ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

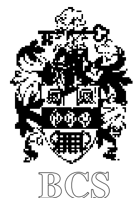
Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Observational Distinguishability of Databases with Object Identity

Anthony S. Kosky

Published in collaboration with the British Computer Society



©*Copyright in this paper belongs to the author(s)*

OBSERVATIONAL DISTINGUISHABILITY OF DATABASES WITH OBJECT IDENTITY*

Anthony Kosky
Information and Computing Sciences Division
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720
Email: Anthony_Kosky@lbl.gov

Abstract

We will examine the problem of distinguishing between database instances and values in models which incorporate object-identities and recursive data-structures. We will show that the notion of observational distinguishability is intricately linked to the languages available for querying a database. In particular we will show that, given a simple query language incorporating a test for equality of object-identities, database instances are indistinguishable iff they are *isomorphic*, and that, in a language without any operators on object-identities, database instances are indistinguishable iff a *bisimilarity* relation holds between them. Further, such a bisimulation relation may be computed on values, but doing so requires the ability to recurse over all the object-identities in an instance.

We will then show that systems of *keys* give rise to observational distinguishability relations which lie between these two extremes. We show that a system of keys satisfying certain restrictions provides us with an efficient means of comparing values, while avoiding the need to compare object identities directly.

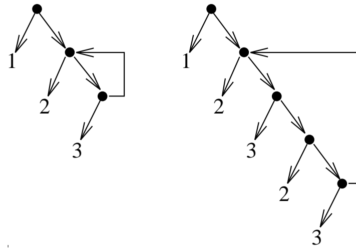
1 Introduction

Suppose you were presented with two database instances and wished to find out whether or not the instances were different using some query interface. Using certain data-models and query languages this might be easy. For example, in a relational database system, simply printing out the two instances and comparing them would suffice. More succinctly, you could find a single query which would produce different results when applied to any two instances if the instances were different. Even if the instances and interface involved more complex but fixed depth types, such as in a nested relational model, as long as the query interface allowed you to "see" instances completely you could distinguish any two distinct instances.

However, in a model allowing recursive or arbitrarily deeply nested data structures, such as a semantic or object-oriented data model [4, 10], this technique will not work. In this case database instances must use some kind of reference mechanism, such as object identities, pointers, logical variables, or some other non-printable values, and so physically differing instances may give identical results on all possible queries. Of the various possible reference mechanisms, we will focus our attention on object identities since they offer the advantage of locational and data independence, and also afford efficient implementation techniques [11].

Suppose, for example, we had the two instances shown bellow: object identities are represented by $\bullet$, and each identity has a value associated with it consisting of an integer and another object identity.

*This research was supported in part by the following grants: DE-FG02-94-ER-61923 Sub 1, BIR94-02292 PRIME, DAAH04-93-G0129, DE-AC03-76SF00098.



If our query language allowed us only to print out the values on paths of any fixed depth, then we could not observe any differences between these two instances: they would both represent the infinite sequence of integers $1, 2, 3, 2, 3, 2, 3, \ldots$. However their representations are clearly different.

Though this hypothetical situation may appear unrealistic, it in fact represents a fundamental problem: in any query or database programming language it is necessary to have some means of comparing data values in an instance. Further, in order to reason about the expressive power of a data-model and query language, it is necessary to be able to compare distinct database instances and to communicate information between them. These issues are complicated by the presence of object identities in a data model: there may be many different ways of representing the same data using different choices, and possibly different structures and interconnections of identities. Consequently we would like to regard object-identities as not directly observable, and equate any values which are *observationally indistinguishable*. We shall see that the notion of observational distinguishability is intricately linked to the languages and operations that are available for querying a database. An understanding of these issues is essential in the design of languages for such data-models.

In this paper we will make use of a data-model equivalent to that of [1] in order to examine these issues. We will define an *isomorphism* relation on database instances, representing when two instances differ only in their choice of object-identifiers, and a bisimulation relation, representing when two instances have the same set of paths. We will prove that, given a simple query language incorporating a test for equality on object-identities, two instances are *indistinguishable* if and only if they are isomorphic, and that, in a query language without any comparison operators available on object identities, two instances are indistinguishable iff they are bisimilar. However, in both of these cases, it is not possible to find a *generic* query to distinguish between instances: that is, it is not possible to find a finite set of queries, dependent only on a database schema, which will evaluate to the same values on two instances if and only if the instances can not be distinguished with any query.

We show that it is possible to compute the bisimulation relation on values of a database, but in order to do so it is necessary for our query language to allow recursion over the finite extents of object-identities in the database. We conclude that isomorphism and bisimulation represent respectively the finest and coarsest possible observational equivalences on instances.

An important class of observational equivalences, in between these two, can be obtained using systems of *keys* to determine object identities. We show that, given certain acyclicity restrictions on a system of keys, the resulting equivalences on values can be computed efficiently without resorting to recursion over the entire set object identities. Consequently, by making such systems of keys primitive in a query language, we can obtain a value-oriented language while achieving much of the efficiency of an object-identity oriented language. Further suitable systems of keys can be used to control the creation of object-identifiers in a manner similar to that of [9], so that we can we can have a query language which supports the creation of object identifiers, but avoids the potential for non-terminating computations present in languages that allow unconstrained creation of object identities, such as IQL ([1]).

2 A Data model with object identities and finite extents

The description of our data-model falls naturally into two parts: the definition of schemas and that of instances. The schemas are defined in terms of *types*, and consist of a type system which is dependent on a finite set of *classes*, and an association between these classes and types.

The model presented here is equivalent to that of [1], and could also be considered to be simplification of the models of [3, 10].

2.1 Types and schemas

The types in our model are similar to the nested relational types of [2] with the additional feature of *class types*. These represent the *extents* present in a database, and therefore go beyond the structural information normally associated with a type system.

In order to describe a particular database system it is necessary to state what classes are present, and also the types of (the values associated with) the objects of each class. We consider that these two pieces of information constitute a database *schema*.

Note that, in many data-models, schemas may represent a wide variety of additional constraints; however we believe that this information represents the minimal information which must be present in the schemas of any data-model.

Assume a finite set of *classes* $\mathcal{C}$, ranged over by $C, C', \ldots$, and a countable set of *attribute labels*, $\mathcal{A}$, ranged over by $a, a', \ldots$. The **types** over $\mathcal{C}$, ranged over by $\tau, \ldots$, consist of *base types* $\underline{b}$, *class types* C, where $C \in \mathcal{C}$, *record types* $(a_1 : \tau_1, \ldots, a_k : \tau_k)$, *variant types* $\langle\!| a_1, \tau_1, \ldots, a_k : \tau_k |\!\rangle$, and *set types* $\{\tau\}$. We write $Types^{\mathcal{C}}$ for the set of types over $\mathcal{C}$.

A **schema** consists of a finite set of classes, $\mathcal{C}$, and a mapping $\mathcal{S} : \mathcal{C} \to Types^{\mathcal{C}}$, such that $\mathcal{S}(C) = \tau^C$ where τ^C is not a class type. (Since $\mathcal{C}$ can be determined from $\mathcal{S}$ we will also write $\mathcal{S}$ for the schema).

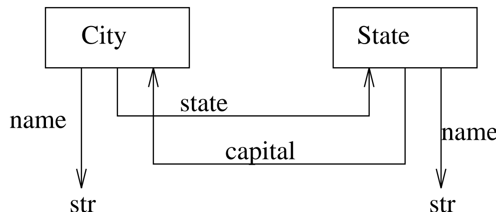


Figure 1: A simple database schema

Example 2.1: As an example let us consider a database with two classes, *Cities* and *States*, illustrated in figure 1. Each City has two components: a *name* and a *state* to which the City belongs, while each State also has two components: a *name* and a city which is its *capital*. Our set of classes is $\mathcal{C} \equiv \{City, State\}$ and the schema mapping, $\mathcal{S}$, is given by

$$\begin{aligned} \mathcal{S}(City) &\equiv (name : str, state : State) \\ \mathcal{S}(State) &\equiv (name : str, capital : City) \end{aligned}$$

That is, a City is a pair consisting of a string (its name) and a State (its State), while a State is a pair consisting of a string (its name) and a City (its capital). ∎

2.2 Values and instances

The values that may occur in a particular database instance depend on the object identities of that instance. Consequently we must first define the domain of database values and the denotations of types for a particular choice of sets of object identities, and then define instances using these constructs.

Suppose, for each class $C \in \mathcal{C}$ we have a disjoint finite set σ^C of *object-identities* of class C.

For each base type $\underline{b}$, assume a domain $\mathbf{D}_{\underline{b}}$ associated with $\underline{b}$. We define the *domain* of our model for the sets objects identities $\sigma^{\mathcal{C}}$, $\mathbf{D}(\sigma^{\mathcal{C}})$, to be the union of the following sets: $\mathbf{D}_{\underline{b}}$ for each base type $\underline{b}$; σ^C for each class $C \in \mathcal{C}$; partial functions with finite domains from $\mathcal{A}$ to $\mathbf{D}(\sigma^{\mathcal{C}})$ for record types; pairs from $\mathcal{A} \times \mathbf{D}(\sigma^{\mathcal{C}})$ for variants; and finite subsets of $\mathbf{D}(\sigma^{\mathcal{C}})$ for set types.

$$
\begin{aligned}
[\![\underline{b}]\!]\sigma^{\mathcal{C}} &\equiv \mathbf{D}_{\underline{b}} \\
[\![C]\!]\sigma^{\mathcal{C}} &\equiv \sigma^C \\
[\![(a_1 : \tau_1 \ldots, a_k : \tau_k)]\!]\sigma^{\mathcal{C}} &\equiv \{f \in \mathcal{A} \xrightarrow{\sim} \mathbf{D}(\sigma^{\mathcal{C}}) \mid dom(f) = \{a_1, \ldots, a_k\} \\
&\qquad \text{and } f(a_i) \in [\![\tau_i]\!]\sigma^{\mathcal{C}},\ i = 1, \ldots, k\} \\
[\![\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle]\!]\sigma^{\mathcal{C}} &\equiv (\{a_1\} \times [\![\tau_1]\!]\sigma^{\mathcal{C}}) \cup \ldots \cup (\{a_k\} \times [\![\tau_k]\!]\sigma^{\mathcal{C}}) \\
[\![\{\tau\}]\!]\sigma^{\mathcal{C}} &\equiv \mathcal{P}_{fin}([\![\tau]\!]\sigma^{\mathcal{C}})
\end{aligned}
$$

Figure 2: The semantic operator on types

We define the semantic operator $[\![\cdot]\!]\sigma^{\mathcal{C}}$ mapping types over $\mathcal{C}$ to subsets of $\mathbf{D}(\sigma^{\mathcal{C}})$ in figure 2.

A database **instance** of schema $\mathcal{S}$ consists of a family of *object sets*, $\sigma^{\mathcal{C}}$, and for each $C \in \mathcal{C}$ a mapping $\mathcal{V}^C : \sigma^C \rightarrow [\![\tau^C]\!]\sigma^{\mathcal{C}}$

Given an instance $\mathcal{I}$ of $\mathcal{S}$ ($\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$), we will also write $[\![\tau]\!]\mathcal{I}$ for $[\![\tau]\!]\sigma^{\mathcal{C}}$.

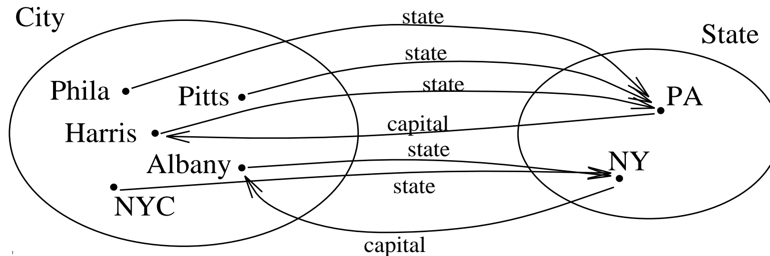


Figure 3: A database instance

Example 2.2: We will describe an instance of the schema introduced in example 2.1. Our object identities are:

$$
\begin{aligned}
\sigma^{City} &\equiv \{Phila, Pitts, Harris, NYC, Albany\} \\
\sigma^{State} &\equiv \{PA, NY\}
\end{aligned}
$$

and the mappings are

$$
\begin{aligned}
\mathcal{V}^{City}(Phila) &\equiv (name \mapsto \text{“Philadelphia”},\ state \mapsto PA) \\
\mathcal{V}^{City}(Pitts) &\equiv (name \mapsto \text{“Pittsburgh”},\ state \mapsto PA) \\
\mathcal{V}^{City}(Harris) &\equiv (name \mapsto \text{“Harrisburg”},\ state \mapsto PA)
\end{aligned}
$$

$$\begin{aligned} \mathcal{V}^{City}(NYC) &\equiv (name \mapsto \text{“New York City”},\ state \mapsto NY) \\ \mathcal{V}^{City}(Albany) &\equiv (name \mapsto \text{“Albany”}, state \mapsto NY) \end{aligned}$$

and

$$\begin{aligned} \mathcal{V}^{State}(PA) &\equiv (name \mapsto \text{“Pennsylvania”},\ capital \mapsto Harris) \\ \mathcal{V}^{State}(NY) &\equiv (name \mapsto \text{“New York”},\ capital \mapsto Albany) \end{aligned}$$

This defines the instance illustrated in figure 3. ■

2.3 Isomorphism of instances

Two instances are said to be *isomorphic* if they differ only in their choice of object identities: that is, one instance can be obtained by *renaming* the object identities of the other instance.

Since object identities are considered to be an abstract notion, and not directly visible, it follows that we would like to regard any two isomorphic instances as the same instance. In particular, any query when applied to two isomorphic instances should return isomorphic results. Isomorphism therefore provides the finest level of distinction between instances that we might hope to observe.

If $\mathcal{I}$ and $\mathcal{I}'$ are two instances of a schema $\mathcal{S}$, and $f^{\mathcal{C}}$ is a family of mappings, $f^C : \sigma^C \rightarrow \sigma'^C$, then we can extend $f^{\mathcal{C}}$ to mappings $f^\tau : [\![\tau]\!]\mathcal{I} \rightarrow [\![\tau]\!]\mathcal{I}'$ as follows:

$$\begin{aligned} f^{\underline{b}} c &\equiv c \\ f^{(a_1:\tau_1,\ldots,a_k:\tau_k)} u &\equiv (a_1 \mapsto f^{\tau_1}(u(a_1)), \ldots, a_k \mapsto f^{\tau_k}(u(a_k))) \\ f^{\langle\!| a_1:\tau_1,\ldots,a_k \tau_k |\!\rangle}(a_i, u) &\equiv (a_i, f^{\tau_i} u) \\ f^{\{\tau\}}\{v_1, \ldots, v_n\} &\equiv \{f^\tau v_1, \ldots, f^\tau v_n\} \end{aligned}$$

A **isomorphism** of two instances, $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and $\mathcal{I}' = (\sigma'^{\mathcal{C}}, \mathcal{V}'^{\mathcal{C}})$, of a schema $\mathcal{S}$ consists of a family of bijections, $f^{\mathcal{C}} : \sigma^C \rightarrow \sigma'^C$, such that for each class $C \in \mathcal{C}$ and each object identity $o \in \sigma^C$

$$\mathcal{V}'^C(f^C o) = f^{\tau^C}(\mathcal{V}^C o)$$

$\mathcal{I}$ and $\mathcal{I}'$, are said to be **isomorphic** iff there exists an isomorphism $f^{\mathcal{C}}$ from $\mathcal{I}$ to $\mathcal{I}'$. We write $\mathcal{I} \cong \mathcal{I}'$ to mean $\mathcal{I}$ is isomorphic to $\mathcal{I}'$.

We will show that, in a query language equipped with an equality test on object identities, isomorphism coincides exactly with observational indistinguishability of instances.

2.4 Bisimulations and correspondences between instances

The data model presented above captures our intuition about how databases with recursive values and extents are represented. We would also like a semantic model where two instances are considered to be different if and only if they are *distinguishable*, or equivalently, a way of grouping together those instances in our model which are indistinguishable. However to talk about whether two instances are distinguishable assumes some latent language for querying the databases, and of course the notion of distinguishability is dependent on this language and the predicates available in it.

It is clear that the isomorphism relation on instances is at least as fine as any possible observational equivalence relation: that is, it should not be possible to distinguish between two isomorphic instances using

any reasonable queries over instances. However there may well be indistinguishable instances that are not isomorphic.

We will construct a "bisimulation" relation on instances based on the idea that no comparisons on object identities are available, and that only base values are directly observable. Other complex values, such as sets and records, can be compared by comparing their component parts. In particular object identities are compared by dereferencing them and comparing their associated values. The equivalence classes of instances under this relation correspond to a *regular tree* or *value based* model of instances (see [1, 12]).

Since we believe that equality tests on base types are common to any query language, and hence that any complex values not containing object identities can be tested for equality by recursively applying type deconstructors and then base equality tests to the values, it follows that bisimulation is the coarsest possible observational equivalence relation on instances: if two instances are not in the bisimulation relation then any reasonable query system should be able to distinguish between them.

We will first define *correspondence* relations between the object identities of two instances, and then define bisimulation to be the largest correspondence relation satisfying certain consistency conditions.

A **correspondence** between two families of object identifiers $\sigma^{\mathcal{C}}$ and $\sigma'^{\mathcal{C}}$ is a family of binary relations $\sim^C \subseteq \sigma^C \times \sigma'^C$.

For each type τ, we can extend $\sim^{\mathcal{C}}$ to a binary relation $\sim^\tau \subseteq [\![\tau]\!]\sigma^{\mathcal{C}} \times [\![\tau]\!]\sigma'^{\mathcal{C}}$, so that $\sim^\tau$ are the smallest relations such that:

1. $c^{\underline{b}} \sim^{\underline{b}} c^{\underline{b}}$ for $c^{\underline{b}} \in \mathbf{D}^{\underline{b}}$,
2. $x \sim^{(a_1:\tau_1,\ldots,a_k:\tau_k)} y$ if $x(a_i) \sim^{\tau_i} y(a_i)$ for $i = 1,\ldots,k$,
3. $(a_i, x) \sim^{\langle\!|a_1:\tau_1,\ldots,a_k:\tau_k|\!\rangle} (a_j, y)$ if $i = j$ and $x \sim^{\tau_i} y$, and
4. $X \sim^{\{\tau\}} Y$ if for every $x \in X$ there is a $y \in Y$ such that $x \sim^\tau y$ and for every $y \in Y$ there is an $x \in X$ such that $x \sim^\tau y$.

A correspondence $\sim^{\mathcal{C}}$ is said to be **consistent** with instances $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and $\mathcal{I}' = (\sigma'^{\mathcal{C}}, \mathcal{V}'^{\mathcal{C}})$ if for each $C \in \mathcal{C}$ and all $o \in \sigma^C$, $o' \in \sigma'^C$, if $o \sim^C o'$ then $\mathcal{V}^C(o) \sim^{\tau^C} \mathcal{V}'^C(o')$. Note that the union of any family of consistent correspondences is also a consistent correspondence.

Let $\mathcal{I}$, $\mathcal{I}'$ be instances of a schema $\mathcal{S}$. Then ${}_{\mathcal{I}}{\approx}_{\mathcal{I}'}$ denotes the *largest* consistent correspondence between $\mathcal{I}$ and $\mathcal{I}'$. We call ${}_{\mathcal{I}}{\approx}_{\mathcal{I}'}$ the **bisimulation** correspondence between $\mathcal{I}$ and $\mathcal{I}'$.

Given any two instances $\mathcal{I}$ and $\mathcal{I}'$, we say $\mathcal{I}$ and $\mathcal{I}'$ are **bisimilar** and write $\mathcal{I} \approx \mathcal{I}'$ if and only if, for each $C \in \mathcal{C}$,

1. for each $o \in \sigma^C$ there is an $o' \in \sigma'^C$ such that $o \,{}_{\mathcal{I}}{\approx}_{\mathcal{I}'}\, o'$,
2. for each $o' \in \sigma'^C$ there is an $o \in \sigma^C$ such that $o \,{}_{\mathcal{I}}{\approx}_{\mathcal{I}'}\, o'$,

Proposition 2.1: The relation $\approx$ is an equivalence relation on the set of all instances $\mathbf{I}$ of a schema $\mathcal{S}$. ∎

Note that the relations $\approx$ and $\cong$ do not in general coincide: it is easy to construct two instances which are bisimilar but not isomorphic, for example by duplicating object identities. The instances illustrated in section 1 are an example of two instances that are bisimilar but not isomorphic.

We will see that, for a query language which does not include any means of directly comparing object-identities, observational indistinguishability coincides exactly with bisimulation of instances.

3 Querying the model

In this section we will present an adaption of the query language *SRI* ([5, 6]) to the model of section 2.2. The language is based on the mechanism of *structural recursion* over sets which was described in [5] as a basis for a query language on the nested relational data-model. Our choice of this mechanism is because its semantics are well understood and because it is known to be strictly more expressive than other formally developed query languages for nested relational model, such as the calculus of [2]. Consequently most of the results on the expresivity of various operators in this language paradigm will automatically carry over to other query language paradigms.

We will present two variants of the query language, *SRI* and *SRI*(=): the = representing the inclusion of the equality predicate on object identities.

The query language is described for a schema $\mathcal{S}$, with classes $\mathcal{C}$, such that $\mathcal{S}(C) = \tau^C$ for each $C \in \mathcal{C}$.

We expand our type system to allow object types, τ, as defined in section 2.1, and rank 1 function types, $\tau \rightarrow T$, where T is a (object or rank 1 function) type.

We assume base types *unit*, *Bool* with associated domains $\mathbf{D}^{unit} \equiv \{\emptyset\}$ and $\mathbf{D}^{Bool} \equiv \{\mathbf{T}, \mathbf{F}\}$, in addition other base types ranged over by $\underline{b}$, with associated domains $\mathbf{D}^{\underline{b}}$. (*Bool* is actually unnecessary since it is equivalent to a variant of units, but is included for convenience). For each other base type $\underline{b}$, and any value $c \in \mathbf{D}^{\underline{b}}$, we assume a corresponding constant symbol $\overline{c}$.

A **ground type** is an object type which contains no class types. Ground types are significant in that values of ground type are considered to be directly observable, while values of non-ground type will contain object identities, which do not have meaning outside of a particular instance. Further the set of values associated with a ground type will not be dependent on a particular instance, so that expressions of ground type can be evaluated in different instances, and their results can be compared.

A **query** is a closed expression of ground type.

The syntax and typing rules for *SRI* are given in figure 4. In *SRI*(=) we assume an additional binary predicate $=^C$ for each class $C \in \mathcal{C}$, with the typing rule

$$\frac{\vdash e_1 : C \quad \vdash e_2 : C}{\vdash e_1 =^C e_2 : Bool}$$

$=^C$ tests whether two terms of type C evaluate to the same object identity. The semantics for *SRI* and *SRI*(=) are given in appendix A.

3.1 Distinguishability of instances in *SRI*(=)

Two instances $\mathcal{I}$ and $\mathcal{I}'$ are said to be **indistinguishable** in some query language L iff, for any query q expressed in the language L, evaluating the query q for either of the two instances $\mathcal{I}$ and $\mathcal{I}'$ returns the same result.

In particular, two instances $\mathcal{I}$ and $\mathcal{I}'$ are **indistinguishable** in *SRI*(=) iff, for every ground type τ and query e such that $\vdash e : \tau$, $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$. ($V[\![\cdot]\!]$ is the semantic operator on *SRI* expressions defined in appendix A).

The following result tells us that isomorphism of instances exactly captures indistinguishability in *SRI*(=), and is therefore an important result in establishing the expressive power of *SRI*(=).

Theorem 3.1: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI*(=) if and only if they are isomorphic.

Products

$$\frac{\vdash e : (a_1 : \tau_1, \ldots, a_k : \tau_k)}{\vdash e.a_i : \tau_i} \qquad \frac{\vdash e_1 : \tau_1 \ \ldots \ \vdash e_k : \tau_k}{\vdash (a_1 = e_1, \ldots, a_k = e_k) : (a_1 : \tau_1, \ldots, a_k : \tau_k)}$$

Variants

$$\frac{\vdash e : \tau_i}{\vdash ins_{a_i}^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle}(e) : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle}$$

$$\frac{\vdash e : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle \ \vdash e_1 : \tau \ \ldots \ \vdash e_k : \tau}{\vdash case\, e\; of\, a_1(x_1^{\tau_1}) \Rightarrow e_1, \ldots,\ a_k(x_k^{t_k}) \Rightarrow e_k : \tau}$$

Sets

$$\frac{}{\vdash \emptyset^\tau : \{\tau\}} \qquad \frac{\vdash e_1 : \tau \ \vdash e_2 : \{\tau\}}{\vdash add(e_1, e_2) : \{\tau\}} \qquad \frac{\vdash e_1 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \ \vdash e_2 : \tau_2 \ \vdash e_3 : \{\tau_1\}}{\vdash sri(e_1, e_2, e_3) : \tau_2}$$

Functions

$$\frac{\vdash e : T_2}{\vdash \lambda x^{\tau_1} \cdot e : \tau_1 \rightarrow T_2} \qquad \frac{\vdash e_1 : \tau_1 \rightarrow T_2 \ \ \vdash e_2 : \tau_1}{\vdash e_1 e_2 : T_2}$$

Booleans

$$\frac{}{\vdash tt : Bool} \qquad \frac{}{\vdash f\!f : Bool} \qquad \frac{\vdash e_1 : Bool \vdash e_2 : \tau \vdash e_3 : \tau}{\vdash if(e_1, e_2, e_3) : \tau}$$

Base values

$$\frac{c \in \mathbf{D}^{\underline{b}}}{\vdash \overline{c} : \underline{b}} \qquad \frac{\vdash e_1 : \underline{b} \ \ \vdash e_2 : \underline{b}}{\vdash e_1 =^{\underline{b}} e_2 : Bool}$$

Others

$$\frac{}{\vdash x^\tau : \tau} \qquad \frac{}{\vdash () : unit} \qquad \frac{}{\vdash C : \{C\}} \qquad \frac{\vdash e : C}{\vdash !e : \tau^C}$$

Figure 4: Typing rules for query language

■

Proof: The if part is straightforward.

For the only-if part, given an instance $\mathcal{I}$ we construct an expression $e_{\mathcal{I}}$ such that $\vdash e_{\mathcal{I}} : Bool$ and $V[\![e_{\mathcal{I}}]\!]\mathcal{I}'$ is true iff $\mathcal{I}' \cong \mathcal{I}$. Details of the construction of $e_{\mathcal{I}}$ are given in Appendix B. ■

Claim: For any reasonable query language L, such that L supports an equality predicate on object identities, any two instances are indistinguishable in L if and only if they are isomorphic.

Justification: We need to show that, in any natural query language we can think of for this model, it is possible to construct an expression equivalent to the expression $e_{\mathcal{I}}$ from the proof of theorem 3.1. We observe that the constructors used in forming $e_{\mathcal{I}}$ do not go beyond those found the nested relational algebra of [7], the calculus of [2] without powerset, or what we would expect to find in any other query language. ■

The previous result tells us that, given any instance $\mathcal{I}$, there is a query which distinguishes $\mathcal{I}$ from any other non-isomorphic instance, but does not tell us how to find such a query without knowing exactly what the instance is already. Our next result tells us that, though any two non-isomorphic instances are distinguishable, it is not possible to find a single query or set of queries which are independent of the database instances, but which will distinguish between non-isomorphic instances. This means that, given two instances and a query interface or language such as *SRI*(=) for examining them, we can not in general decide whether or not the two instances are isomorphic, or find a query which distinguishes between them.

We must first define the notion of *Z-internal* functions on instances [8].

Suppose that ϕ is a function from instances of a schema $\mathcal{S}$ to some set D, and Z is a finite set of base values.

For each $v \in D$ we write $Supp(v)$ for the set of base values occuring in v (that is values in $\mathbf{D}^{\underline{b}}$ for some base type $\underline{b}$). Also we write $Supp(\mathcal{I})$ for the set of base values occuring in an instance $\mathcal{I}$. Then ϕ is said to be **Z-internal** iff for any instance $\mathcal{I}$, $Supp(\phi(\mathcal{I})) \subseteq Supp(\mathcal{I}) \cup Z$. That is, ϕ does not introduce any new base values, other than those in Z.

Lemma 3.2: For any closed $SRI(=)$ expression, e, there exists a finite set Z such that the mapping $V[\![e]\!]$ is Z-internal. ∎

Proof: Let $Const(e)$ denote the set of constants occuring in an expression e. We can show that $V[\![e]\!]$ is Z-internal where $Z = \{c | \overline{c} \in Const(e)\} \cup \{\mathbf{T}, \mathbf{F}\}$. It is sufficient to argue that there are no operators in the language which introduce new base values, other than predicates which may introduce the values **T** or **F**. More formally the result may be proved using induction on $SRI(=)$ expressions. ∎

Proposition 3.3: For any non-trivial schema $\mathcal{S}$, it is not possible to build a generic expression in $SRI(=)$ which tests whether two instances are isomorphic. In other words, given a schema $\mathcal{S}$, it is not possible to construct a value $e_{\mathcal{S}}$, depending only on $\mathcal{S}$, such that for any two instances $\mathcal{I}$ and $\mathcal{I}$, $V[\![e_{\mathcal{S}}]\!]\mathcal{I} = V[\![e_{\mathcal{S}}]\!]\mathcal{I}'$ iff $\mathcal{I}$ and $\mathcal{I}'$ are isomorphic. ∎

Proof: Suppose there is such a query e, and $\vdash e : \tau$. Then there is a finite Z such that $V[\![e]\!]$ is Z-internal. For any instances $\mathcal{I}$ and $\mathcal{I}'$, $[\![\tau]\!]\mathcal{I} = [\![\tau]\!]\mathcal{I}' = T$, where T is a possibly infinite set of values. However we can choose a finite set of base values, say W, such that there exist instances $\mathcal{I}$ with $Supp(\mathcal{I}) \subseteq W$. So, for any instance $\mathcal{I}$ with $Supp(\mathcal{I}) \subseteq W$, $V[\![e]\!]\mathcal{I} \in T$ and $Supp(V[\![e]\!]\mathcal{I}) \subseteq W \cup Z$. The set $\{v \in T \,|\, Supp(v) \subseteq W \cup Z\}$ is finite. However there are *infinitely many non-isomorphic instances,* $\mathcal{I}$, with $Supp(\mathcal{I}) \subseteq W$: given one such instance we can produce infinitely many more of them by introducing duplicates of object identities. It follows by a simple cardinality argument that e can not distinguish between these instances. ∎

Note that this proof requires only that $SRI(=)$ expressions be Z-internal for some finite Z. Consequently the result holds equally well for any other *pure* query language: that is, any language incorporating operators to extract, manipulate and compare data from an instance, but which cannot express general computations.

3.2 Computing bisimulation correspondence using *SRI*

Recall that the query language *SRI* is the same as $SRI(=)$, only without the $=^C$ predicates on object identities. So *SRI* gives us no way of directly comparing object identities.

Proposition 3.4: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI* iff $\mathcal{I} \approx \mathcal{I}'$. ∎

Proof outline: The proof consists of two parts. First we must prove that for any *SRI* query e, if $\mathcal{I} \approx \mathcal{I}'$ then $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$. This proof proceeds by induction on *SRI* expressions.

The second part of the proof is to show that, if $\mathcal{I}$ and $\mathcal{I}'$ are not bisimilar then they are distinguishable in *SRI*. Suppose $\mathcal{I} \not\approx \mathcal{I}'$. Then we can assume, with out loss of generality, that there is a class C and an object identity $o \in \sigma^C$ such that $o \not\approx^C o'$ for any $o' \in \sigma'^C$. We can then build a series of *SRI* functions, each of which unfolds object identities of class C to succesively greater depths, and show that, for any $o' \in \sigma'^C$, if $o \not\approx^C o'$ then there will eventually be an expression in this series which distinguishes between the two. For details of both parts of this proof see [12]. ∎

Claim: In any reasonably expressive query language, L, such that L does not support any means of directly comparing object identities, observational indistinguishability of instances in L will coincide prescisely with bisimilarity.

Justification: First note that *SRI* is at least as expressive as any other established query language which does not support comparisons of object identities. Consequently, if two instances are indistinguishable in *SRI* then they will also be indistinguishable in any other such language.

The proof of the second part of proposition 3.4 relies on being able to create queries which unfold nested values to any fixed finite height. We observe that any query language equipped with constructors and destructors for each of the basic types, basic logical operators and equality tests on each base type can express such finite unfoldings and tests of values. We claim that such operators will be present in any reasonable query language for nested or recursive data-structures. ∎

Using *SRI* (or some other reasonably expressive query language) we can also test for the bisimulation correspondence relation described in section 2.4 on individual values. That is, for any type τ, we can form a function expression $Cor^{\tau} : (\tau \times \tau) \rightarrow Bool$ such that, for any $u, v \in [\![\tau]\!]\mathcal{I}$, $V[\![Cor]\!]_{\mathcal{I}}(u, v) = True$ iff $u \approx_{\mathcal{I}} v$.

This result tells us that *SRI* has the same expressive power as *SRI*($\approx$) (the language *SRI* augmented with predicates for testing $\approx$).

This result is a little surprising since our values are recursive, and we can not tell how deeply we need to unfold two values in order to tell if they are bisimilar.

We are saved by the fact that all our object identities come from a fixed set of finite extents. The cardinality of these extents provide a bound on the number of unfoldings that must be carried out: if no differences between two values can be found after $\sum\{|C| \mid C \in \mathcal{C}\}$ dereferencings of object identifiers, then the values are equivalent. Consequently we can implement *Cor* by iterating over each class, and for each identifier in a class unfolding both values.

Unfortunately this implementation of $\approx$ seems to go against our philosophy of the non-observability of object identities: if we can't observe object identities then should we be able to count them? From a more pragmatic standpoint, a method of comparing values which requires us to iterate over all the objects in a database is far too inefficient to be practical, especially when dealing with large databases. We would like to know if we can test for $\approx$ without iterating over the extents of an instance. The following subsection will show that this is not possible.

N-bounded values and SRI^N

A value v is said to be **N-bounded** iff any set values occuring in v have cardinality at most N. An instance $\mathcal{I}$ is **N-bounded** iff for each class $C \in \mathcal{C}$ and every $o \in \sigma^C, \mathcal{V}^C(o)$ is N-bounded.

Note that, for any instance $\mathcal{I}$ there is an N sufficiently large that $\mathcal{I}$ is N-bounded.

We now define a variant of the language *SRI* which has the same power as *SRI* when restricted to N-bounded values, but which will not allow recursion over sets of cardinality greater than N.

The language SRI^N is the same as the language *SRI* except that an expression $sri(f, e, u)$ is not defined if $|V[\![u]\!]_{\mathcal{I}}|$ is greater than N.

Proposition 3.5: It is not in general possible to compute the correspondence relations $\approx$ on N-bounded instances using the language SRI^N. That is, there exists a schema $\mathcal{S}$ and type τ such that there is no expression *Cor* with $\vdash Cor : \tau \times \tau \rightarrow Bool$ such that $V[\![Cor]\!]_{\mathcal{I}}$ coincides precisely with $\approx^{\tau}$. ∎

Proof: First note that for any SRI^N expression e, there is a constant k^e, such that any evaluation of an application of e will involve less than k^e dereferences of objects. Consequently it is enough to construct a schema with a recursive structure such that, for any constant k, we can construct an instance containing two objects which require $k + 1$ dereferences in order to distinguish between them. ∎

This tells us that we can not hope to test if two values are equivalent using *SRI*, or any other reasonable query language, without making use of recursion over classes. We conclude that a more efficient mechanism for comparing values is needed.

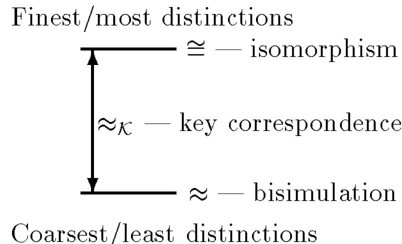


Figure 5: A spectrum of observational equivalence relations

4 Keys

We have seen that comparing database instances and values in instances involving object identities is problematic. On the one hand, we may consider only the *values* in a database to be significant and not wish to allow direct equality tests on object identities, since this would force us to distinguish between different representations of the same values. On the other hand, we have shown that computing bisimulation or value-based equivalence requires the ability to recurse over all the object identities in an instance. Such an equivalence relation is expensive to use in a query language over databases, and a more efficient means of comparing values is required.

A solution, common in many practical database systems, is to use *keys*: simple values that are associated with and used to compare object identities. Two object identities are taken to be equivalent iff their keys are equivalent. In a sense this can be thought of as computing an equvalence similar to $\approx$, but restricting the parts of the instance that are tested for comparison. However it is also possible to have *external keys* which depend not only on the value associated with a particular object in the database, but on other objects and values in the database as well.

In this section we will formalize the idea of keys, and show how they can determine equivalences on values that lie in between equality and bisimulation, as illustrated informally in figure 5. We show that, if a key specification satisfies certain *acyclicity* properties, then the resulting equivalence on values can be computed without resorting to recursion over the entire set of object identities.

4.1 Key specifications

Suppose we have a schema $\mathcal{S}$ with classes $\mathcal{C}$. A **key specification** for $\mathcal{S}$ consists of a type κ^C for each class $C \in \mathcal{C}$, and for any instances $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, a family of functions $\mathcal{K}^C_{\mathcal{I}} : \sigma^C \rightarrow [\![\kappa^C]\!]\mathcal{I}$ for each $C \in \mathcal{C}$. We write $\mathcal{K}^{\mathcal{C}}$ for such a key specification.

The idea is that, for any instance $\mathcal{I}$, $\mathcal{K}^C_{\mathcal{I}}$ will map object-identities of class C to their keys, and that any two object identities will be considered to be equivalent iff they have the same, or equivalent keys.

Example 4.1: Consider the schema described in example 2.1. We would like to say that a State is determined uniquely by its name, while a City is determined uniquely by its name and its state (one can have two Cities with the same name in different states). The types of our key specification are therefore

$$\begin{aligned}\kappa^{City} &\equiv (name : str, state : State)\\ \kappa^{State} &\equiv str\end{aligned}$$

For an instance $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ the mappings $\mathcal{K}^{\mathcal{C}}_{\mathcal{I}}$ are given by

$$\mathcal{K}^{City}_{\mathcal{I}}(o) \equiv \mathcal{V}^{City}(o)$$

$$\mathcal{K}_{\mathcal{I}}^{State}(o) \quad \equiv \quad (\mathcal{V}^{State}(o)).name$$

■

A key specification is said to be **well-defined** iff for any two instances, $\mathcal{I}$ and $\mathcal{I}'$, if $f^{\mathcal{C}}$ is an isomorphism from $\mathcal{I}$ to $\mathcal{I}'$, then for each $C \in \mathcal{C}$ and each $o \in \sigma^C$, $f^{\kappa^C}(\mathcal{K}_{\mathcal{I}}^C(o)) = \mathcal{K}_{\mathcal{I}'}^C(f^C(o))$

Well-definedness simply ensures that a key specification is not dependent on the particular choice of object identities in an instance, and will give the same results when applied to two instances differing only in their choice of object identities. We will assume that all key specifications we consider are well-defined.

Two key specifications, $\mathcal{K}^{\mathcal{C}}$ and $\mathcal{K}'^{\mathcal{C}}$, are said to be *equivalent* iff, for any instance $\mathcal{I}$, any $C \in \mathcal{C}$ and any $o_1, o_2 \in \sigma^C$, $\mathcal{K}_{\mathcal{I}}^C(o_1) = \mathcal{K}_{\mathcal{I}}^C(o_2)$ if and only if $\mathcal{K}_{\mathcal{I}}'^C(o_1) = \mathcal{K}_{\mathcal{I}}'^C(o_2)$.

The **dependency graph**, $G(\mathcal{K}^{\mathcal{C}})$, of a key specification $\mathcal{K}^{\mathcal{C}}$ is a directed graph with nodes $\mathcal{C}$ such that $G(\mathcal{K}^{\mathcal{C}})$ contains the edge (C', C) if and only if the class C' occurs in κ^C.

For example, the dependency graph of the key specification described in example 4.1 would have two nodes, *City* and *State*, and a single edge from *State* to *City*.

Proposition 4.1: For any key specification, $\mathcal{K}^{\mathcal{C}}$, if the dependency graph $G(\mathcal{K}^{\mathcal{C}})$ is acyclic then there is an equivalent key specification $\mathcal{K}'^{\mathcal{C}}$ such that each type κ'^C is ground (contains no classes). ■

We will see that key specifications with acyclic graphs are particularly useful later.

Language	Observational equivalences computable on values	Observational equivalence on instances
SRI(=) *SRI* with equality test on object-identities	$=^\tau$ — equality on all types	$\cong$ — isomorphism
SRI($\mathcal{K}$) $\mathcal{K}$ an acyclic key specification	$\approx_{\mathcal{K}}^\tau$ — key correspondence	$\approx_{\mathcal{K}}$ — key correspondence
SRI($\mathcal{K}$) $\mathcal{K}$ a general key specification	$\approx_{\mathcal{K}}^\tau$ — key correspondence (computing requires recursion over extents of object-identifiers)	$\approx_{\mathcal{K}}$ — key correspondence
SRI *SRI* with no comparisons on object-identities	$\approx^\tau$ — bisimulation (computing requires recursion over extents of object-identifiers)	$\approx$ — bisimulation

Figure 6: A summary of the operators considered and the resulting observational equivalences

4.2 Key correspondences

Given a key specification, $\mathcal{K}^{\mathcal{C}}$ and two instances $\mathcal{I}$ and $\mathcal{I}'$, we define the family of relations $\approx_{\mathcal{K}}^\tau \subseteq [\![\tau]\!]\mathcal{I} \times [\![\tau]\!]\mathcal{I}'$ to be the largest relations such that

1. if $c^{\underline{b}} \approx_{\mathcal{K}}^{\underline{b}} c'^{\underline{b}}$ for $c^{\underline{b}}, c'^{\underline{b}} \in \mathbf{D}^{\underline{b}}$ then $c^{\underline{b}} \equiv c'^{\underline{b}}$,
2. if $x \approx_{\mathcal{K}}^{(a_1:\tau_1,\ldots,a_k:\tau_k)} y$ then $x(a_i) \approx_{\mathcal{K}}^{\tau_i} y(a_i)$ for $i = 1, \ldots, k$,
3. if $(a_i, x) \approx_{\mathcal{K}}^{\langle\!|a_1:\tau_1,\ldots,a_k:\tau_k|\!\rangle} (a_j, y)$ then $i = j$ and $x \approx_{\mathcal{K}}^{\tau_i} y$,

4. if $X \approx_{\mathcal{K}}^{\{\tau\}} Y$ then for each $x \in X$ there is a $y \in Y$ such that $x \approx_{\mathcal{K}}^{\tau} y$ and for each $y \in Y$ there is an $x \in X$ such that $x \approx_{\mathcal{K}}^{\tau} y$, and

5. for each $C \in \mathcal{C}$ and any $o \in \sigma^C$, $o' \in \sigma'^C$, if $o \approx_{\mathcal{K}}^{C} o'$ then $\mathcal{K}_{\mathcal{I}}^{C}(o) \approx_{\mathcal{K}}^{\kappa^C} \mathcal{K}_{\mathcal{I}'}^{C}(o')$.

Note: For any schema $\mathcal{S}$, if we take the key specification given by $\kappa^C \equiv \tau^C$ for each $C \in \mathcal{C}$, and for any instance $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and each $C \in \mathcal{C}$, $\mathcal{K}_{\mathcal{I}}^{C} \equiv \mathcal{V}^C$ then the relations $\approx_{\mathcal{K}}^{\tau}$ and $\approx^{\tau}$ relations are the same. In general $\approx_{\mathcal{K}}$ may be *finer* than $\approx$ since we do not restrict the keys to be functions of the *values* associated with an object identity.

$\approx_{\mathcal{K}}^{\mathcal{C}}$ is called the *correspondence generated by* $\mathcal{K}^{\mathcal{C}}$.

Proposition 4.2: If $\mathcal{K}^{\mathcal{C}}$ is a key specification then, for any instance $\mathcal{I}$ and each type τ, $\approx_{\mathcal{K}}^{\tau}$ is an equivalence relation. ■

An instance $\mathcal{I}$ is said to be **consistent** with a key specification $\mathcal{K}^{\mathcal{C}}$ iff for each $C \in \mathcal{C}$, any $o, o' \in \sigma^C$, if $o \approx_{\mathcal{K}}^{C} o'$ then $\mathcal{V}^C(o) \approx_{\mathcal{K}}^{\tau^C} \mathcal{V}^C(o')$.

Suppose $\mathcal{K}$ is a key-specification for a schema $\mathcal{S}$. Given two instances of $\mathcal{S}$, say $\mathcal{I}$ and $\mathcal{I}'$, we say $\mathcal{I}$ is $\mathcal{K}$-equivalent to $\mathcal{I}'$, and write $\mathcal{I} \approx_{\mathcal{K}} \mathcal{I}'$ iff

1. For each $C \in \mathcal{C}$, each $o \in \sigma^C$ there is an $o' \in \sigma'^C$ such that $o \approx_{\mathcal{K}}^{C} o'$, and for each $o' \in \sigma'^C$ there is an $o \in \sigma^C$ such that $o \approx_{\mathcal{K}}^{C} o'$; and

2. For each $C \in \mathcal{C}$, $o \in \sigma^C$ and $o' \in \sigma' C$, if $o \approx_{\mathcal{K}}^{C} o'$ then $\mathcal{V}^C(o) \approx_{\mathcal{K}}^{\tau^C} \mathcal{V}'^C(o')$.

4.3 Keyed schema

A **keyed schema** is a pair consisting of a schema $\mathcal{S}$ and a key specification $\mathcal{K}^{\mathcal{C}}$ on $\mathcal{S}$. A **simply keyed schema** is a keyed schema $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$ such that the dependency graph of $\mathcal{K}^{\mathcal{C}}$ is acyclic.

An instance of a keyed schema $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$ is an instance $\mathcal{I}$ of $\mathcal{S}$ such that $\mathcal{I}$ is consistent with $\mathcal{K}^{\mathcal{C}}$.

Lemma 4.3: For any instances $\mathcal{I}$ and $\mathcal{I}'$ of a keyed schema $(\mathcal{S}, \mathcal{K})$, if $\mathcal{I} \approx_{\mathcal{K}} \mathcal{I}'$ then $\approx_{\mathcal{K}}$ is a consistent correspondence between $\mathcal{I}$ and $\mathcal{I}'$. ■

Proposition 4.4: For any two instances, $\mathcal{I}$ and $\mathcal{I}'$, of a simply keyed schema $(\mathcal{S}, \mathcal{K})$, if $\mathcal{I} \approx_{\mathcal{K}} \mathcal{I}'$ then $\mathcal{I} \approx \mathcal{I}'$. ■

4.4 Computing key correspondences

Given a keyed schema, $(\mathcal{S}, \mathcal{K})$, we define the language *SRI*$(\mathcal{K})$ for the schema to be the language *SRI* extended with new operators key^C for each $C \in \mathcal{C}$. The typing rules for these new operators are:

$$\frac{\vdash e : C}{\vdash key^C e : \kappa^C}$$

and the semantics are given in appendix A.

Similarly we define the language $SRI^N(\mathcal{K})$ as an extension of SRI^N.

We get the same results for computability of key correspondences, $\approx_{\mathcal{K}}$, as we did for bisimulation correspondence, namely

1. We can find a formula in $SRI(\mathcal{K})$ to compute $\approx^{\tau}_{\mathcal{K}}$ for each type τ.
2. We cannot in general find a formula to compute $\approx^{\tau}_{\mathcal{K}}$ on N-bounded values in SRI^N for any N.

However the following result goes some way towards justifying our earlier statement that key specifications with acyclic dependency graphs are of particular interest.

Proposition 4.5: For any simply keyed schema $(\mathcal{S}, \mathcal{K})$ there is an M such that for any $N \geq M$, and any type τ, $\approx^{\tau}_{\mathcal{K}}$ can be computed on N-bounded values using $SRI^N(\mathcal{K})$. That is, for each type τ, there is a formula $Cor^{\tau}_{\mathcal{K}}$ of $SRI^N(\mathcal{K})$ such than $\vdash Cor^{\tau}_{\mathcal{K}} : \tau \times \tau \rightarrow Bool$ and for any two N-bounded values $u, v \in [\![\tau]\!]\mathcal{I}$, $V[\![Cor^{\tau}_{\mathcal{K}}]\!]_{\mathcal{I}}(u, v) = \mathbf{T}$ iff $u \approx^{\tau}_{\mathcal{K}} v$. ■

It follows that acyclic key specifications provide us with an efficient means of comparing recursive values which incorporate object identities, without having to examine the object identities directly.

5 Conclusions

We have seen that there are a variety of different observational equivalences possible on recursive database instances using object identities, and that the observational equivalence relation generated by a particular query system is dependent on the means of comparing object identities available in that system. These range from equality tests on object identities, which in a suitable query language allow us to distinguish between non-isomorphic instances, to an absence of any means of comparing on object identities, which leads to a minimal observational equivalence of *bisimulation* in any reasonable query system. These results are summarized in figure 6.

Systems of *keys* generate various observational equivalences lying between these two. Use of keys, particularly acyclic key specifications, can provide an efficient method of comparing values in a query language without resorting to direct comparisons of object identities. We therefore believe that such systems of keys can play an important part in the development of practical languages for databases with object-identity.

We also saw that, by making use of the knowledge that object identities arise from *finite extents*, we can compute whether two values in a database are bisimilar, or key-equivalent, though in general we cannot compute these relations without using recursion over the extents of object identities. This raises the interesting question of what other, more general functions on recursive values can be computed using the knowledge of these finite extents, and is a topic for further research.

Acknowledgments: I would like to thank Susan Davidson, Catriel Beeri, Peter Buneman, Dan Suciu, Rona Machlin, Val Tannen, Leonid Libkin, Serge Abiteboul, Jan Van den Bussche and all at the Tuesday afternoon group for their help and advice in developing and presenting these ideas.

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[2] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[3] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[4] F. Bancilhon. Object-oriented database systems. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 152–162, Los Angeles, California, 1988.

[5] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[6] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[7] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[8] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):865–886, August 1986.

[9] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[10] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[11] Setrag N. Khoshafian and George P. Copeland. Object identity. In Stanley B. Zdonik and David Maier, editors, *Readings in Object Oriented Database Systems*, pages 37–46. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[12] Anthony Kosky. *Transforming Databases with Recursive Data Structures.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, November 1995.

A Semantics of $SRI(=)$

For each type τ we assume a countably infinite set of variables, $x^\tau, y^\tau, \ldots$. Let Var be the set of variables of the language $SRI(=)$. An *environment* for instance $\mathcal{I}$ is a partial mapping $\rho : Var \xrightarrow{\sim} \mathbf{D}(\mathcal{I})$ such that $\rho(x^\tau) \in [\![\tau]\!]\mathcal{I}$ for each variable x^τ of type τ.

We define the semantic function $V[\![\cdot]\!]$ from expressions of $SRI(=)$ and $\mathcal{I}$-environments to $\mathbf{D}(\mathcal{I})$ by

$$
\begin{array}{l}
V[\![e.a]\!]\mathcal{I}\rho \;\equiv\; (V[\![e]\!]\mathcal{I}\rho)(a) \\
V[\![(a_1 = e_1, \ldots, a_k = e_k)]\!]\mathcal{I}\rho \;\equiv\; (a_1 \mapsto V[\![e_1]\!]\mathcal{I}\rho, \ldots, a_k \mapsto V[\![e_k]\!]\mathcal{I}\rho) \\
V[\![ins_a e]\!]\mathcal{I}\rho \;\equiv\; (a, V[\![e]\!]\mathcal{I}\rho) \\
V[\![case\, e\; of\, a_1(x_1) \Rightarrow e_1, \ldots, a_k(x_k) \Rightarrow e_k]\!]\mathcal{I}\rho \;\equiv\; \left\{ \begin{array}{ll} V[\![e_1]\!]\mathcal{I}(\rho[x_1 \mapsto u]) & \text{if } V[\![e]\!]\mathcal{I}\rho = (a_1, u) \\ \vdots & \vdots \\ V[\![e_k]\!]\mathcal{I}(\rho[x_k \mapsto u]) & \text{if } V[\![e]\!]\mathcal{I}\rho = (a_k, u) \end{array} \right. \\
V[\![\emptyset]\!]\mathcal{I}\rho \;\equiv\; \{\} \\
V[\![add(e_1, e_2)]\!]\mathcal{I}\rho \;\equiv\; \{V[\![e_1]\!]\mathcal{I}\rho\} \cup V[\![e_2]\!]\mathcal{I}\rho \\
V[\![sri(e_1, e_2, e_3)]\!]\mathcal{I}\rho \;\equiv\; f(u_1, f(u_2, \ldots f(u_n, v)\ldots)) \text{ where } \begin{array}{l} V[\![e_1]\!]\mathcal{I}\rho = f \\ V[\![e_2]\!]\mathcal{I}\rho = v \\ V[\![e_3]\!]\mathcal{I}\rho = \{u_1, \ldots, u_n\} \end{array} \\
V[\![\lambda x \cdot e]\!]\mathcal{I}\rho \;\equiv\; (u \mapsto V[\![e]\!]\mathcal{I}(\rho[x \mapsto u])) \\
V[\![e_1 e_2]\!]\mathcal{I}\rho \;\equiv\; (V[\![e_1]\!]\mathcal{I}\rho)(V[\![e_2]\!]\mathcal{I}\rho) \\
V[\![tt]\!]\mathcal{I}\rho \;\equiv\; \mathbf{T} \\
V[\![ff]\!]\mathcal{I}\rho \;\equiv\; \mathbf{F} \\
V[\![if(e_1, e_2, e_3)]\!]\mathcal{I}\rho \;\equiv\; \left\{ \begin{array}{ll} V[\![e_2]\!]\mathcal{I}\rho & \text{if } V[\![e_1]\!]\mathcal{I}\rho = \mathbf{T} \\ V[\![e_3]\!]\mathcal{I}\rho & \text{otherwise} \end{array} \right. \\
V[\![\overline{c}]\!]\mathcal{I}\rho \;\equiv\; c \text{ where } c \in \mathbf{D}^{\underline{b}} \\
V[\![e_1 =^{\underline{b}} e_2]\!]\mathcal{I}\rho \;\equiv\; \left\{ \begin{array}{ll} \mathbf{T} & \text{if } V[\![e_1]\!]\mathcal{I}\rho = V[\![e_2]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{array} \right. \\
V[\![x]\!]\mathcal{I}\rho \;\equiv\; \rho(x) \\
V[\![()]\!]\mathcal{I}\rho \;\equiv\; \emptyset \\
V[\![C]\!]\mathcal{I}\rho \;\equiv\; \sigma^C \\
V[\![!e]\!]\mathcal{I}\rho \;\equiv\; \mathcal{V}^C(V[\![e]\!]\mathcal{I}\rho) \text{ where } V[\![e]\!]\mathcal{I}\rho \in \sigma^C \\
V[\![e_1 =^C e_2]\!]\mathcal{I}\rho \;\equiv\; \left\{ \begin{array}{ll} \mathbf{T} & \text{if } V[\![e_1]\!]\mathcal{I}\rho, V[\![e_2]\!]\mathcal{I}\rho \in \sigma^C \\ & \text{and } V[\![e_1]\!]\mathcal{I}\rho = V[\![e_2]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{array} \right.
\end{array}
$$

where $\emptyset$ denotes the unique value of type *unit*.

For $SRI(\mathcal{K})$ the semantics is extended with

$$V[\![key^C e]\!]^{\kappa^C}_{\mathcal{I}} \rho \equiv \mathcal{K}^C_{\mathcal{I}}(V[\![e]\!]^C_{\mathcal{I}}\rho)$$

In order for a structural induction (*sri*) expression to be well defined the first parameter must represent a function which is idempotent and commutative in its first argument. For further details of the semantic considerations for languages such as $SRI(=)$ see [6, 5].

B Proof of Theorem 3.1

First we will add some additional predicates and logical operators to the language *SRI*(=). These will not actually add to the expressive power of the language, but rather are *macros* or syntactic sugar for more complicated *SRI*(=) expressions. The syntax and typing rules for these new constructs are shown below.

Logical Operators

$$\frac{\vdash e_1 : Bool \quad \vdash e_2 : Bool}{\vdash e_1 \wedge e_2 : Bool} \qquad \frac{\vdash e_1 : Bool \quad \vdash e_2 : Bool}{\vdash e_1 \vee e_2 : Bool} \qquad \frac{\vdash e : Bool}{\vdash \neg e : Bool}$$

Quantifiers

$$\frac{\vdash e_1 : \{\tau\} \quad \vdash e_2 : Bool}{\vdash \exists x^\tau \in e_1 \cdot e_2 : Bool} \qquad \frac{\vdash e_1 : \{\tau\} \quad \vdash e_2 : Bool}{\vdash \forall x^\tau \in e_1 \cdot e_2 : Bool}$$

Predicates

$$\frac{\vdash e : \tau \quad \vdash e' : \tau}{\vdash e =^\tau e' : Bool} \qquad \frac{\vdash e : \tau \quad \vdash e' : \{\tau\}}{\vdash e \in^\tau e' : Bool}$$

The operators $\wedge$, $\vee$ and $\neg$ and the quantifiers $\exists, \forall$ have their normal meanings. The predicates $=^\tau$ represent an extension of the predicates $=^C$ to general types, and $\in^\tau$ represents using structural recursion to compare a value to each value in a set.

To simplify things we will assume that our schema, $\mathcal{S}$, involves only a single class C. The construction of the distinguishing expression works just as well for the case where $\mathcal{S}$ has multiple classes, though the nested subscripts and superscripts become rather unmanageable.

Suppose $\mathcal{I} = (\sigma^\mathcal{C}, \nu^\mathcal{C})$ is an instance of schema $\mathcal{S}$, such that

$$\sigma^C = \{o_1, \ldots, o_k\}$$

and

$$\nu^C(o_i) = p^i[o_{m^i_1}, \ldots, o_{m^i_{n^i}}]$$

where $o_{m^i_1}, \ldots, o_{m^i_{n^i}}$, are the object identities occuring in $\nu^C(o_i)$.

We will write $p^i[x_1, \ldots, x_{n^i}]$ for the *expression* formed by replacing each occurrence of $o_{m^i_r}$ by the variable x_r (There is need for an inductive definition for turning values into expressions here, which is straightforward).

Also we will use the shorthand expression $Dist(e_1, \ldots, e_n)$ defined by

$$Dist(e_1, \ldots, e_n) \equiv e_1 \neq e_2 \wedge \ldots \wedge e_1 \neq e_n \wedge e_2 \neq e_3 \wedge \ldots \wedge e_2 \neq e_n \wedge \ldots \wedge e_{n-1} \neq e_n$$

So $V[\![Dist(e_1, \ldots, e_n)]\!]\mathcal{I}\rho = \mathbf{T}$ iff the values $V[\![e_1]\!]\mathcal{I}\rho, \ldots, V[\![e_n]\!]\mathcal{I}\rho$ are distinct.

Now we can define $e_\mathcal{I}$ as follows:

$$\begin{aligned} e_\mathcal{I} \equiv\ & \exists x_1 \in C \cdot \ldots \exists x_k \in C \cdot \\ & Dist(x_1, \ldots, x_k) \wedge \\ & (\forall y \in C \cdot (y = x_1 \vee y = x_2 \vee \ldots \vee y = x_k)) \wedge \\ & (!x_1 = p^1[x_{m^1_1}, \ldots, x_{m^1_{n^1}}]) \wedge \ldots \wedge (!x_k = p^k[x_{m^k_1}, \ldots, x_{m^k_{n^k}}]) \end{aligned}$$

So $e_\mathcal{I}$ states first that there are n distinct elements of class C, which are bound to the variables $x_1, \ldots, x_n$, next that every object identity of class C is one of these n identities, and finally that the values associated with each of $x_1, \ldots, x_n$ correspond to the values associated with the object identities in the instance.

For any instance $\mathcal{I}'$ we now have $V[\![e_{\mathcal{I}}]\!]\mathcal{I}' = \mathit{True}$ iff $\mathcal{I}' \cong \mathcal{I}$.