance. In

[*]Appeared in T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases*, pages 213–228, volume 1013 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[†]Projet Verso, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. E-mail: serge.abiteboul@inria.fr, vdbuss@uia.ac.be.

particular, the only comparison on oids that makes sense on a logical level is simply testing whether two given oids are in fact one and the same. In this way one can check whether some complex value is shared or not. However, in many applications, even this comparison is not really needed, since sharing is mostly an implementation issue and often need not be part of the application semantics.

It is thus of interest to see what happens when objects can only be distinguished by looking at their values, possibly dereferencing the oids appearing therein (and this recursively). Note that this corresponds to what is available in typical visual interfaces for browsing object databases (e.g., O_2Look in the O_2 system [P$^+$92]), where basic values (such as strings, numbers, or bitmap images) can be directly observed but where oids can only be inspected by dereferencing them and inspecting their associated complex value in turn. When two objects are indistinguishable in this manner, they are typically called *deep-equal*. The notion of deep equality is since long well-known in object-oriented programming and databases (e.g., [KC86, SZ90]), but a systematic study of its fundamental properties has not yet been carried out. It is our aim in this paper to contribute towards this goal.

We will look at three possible natural formalizations of deep equality, and show that they are all equivalent.

The first is inspired by the "pure value-based" model of object databases in terms of infinite trees, introduced in [AK89]. The complex value of an object can be viewed as a tree, the leafs of which are basic values or oids. By replacing each leaf oid by the tree corresponding to its value, and this recursively, we obtain the "unfolding" of the entire value structure than can be seen from the object by "following pointers in the forward direction only". This unfolding can be infinite when the instance contains cyclic references (which is often the case). Two objects can thus be called deep-equal if their associated, possibly infinite, value-trees are equal.

The second formalization is more abstract: deep equality can be defined as the coarsest equivalence relation among objects (extended to complex values in the natural way) satisfying the requirement that two objects are equivalent if and only if their values are. Deep equality can thus be viewed as the greatest fixpoint of an operator which maps equivalence relations to finer ones. This yields a polynomial-time algorithm for testing deep equality.

Our third formalization is inspired by the idea of indistinguishability discussed in the beginning of this introduction. We define a class of logical *observation formulas*, a subclass of any reasonable object calculus query language. Observation formulas can observe and compare basic values, can dereference oids, and can traverse paths in complex values. Thus, two objects can be defined to be deep-equal if they cannot be distinguished by any observation formula.

In this paper we also study the expressibility of deep equality in deductive database languages. Deep equality is readily expressible in the language of fixpoint logic. However, we show that deep equality is not expressible in the language of Datalog with stratified negation. It is expressible in this language on databases containing only tuple values of bounded width (or set values of bounded cardinality). Up to now, the only examples of queries known to be in fixpoint logic but not in stratified datalog were based on game trees (e.g., [Kol91]). We will show that these game-tree queries can also be understood in the context of deep equality, which might perhaps be more "natural" for some.

Denninghoff and Vianu [DV93] and, more recently, Kosky [Kos95] have also introduced a notion of "similarity" of objects, which corresponds to our second formalization of deep equality. Both [DV93] and [Kos95] noted the analogy with the infinite value-trees mentioned above. One of our contributions is to make this very precise. Also, Denninghoff and Vianu only considered tuple values, no set values. One might expect at first that the presence of set values would make the computational complexity of testing deep equality intractable; our results imply that even with set values it remains computable in polynomial time. We also point out that Kosky studied the indistinguishability of two entire database instances, rather than of two objects within one single instance as we do. Finally, deep equality is the object database analog of the notion of strong bisimilarity in transition systems, studied in the theory of communication and concurrency [Mil89].

This paper is organized as follows. In Section 2, we introduce the data model we will use. It is a standard object database model as used in, e.g., the O_2 system [KLR92]. In Section 3, we recall the infinite value-trees associated with objects. In Section 4, we give the fixpoint definition of deep equality, relate it to the infinite tree definition, and show how it can be computed in polynomial time. In Section 5, we characterize deep equality as indistinguishability by observation formulas. Finally, in Section 6, we study the expressibility of deep equality in deductive database languages.

2 Data model

In this paper, we consider an *object database* to be simply a collection of *objects*, where each object consists of an *identifier* and a *value*. The value of an object can be complex in structure and can contain references to (i.e., identifiers of) other objects. We do not consider database schemas and value types in this paper, since they are irrelevant to our purposes. The reader who wishes to apply our treatment to a setting with schemas and types will encounter no difficulties in doing so.

More formally, assume given two disjoint sets of *basic values* and *object identifiers (oids)*.

Given a set of oids O, the set of *values over O* is inductively defined as follows:

1. Each basic value is a value over O;

2. Each element of O is a value over O;

3. If v_1, ..., v_n are values over O, then the tuple $[v_1, \ldots, v_n]$ is a value over O;

4. If $v_1, \ldots, v_n$ are values over O, then the set $\{v_1, \ldots, v_n\}$ is a value over O.

An *object database* now consists of a finite set O of oids, together with a mapping ν assigning to each oid $o \in O$ a value $\nu(o)$ over O. The pair $(o, \nu(o))$ can be thought of as *the object o*.

Throughout the remainder of this paper, we will assume that value $\nu(o)$ of any object o in the database is either a basic value, a tuple consisting of basic values and oids, or a set consisting of basic values and oids. Hence, we do not consider objects whose value is simply another oid, or whose value is a complex value with nested structure. The first case is related to standard assumptions in the theory of infinite regular trees, as will become clear in the next section. The second case is for clarity of exposition only.

An object whose value is simply the identifier of another object can always be replaced by the latter object. Or alternatively, its value can be changed into a unary tuple having the identifier as its single component.

Complex values with nested structure can be simulated by introducing new objects. For example, an object α with the non-flat value

$$\nu(\alpha) = [\{1, 2, 3\}, \{3, 4\}]$$

can be simulated by introducing two new objects β and γ with flat values $\{1, 2, 3\}$ and $\{3, 4\}$, respectively, and changing $\nu(\alpha)$ to $[\beta, \gamma]$. In order to apply the treatment presented in the remainder of this paper to databases containing non-flat values, it suffices to think of such values as objects having the appropriate values.

To conclude this section, we introduce one last definition regarding the model:

Definition 1 A *tuple database* is a database is which no set values occur.

3 Objects and infinite trees

A flat tuple value $[v_1, \ldots, v_n]$ can be viewed as an ordered tree of depth at most one, where the root is labeled by the n-ary tuple constructor symbol $\times^n$, and the children of the root are labeled by v_1, ..., v_n, respectively. (Note that n may equal 0, in which case the tree consists of a single node labeled $\times^0$.) Similarly, a basic value v can be viewed as a trivial tree consisting of a single node labeled v.

Now assume we are working with a tuple database. So, the value of every object is either a basic value or a tuple value. In the tree corresponding to such a value, we can replace the leaf nodes labeled by oids by the trees corresponding to the values of the oids, obtaining a deeper tree. We can repeat this for the oids appearing in these values in turn. If we keep on repeating this process, it eventually stops if the database does not contain cyclic references. However, if there are cyclic references, the process can go on forever and yields a tree which is infinite. In both cases, we obtain a tree in which all leafs are labeled by basic values; there are no longer any leafs labeled by oids. We call such trees *ground trees*.

Example 1 For example, consider a part-subpart database, where each object is a part having a type (a basic value) and a list of subparts (a tuple of oids):

$$\begin{array}{rcl}
\nu(o_1) & = & [\text{car}, \ell_1] \\
\nu(\ell_1) & = & [o_2] \\
\nu(o_2) & = & [\text{engine}, \ell_2] \\
\nu(\ell_2) & = & [o_3, o_4] \\
\nu(o_3) & = & [\text{valve}, \ell_3] \\
\nu(o_4) & = & [\text{valve}, \ell_3] \\
\nu(\ell_3) & = & [\,]
\end{array}$$

Then the tree associated with $\nu(o_1)$ according to the procedure described above can be written (in infix notation) as

$$\Big[\text{car}, \Big[\big[\text{engine}, \big[[\text{valve}, [\,]], [\text{valve}, [\,]]\big]\big]\Big]\Big].$$

Since there are no cyclic references, the tree is finite. ■

Example 2 Now consider a database containing the six objects *adam*, *eve*, *adam*$'$, *eve*$'$, *adam*$''$, and *eve*$''$, with the following values:

$$\begin{array}{rcl}
\nu(\mathit{adam}) & = & [\text{adam}, \mathit{eve}] \\
\nu(\mathit{eve}) & = & [\text{eve}, \mathit{adam}'] \\
\nu(\mathit{adam}') & = & [\text{adam}, \mathit{eve}'] \\
\nu(\mathit{eve}') & = & [\text{eve}, \mathit{adam}] \\
\nu(\mathit{adam}'') & = & [\text{adam}, \mathit{eve}''] \\
\nu(\mathit{eve}'') & = & [\text{eve}, \mathit{adam}''].
\end{array}$$

The tree associated with *adam* is infinite: from the root emanates an infinite path of right children. The internal nodes all have one left child alternatingly labeled 'adam' and 'eve' starting with 'adam' at the root. This same tree is associated to the objects *adam*$'$ and *adam*$''$ as well. The tree associated with *eve* is similar to that of *adam*; it only differs in that the labeling starts with 'eve' at the root. Again the same tree is associated to *eve*$'$ and *eve*$''$ as well. ■

How the infinite tree associated to an object can be defined formally was shown in [AK89]: one considers the set of all tree equations of the form $o = \nu(o)$, with o an oid in the database. One considers in this system of equations the oids as indeterminates, standing for (possibly infinite) ground trees. A *solution* to the system of equations is a substitution assigning to each oid o a ground tree $tree(o)$ such that all equations become equalities under this substitution. There always exists a unique such solution [Cou83].[1] Each tree $tree(o)$ is *regular*: although it may be infinite, it has only a finite number of distinct subtrees.

For an object o, $tree(o)$ is the entire value structure that becomes visible from o by following oid references in the forward direction only. Hence, it seems natural to adopt the following definition:

Definition 2 Two objects o and p in a tuple database are called *deep-equal*, denoted by $o \overset{\mathrm{d}}{=} p$, if $tree(o) = tree(p)$.

This definition immediately raises two problems, however:

1. How can deep equality be effectively tested for?
2. Up to now we have only considered tuple databases. How do we define deep equality when finite set values can occur?

We comment on these two problems separately:

1. Algorithms are known to test for equality of regular trees defined by equations, by reduction to equivalence of automata [Cou83]. However, we would like a direct procedure, expressed directly in terms of the database objects and values. Such a procedure would have the advantage of being more readily implementable in a sufficiently strong database query language.
2. The difference between sets and tuples is that the latter are ordered while the former are not. The general theory of infinite trees [Cou83] deals explicitly with ordered trees only. Nevertheless, as pointed out

[1]Note that incompletely specified systems of equations, like $\{o = o', o' = o\}$, cannot occur since we assumed from the outset that the value of an oid cannot be simply another oid.

in [AK89], one can in principle still assign regular trees to objects in databases with set values [AK89] (given that the sets are finite). This leads to trees in which certain nodes represent sets rather than tuples. However, the children of these nodes must be thought of as unordered, and duplicate subtrees can occur which should be identified (note that these subtrees can in turn contain set nodes). The proper notion of *equality* in this setting is no longer immediately clear.

In the next section, we will address and solve the two problems together.

4 Deep equality

In the previous section, we have defined deep equality in the special case of tuple databases. We next present a characterization of deep equality in this case which will suggest a definition in the general case, as well as a direct polynomial-time algorithm for testing deep equality.

Thereto, we first need to make the following convention. Consider a fixed equivalence relation on a set O of oids. We can extend $\equiv$ in a natural way to values over O in the following inductive manner:

1. The only value equivalent to a basic value is the basic value itself;
2. Two tuple values of the same width are equivalent if they are equivalent component-wise;
3. Two set values are equivalent if each element in the first set is equivalent to an element in the second set, and vice versa.
4. No other values are equivalent.

Another way of looking at this is as follows: for each equivalence class of oids, choose a unique representative. Given two values v and w, replace each oid occurring in them by the representative of its equivalence class, yielding $\bar{v}$ and $\bar{w}$. Then v and w are equivalent if and only if $\bar{v} = \bar{w}$. So this is indeed a very natural and canonical extension. If v and w are flat values (as we have assumed from the outset), the test $\bar{v} = \bar{w}$ can be implemented in time $O(n)$ for tuples (if the representative of each oid is already available), and time $O(n \log n)$ for sets (which have to be sorted and duplicate-eliminated first).

In what follows, we will implicitly extend equivalence relations on oids to equivalence relations on values in this fashion.

We can now present the following definition and proposition:

Definition 3 An equivalence relation on the oids in a database is called *value-based* if under this relation, two oids o and p are equivalent if and only if their values $\nu(o)$ and $\nu(p)$ are.

So, under a value-based equivalence relation, equivalence of objects depends solely on the values of these objects. Note that these values can contain oids in turn, so the definition is recursive.

We now establish:

Proposition 1 *On tuple databases, deep equality is the coarsest value-based equivalence relation on oids.*

Proof. First, we show that deep equality is indeed value-based. Consider two oids o and p with $o \overset{\mathrm{d}}{=} p$, i.e., $\mathit{tree}(o) = \mathit{tree}(p)$. Then

$$\mathit{tree}(\nu(o)) = \mathit{tree}(o) = \mathit{tree}(p) = \mathit{tree}(\nu(p)).$$

We distinguish two possiblities:

1. $\nu(o)$ and $\nu(p)$ are basic values, in which case they must be identical and hence equivalent;

2. $\nu(o)$ and $\nu(p)$ are tuples. Since $\mathit{tree}(\nu(o)) = \mathit{tree}(\nu(p))$, the corresponding tuple components are either equal (if they are basic values), or have equal trees (if they are oids). In both cases, the tuple components are deep equal, whence $\nu(o)$ and $\nu(p)$ are deep equal.

Conversely, if the values of two oids are deep equal then the two oids are deep equal as well since the tree of an oid equals the tree of its value.

We next show that deep equality is the coarsest. Thereto, let $\equiv$ be any value-based equivalence relation on the oids of the database. Consider two oids o and p with $o \equiv p$. We have to show that $o \overset{\mathrm{d}}{=} p$.

First, we need the notion of *partial branch* in an ordered tree. The set of all partial branches in an ordered tree is the set of all sequences of natural numbers defined as follows:

1. The empty sequence is a partial branch, representing the root of the tree.
2. If b is a partial branch denoting a node n in the tree, and i is a natural number such that n has an i-th child, then (b, i) is a partial branch denoting this child.

The node represented by a partial branch b of a tree t is denoted by $t[b]$.

By induction, we prove the following lemma: *For every partial branch b in $tree(o)$, b is also a partial branch in $tree(p)$ and the nodes $tree(o)[b]$ and $tree(p)[b]$ represent basic values or oids that are equivalent under $\equiv$.*

If b is empty, we have $tree(o)[b] = o$ and $tree(p)[b] = p$ and indeed we have $o \equiv p$.

Now let (b, i) be a partial branch in $tree(o)$. So $tree(o)[b]$ has an i-th child, and hence $tree(o)[b]$ represents an oid, denoted by o'. By induction, we know that $tree(p)[b]$ represents an oid p' equivalent to o' under $\equiv$. Since $\equiv$ is value-based, we know that $\nu(o) \equiv \nu(p)$. Since $tree(o)[b]$ is the root of $tree(o')$ occurring as a subtree in $tree(o)$, we know that $tree(o)[b, i]$ ($tree(p)[b, i]$) represents the i-th component of $\nu(o)$ ($\nu(p)$). Since $\nu(o) \equiv \nu(p)$, the fact to be proven follows.

A consequence of the lemma is that every partial branch in $tree(o)$ is also a partial branch in $tree(p)$ with the same labeling of the nodes along the branch. By symmetry, we have also the converse and we can conclude that $tree(o)$ and $tree(p)$ have the same set of "labeled partial branches". It is well-known [Cou83] that two (possibly infinite) ordered trees are equal if and only if their sets of labeled partial branches are equal. Hence, $tree(o) = tree(p)$ and thus $o \overset{\mathrm{d}}{=} p$, as had to be shown. ■

Example 3 To illustrate the above proposition, we point out that in general there may exist several different value-based equivalence relations on oids (hence the qualification "the coarsest" really makes a difference). The simplest example is provided by two mutually dependent objects o_1 and o_2 as follows:

$$\begin{aligned} \nu(o_1) &= [o_2] \\ \nu(o_2) &= [o_1] \end{aligned}$$

Both the equality relation (under which o_1 and o_2 are not equivalent) and the full relation (under which they are equivalent) are value-based. The full relation is of course the coarsest of the two, and indeed, o_1 and o_2 are deep-equal. ■

Proposition 1 yields insight in the concept of deep equality: deep equality is the equivalence relation which makes the fewest possible distinctions among oids, while at the same time distinguishing among all different basic values, such that objects and their values are identified. Moreover, the reader familiar with the theory of communication and concurrency will have noticed the analogy with the observational equivalence concept of *strong bisimilarity* [Mil89].

We therefore propose to adopt Proposition 1 as the *definition* of deep equality in the general (i.e., not necessarily tuple database) case. Indeed, the notion of value-based equivalence relation is also well-defined in the presence of set values. Thus:

Definition 4 *Deep equality*, denoted $\overset{\mathrm{d}}{=}$, is the coarsest value-based equivalence relation on the oids in the database.

To see that this definition is well-defined, i.e., that there is a unique coarsest value-based equivalence relation, consider the following operator on equivalence relations:

Definition 5 Let $\equiv$ be an equivalence relation on the oids of some fixed database. The *value refinement of* $\equiv$, denoted by *Refine*($\equiv$), is the equivalence relation on the same set of oids under which two oids are equivalent if and only if their values are equivalent under $\equiv$.

This operator is monotone with respect to set inclusion. It thus follows from Tarski's fixpoint theorem that it has a unique greatest fixpoint. Moreover, an equivalence relation is a fixpoint of the operator *Refine* precisely when it is value-based. Putting everything together, we can thus conclude:

Lemma 1 *Deep equality is the greatest fixpoint of the operator Refine.*

As is well-known, this greatest fixpoint can be computed as follows:

1. Start with the coarsest possible equivalence relation on the oids of the database, under which any two oids are equivalent;

2. Apply *Refine* repeatedly until a fixpoint is reached.

Since at every iteration that has not yet reached the fixpoint, at least one equivalence class will be split, the number of iterations until the fixpoint is reached is at most linear.

A polynomial-time algorithm for computing deep equality is now readily derived, using techniques similar to those used in stable coloring algorithms for testing isomorphism for certain classes of graphs [RC77]. One starts by coloring each oid with the same color. During the iteration, one replaces the color of an oid by the coloring of its value. Between rounds, the colors are replaced by their order numbers in the lexicographic order of all the occurring colors. This always keeps the colors short. The algorithm stops when the coloring stabilizes, i.e., when no new differences between oids are discovered.

Example 4 An example of how the algorithm proceeds on a database consisting of objects having values of the form $[v, o]$, where v is a basic value and o is an oid, is shown in Figure 1. Horizontal arrows represent the links from oids to oids; vertical arrows represent the links from oids to basic values. The second attribute of o_3 is assumed to be o_3 itself (not shown in the figure). The colors are given as numbers. There are three iterations in this example. The objects with the same color in the final stable coloring are those that are deep-equal (in this example, these are the pairs $o_1 \overset{\mathrm{d}}{=} o_1'$ and $o_2 \overset{\mathrm{d}}{=} o_2'$, plus all identical pairs). ■

5 Indistinguishability

As discussed in the introduction, a basic intuition underlying deep equality is that deep-equal objects can not be distinguished by observing basic values, dereferencing oids, and following paths in complex values. To make this intuition precise, we need to define a query language in which two objects are indistinguishable if and only if they are deep-equal. In analogy with the notion of value-based equivalence relation of the previous section, we call such a query language *value-based.* In this section, we will define a value-based calculus language called the *observation calculus.*

A first observation is that in a value-based language, equality comparisons on oids cannot be permitted. Indeed, recall Example 2. Objects *adam* and database:

$$\begin{tikzcd}
o_0 \arrow[r] \arrow[d] & o_1 \arrow[r] \arrow[d] & o_2 \arrow[r] \arrow[d] & o_3 \arrow[d] & o_2' \arrow[l] \arrow[d] & o_1' \arrow[l] \arrow[d] \\
a \arrow[r] & a \arrow[r] & a \arrow[r] & b & a \arrow[l] & a \arrow[l]
\end{tikzcd}$$

initial coloring:	1	1	1	1	1	1
first iteration:	1	1	1	2	1	1
second iteration:	1	1	2	3	2	1
stable coloring:	1	2	3	4	3	2

Figure 1: Testing deep-equality.

$adam''$ are deep-equal, but they can be distinguished using the following formula $\varphi(x)$ using a comparison:

$$\exists y, \exists z : y = \nu(x).2 \wedge z = \nu(y).2 \wedge z \neq x$$

Indeed, $\varphi(adam)$ is true while $\varphi(adam'')$ is false.

A second observation is that quantifiers must be "range-restricted" (as is actually the case in the formula φ above). Indeed, recall Figure 1. Objects o_1 and o_1' are deep-equal, but they can be distinguished using the following formula $\psi(x)$ using an unrestricted quantifier:

$$\exists y : x = \nu(y).2$$

Indeed, $\psi(o_1)$ is true while $\psi(o_1')$ is false. Note that this example also illustrate that unrestricted quantifiers effectively allow "backwards following of pointers" and hence can break deep-equality.

We now turn to the definition of the observation calculus.

Definition 6 The *observation calculus* uses variables ranging over basic values and oids. The formulas of the observation calculus are inductively defined as follows:

1. **true** is a formula;

2. If x and y are variables and v is a basic value, then $x =_b y$ and $x =_b v$ are formulas;

3. If φ and ψ are formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$;

4. If φ is a formula in which variable x does not occur and in which variable y occurs only free, then the following are formulas:

   - $(\exists y : y =_b \nu(x))\varphi$;
   - $(\exists y : y = \nu(x).i)\varphi$, with i a natural number;
   - $(\exists y : y \in \nu(x))\varphi$.

The semantics of observation formulas is the obvious one, with the following precautions:

- The equality predicate $=_b$ is only defined on basic values: from the moment that one of x and y is an oid, $x =_b y$ becomes false.

- The quantifier $(\exists y : y =_b \nu(x))$ can only be true when x is an oid such that $\nu(x)$ is a basic value; in this case y is bound to this basic value.

- The quantifier $(\exists y : y = \nu(x).i)$ can only be true when x is an oid such that $\nu(x)$ is a tuple of at least i components; in this case y is bound to this component.

- Finally, the quantifier $(\exists y : y \in \nu(x))$ can only be true when x is an oid such that $\nu(x)$ is a set; in this case y ranges over the elements of this set.

As usual, disjunction and universal quantifiers can be simulated using negation. We would like to repeat that observation formulas are meant as a simple-to-define formalization of typical object database browsing interfaces, as discussed in the introduction, and not as a user-friendly language.

Example 5 Consider a part-subpart database. Each part object has as value a tuple $[v, s]$, where v is the part type (a basic value) and s is a set object. Each set object has as value a set of part oids (the subparts). The following observation formula $\varphi(x_1, x_2)$, checks whether part object x_2 has at least all types of subparts as object x_1:

$$\begin{aligned}&(\exists s_1 : s_1 = \nu(x_1).2)(\exists s_2 : s_2 = \nu(x_2).2)\\ &\quad(\forall y_1 : y_1 \in \nu(s_1))(\exists y_2 : y_2 \in \nu(s_2))(\exists k_1 : k_1 = y_1.1)(\exists k_2 : k_2 = y_2.1)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k_1 =_b k_2.\end{aligned}$$

■

Formally, two objects o_1 and o_2 in a fixed database are called *indistinguishable by observation formulas* if for every observation formula $\varphi(x)$, $\varphi(o_1)$ holds in the database if and only if $\varphi(o_2)$ holds in the database. We now establish the announced result:

Proposition 2 *Two objects are deep-equal if and only if they are indistinguishable by observation formulas.*

Proof. *If.* Let O be the set of oids in the database, and let n be the cardinality of O. Recall from the previous section that deep equality equals *Refine*$^n(O \times O)$. For any natural number i, denote *Refine*$^i(O \times O)$ by $\equiv_i$. Furthermore, let $\mathcal{C}_i$ denote the partition of O according to $\equiv_i$.

By induction, we prove the following lemma: *For each i, and each equivalence class C in $\mathcal{C}_i$, there is an observation formula ψ_i^C defining C.*

The base case $i = 0$ is trivial; $\mathcal{C}_0$ consists of O only, and ψ_i^O is simply **true**.

Now let $i > 0$. Recall that, by the definition of *Refine*, two objects are equivalent under $\equiv_i$ if and only if their values are equivalent under $\equiv_{i-1}$. Let $C \in \mathcal{C}_i$. So C consists of all objects equivalent to a certain object o. We distinguish three possibilities:

1. The value of o is a basic value v. Then $\psi_i^C(x)$ is

$$(\exists y : y =_b \nu(x))y =_b v.$$

2. The value of o is a tuple $[v_1, \ldots, v_k]$. For each ℓ between 1 and k, let y_ℓ be a variable, and let the formula $\varphi_\ell(y_\ell)$ be either

   - $y_\ell =_b v_\ell$, if v_ℓ is a basic value; or

- $\psi^B_{i-1}(y_\ell)$, if v_ℓ is an oid, where B is the equivalence class of v_ℓ under $\equiv_{i-1}$.

The desired formula $\psi^C_i(x)$ now is

$$(\exists y_1 : y_1 = \nu(x).1) \dots (\exists y_k : y_k = \nu(x).k) \varphi_1(y_1) \wedge \dots \wedge \varphi_k(y_k).$$

3. The value of o is a set $\{v_1, \dots, v_m\}$. For each ℓ between 1 and m, let $\varphi_\ell(y)$ be defined as in the previous item. The desired formula $\psi^C_i(x)$ then is

$$\begin{aligned}&(\forall y : y \in \nu(x))(\varphi_1(y) \vee \dots \vee \varphi_m(y)) \wedge \\ &\qquad \big((\exists y : y \in \nu(x))\varphi_1(y)\big) \wedge \dots \wedge \big((\exists y : y \in \nu(x))\varphi_m(y)\big).\end{aligned}$$

Now let o and p be oids such that $o \overset{\mathrm{d}}{\neq} p$, i.e., $o \not\equiv_n p$. We have to show that o and p can be distinguished by an observation formula. By the lemma, the equivalence class of o under $\equiv_n$ can be defined by an observation formula ψ. Obviously, $\psi(o)$ holds while $\psi(p)$ does not, and thus ψ distinguishes between o and p.

Only if. Let o and p be oids such that $o \overset{\mathrm{d}}{=} p$. We prove by induction that for each observation formula $\varphi(x)$, $\varphi(o)$ holds iff $\varphi(p)$ holds. The base case is trivial; the atomic formula **true** is always true, and the atomic formulas $x =_b y$ and $x =_b v$ are always false on oids. The cases of negation and conjunction are straightforward. For the case of existential quantification, we distinguish three possibilities:

1. $\varphi(x)$ is $(\exists y : y =_b \nu(x))\psi$. We have:

$$\begin{aligned}\varphi(o) \text{ holds} \quad &\Leftrightarrow \quad \nu(o) \text{ is a basic value } v \text{ and } \psi(v) \text{ holds} \\ &\Leftrightarrow \quad \nu(p) \text{ is a basic value } v \text{ and } \psi(v) \text{ holds} \\ &\Leftrightarrow \quad \varphi(p) \text{ holds.}\end{aligned}$$

The second equivalence follows from the deep equality of o and p.

2. $\varphi(x)$ is $(\exists y : y = \nu(x).i)\psi$. We have:

$$\begin{aligned}\varphi(o) \text{ holds} \;&\Leftrightarrow\; \nu(o) \text{ is a tuple with } i\text{-th component } v \text{ and } \psi(v) \text{ holds}\\ &\Leftrightarrow\; \nu(p) \text{ is a tuple with } i\text{-th component } v \text{ and } \psi(v) \text{ holds}\\ &\Leftrightarrow\; \varphi(p) \text{ holds.}\end{aligned}$$

The second equivalence follows from the deep equality of o and p (and thus the deep equality of $\nu(o)$ and $\nu(p)$) and the induction hypothesis (in case v is an oid).

3. $\varphi(x)$ is $(\exists y : y \in \nu(x))\psi$. This case is analogous to the previous one. ■

To conclude, we note that a number of variations on the above theme are possible.

If the number of quantifiers in observation formulas is bounded, then indistinguishability amounts to deep equality up to a bounded depth in the infinite trees only, or equivalently, to a bound on the number of iterations in the fixpoint algorithm for deep equality.

One might also ask what happens in the case of the natural calculus, more powerful than the observation calculus, obtained by allowing unrestricted quantifiers $(\exists x)$ ranging over all oids and basic values in the database, and allowing $y =_b \nu(x)$, $y = \nu(x).i$, and $y \in \nu(x)$ as atomic formulas. As noted in the beginning of this section, this amounts to allowing pointers to be followed backwards as well. One can then show that o_1 and o_2 are indistinguishable if and only if there exists a surjective strong homomorphism of the database to itself, mapping o_1 to o_2, which is the identity on basic values, and conversely, another such homomorphism must exist mapping o_2 to o_1. This can be easily proven by reduction to a well-known fact in model theory which says that two relational structures are indistinguishable in first-order logic without equality if and only if there exist strong surjective homomorphisms between them. This reduction works by representing an object database instance as a relational structure in the natural way.

6 Expressibility

Is deep equality expressible in deductive database languages? The answer may depend on the kind of databases under consideration. In the special case of tuple databases, for instance, deep equality is readily expressed by the following program in Datalog with stratified negation. The atomic EDB predicates are the same as those of the calculus discussed at the end of the previous section.

$$
\begin{array}{l}
not\_deq(z,w) \leftarrow z =_b \nu(x), w =_b \nu(y), z \neq_b w \\
not\_deq(x,y) \leftarrow z =_b \nu(x), w =_b \nu(y), z \neq_b w \\
not\_deq(x,y) \leftarrow z = \nu(x).1, w = \nu(y).1, not\_deq(z,w) \\
\vdots \\
not\_deq(x,y) \leftarrow z = \nu(x).k, w = \nu(y).k, not\_deq(z,w) \\
deq(x,y) \leftarrow \neg not\_deq(x,y)
\end{array}
$$

Here, k is the maximum width of any typle appearing in the database.[2] Note that only two strata are needed. In particular, the complement of deep equality is expressible in Datalog without negation (only non-equality).

In the general case, i.e., when set values can occur in the database, the use of negation becomes fundamental. For example, on databases where the value of each object is either a basic value or a set of oids, we can express deep equality as $\neg not\_deq(x,y)$, where $not\_deq(x,y)$ is defined as the least fixpoint of the following first-order query:

$$
\begin{array}{ll}
 & (\exists x')(\exists y')(x =_b \nu(x') \wedge y =_b \nu(y') \wedge x \neq_b y) \\
\vee & (\exists z)(\exists w)(z =_b \nu(x) \wedge w =_b \nu(y) \wedge z \neq_b w) \\
\vee & (\exists z \in \nu(x))(\forall w \in \nu(y)) not\_deq(z,w) \\
\vee & (\exists w \in \nu(y))(\forall z \in \nu(x)) not\_deq(z,w).
\end{array}
$$

Because of the recursion through univeral quantification, this fixpoint does not correspond in any straightforward way to a program in Datalog with stratified negation. In fact, we can show that no such program exists:

Proposition 3 *Deep equality is not expressible in Datalog with stratified negation.*

Proof. The proof is based on a paper by Kolaitis [Kol91], where an analysis of the expressive power of stratified Datalog is presented in terms of two families of tree structures $B_{i,k}$ and $B'_{i,k}$, for $i \geq 0$ and $k \geq 1$. These structures

[2]When the database is an instance of a known database schema, k is known in advance.

had been discovered earlier by Chandra and Harel [CH82], and are defined as follows. For any fixed k, the definition is by induction on i. Each structure consists of a binary relation *Move*, giving the directed edges in the tree, and a unary relation *Black*, coloring certain leafs in the tree.

- $B_{0,k}$ and $B'_{0,k}$ consist of a single node colored *Black* in $B_{0,k}$ but not in $B'_{0,k}$. The *Move* relation is empty in both.
- $B_{i+1,k}$ consists of a copy of $B'_{i,k}$, k disjoint copies of $B_{i,k}$, and a new root node with *Move*-edges to the roots of all these copies.
- $B'_{i+1,k}$ consists of $k+1$ copies of $B_{i,k}$ and a new root node with *Move*-edges to the roots of all these copies.

Kolaitis proved the following fact, which we denote by $(*)$: *for every stratified program P there is a natural number ℓ such that P is equivalent, on all structures $B_{\ell+2,k}$ and $B'_{\ell+2,k}$ for any k, to a first-order formula χ in Σ_{ℓ,k_0} for some k_0.* The latter means that χ is a prenex normal form formula with ℓ alternations of quantifiers, starting with an existential one, and such that each block of quantifiers of the same type has length at most k_0.

Chandra and Harel had proved the following fact, which we denote by $(\dagger)$: *for any ℓ and k_0, the structures $B_{\ell+2,k_0}$ and $B'_{\ell+2,k_0}$ are indistinguishable by any formula in Σ_{ℓ,k_0}.*

As a result, for any program P there are natural numbers ℓ and k_0 such that $B_{\ell+2,k+0}$ and $B'_{\ell+2,k_0}$ are indistinguishable by P.

Now define the disjoint sum $C_{i,k} = B_{i,k} \oplus B_{i,k}$ consisting of two disjoint copies of $B_{i,k}$, and $C'_{i,k} = B_{i,k} \oplus B'_{i,k}$ consisting of a copy of $B_{i,k}$ and a copy of $B'_{i,k}$. Inspection of Kolaitis's proof yields that the above fact $(*)$ also holds when the disjoint sums C and C' are substituted for the single structures B and B'. Indeed, the key to the proof of $(*)$ is Lemma 5 in [Kol91], which is proven by verifying that the number of n-types on $B_{i,k}$ and $B'_{i,k}$ can be bounded by functions $f_n(i)$ and $f'_n(i)$ that depend only on i. Since the number of n-types on a disjoint sum of structures is at most the sum of the numbers on the component structures, the Lemma carries over.

Moreover, also the fact $(\dagger)$ carries over. Indeed, Chandra and Harel's proof is an Ehrenfeucht-Fraïssé game argument, and a winning strategy on two structures $\mathcal{A}$ and $\mathcal{B}$ readily yields a winning strategy on the two structures $\mathcal{A} \oplus \mathcal{A}$ and $\mathcal{A} \oplus \mathcal{B}$ as well.

We can conclude that for any program P there are natural numbers ℓ and k_0 such that $B_{\ell+2,k_0} \oplus B_{\ell+2,k_0}$ and $B_{\ell+2,k_0} \oplus B'_{\ell+2,k_0}$ are indistinguishable by P.

We are now ready to establish the link of the above with deep equality. Any tree structure as above can be viewed as a database as follows. Each node is an object. An internal node has the set of its children in the tree as value. A leaf node colored *Black* has a basic value as value, say 1, and a leaf node not colored *Black* has a different basic value as value, say 0. Under this view, the following is readily verified by induction on i: *for any k and i, the roots of the two trees in the structure $B_{i,k} \oplus B'_{i,k}$ are not deep equal.* On the other hand, the roots of the two trees in the structure $B_{i,k} \oplus B_{i,k}$ are trivially deep equal.

Now assume that, for the sake of contradiction, a program P exists which expresses deep equality on any database $C_{i,k} = B_{i,k} \oplus B_{i,k}$ or $C'_{i,k} = B_{i,k} \oplus B'_{i,k}$. Replace each atomic formula of the form $y \in \nu(x)$ by $Move(x,y)$, replace $\nu(x) =_b 1$ by $Black(x)$, and replace $\nu(x) =_b 0$ by $\neg Black(x)$. By the previous observation on deep equality, the program will dinstinguish between $C_{i,k}$ and $C'_{i,k}$ for all i and k; however, we know that there exist ℓ and k_0 such that P cannot distinguish between $C_{\ell+2,k_0}$ and $C'_{\ell+2,k_0}$. This yields the desired contradiction. ■

References

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

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an object-oriented database system: The story of O_2*. Morgan Kaufmann, 1992.

[BDS95] P. Buneman, S.B. Davidson, and D. Suciu. Programming constructs for unstructured data. Department of Computer and Information Science, University of Pennsylvania, 1995. To appear in the proceedings of the *Fifth International Workshop on Database Programming Languages*, held in Gubbio, Italy, September 1995.

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

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DV93] 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.

[KC86] S.N. Khoshafian and G.P. Copeland. Object identity. In N. Meyrowitz, editor, *Object-oriented programming systems, languages and applications: Proceedings OOPSLA'86*, SIGPLAN Notices 21:11, pages 406–416. ACM Press, 1986.

[KLR92] P. Kanellakis, C. Lécluse, and P. Richard. The O_2 data model. In Bancilhon et al. [BDK92], chapter 3.

[Kol91] P.G. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90:50–66, 1991.

[Kos95] A. Kosky. Observational distinguishability of databases with object identity. Technical Report MS-CIS-95-20, University of Pennsylvania, 1995. To appear in the proceedings of the *Fifth International Workshop on Database Programming Languages*, held in Gubbio, Italy, September 1995.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[P$^+$92] D. Plateau et al. Building user interfaces with LOOKS. In Bancilhon et al. [BDK92], chapter 22.

[RC77] R.C. Read and D.G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[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.