TRANSFORMING DATABASES WITH RECURSIVE DATA STRUCTURES

Anthony Kosky

A DISSERTATION
in
COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.
1996

To my parents.

WARRANTY

Congratulations on your acquisition of this dissertation. In acquiring it you have shown yourself to be a computer scientist of exceptionally good taste with a true appreciation for quality. Each proof, algorithm or definition in this dissertation has been carefully checked by hand to ensure correctness and reliability. Each word and formula has been meticulously crafted using only the highest quality symbols and characters. The colours of inks and paper have been carefully chosen and matched to maximize contrast and readability.

The author is confident that this dissertation will provide years of reliable and trouble free service, and offers the following warranty for the lifetime of the original owner: If at any time a proof or algorithm should be found to be defective or contain bugs, simply return your dissertation to the author and it will be repaired or replaced (at the author's choice) free of charge. Please note that this warranty does not cover damage done to the dissertation through normal wear-and-tear, natural disasters or being chewed by family pets. This warranty is void if the dissertation is altered or annotated in any way.

Concepts described in this dissertation may be new and complicated. The author accepts no liability for any confusion or damage incurred during the reading and contemplation of the dissertation. Children under the age of five should not attempt to read this dissertation without proper adult supervision.

Comments, suggestions and personal abuse are all welcome and should be sent to the author via electronic mail.

ACKNOWLEDGMENTS

This dissertation marks the end of six years which I spent engaged in studies and research at the Department of Information and Computer Science of the University of Pennsylvania. Though only a part of that time was spent directly on the work described in this dissertation, it nevertheless reflects many influences, both from my time at Penn and from my studies prior to that in England. There are many people to thank, both for their direct contributions to this work, and also for their roles in developing my understanding and appreciation of theoretical computer science, databases, programming languages, and many other subjects of relevance.

Firstly I would like to thank my advisors, Peter Buneman and Susan Davidson for their help, suggestions, support, advice and encouragement, and for introducing me to the subject of databases. Peter was also responsible for giving me the opportunity to enroll in a PhD program at Penn. I would like to thank my committee members, Tim Griffin, Victor Markowitz, Carl Gunter, Val Tannen and Chris Overton for their comments and advice. This work has also been influenced greatly by the discussions of the "Tuesday afternoon group" including Leonid Libkin, Limsoon Wong, Dan Suciu, Rona Machlin, Wenfei Fan and Kyle Hart. I would especially like to thank Leonid for his many helpful comments and advice, and for his thorough reading of the proposal for this dissertation. Barbara Eckman and Carmem Hara did much of the work on the trials of the prototype transformation system described in part IV. Barbara also helped to explain the Molecular Biology Databases and the database problems that inspired much of this work. I am also grateful to Catriel Beeri, Jan Van den Bussche and Serge Abiteboul for their comments on my other papers related to this work. Edward T. Bear gave consistent support and encouragement, and helped with some of the more technically difficult proofs in this dissertation.

One of the most enjoyable aspects of my research at Penn was the collaboration with members of the computational biology group, not only because it gave me an opportunity to look at some practical applications for my work, but also because it gave me a chance to learn a little about the fascinating subjects of molecular biology and genetics. I would like to thank Chris Overton and David Searls for sharing there enthusiasm for these subjects, and for their many impromptu biology lessons.

There are also many people who have contributed to my development first as a mathematician and then as a computer scientist. I would like to thank the lecturers of the Department of Mathematics at the University of Kent at Canterbury, in particular John Earl, who helped me to develop an appreciation for the beauty of pure mathematics. My introduction to computer science came when I did a Masters degree at the Department of Computing at Imperial College of Science and Technology. In particular I was introduced to the subjects of formal methods and functional programming by the lectures of Samson Abramsky, Mike Smyth, Steve Vickers, Pete Harrison, Chris Hankin and others. Samson Abramsky also supervised my masters thesis and recommended me as a possible PhD student at the University of Pennsylvania, for which I am especially grateful. My knowledge and appreciation of theoretical computer science has been extended further while at the Penn, through the lectures Val Tannen, Carl Gunter, Scott Weinstein, Peter Freyd and others.

Many of the staff at the University of Pennsylvania have helped me in dealing with bureaucracy and various administrative details. I would particularly like to thank Mike Felker who's help allowed me to finish off and co-ordinate this PhD while working in California. I would also like to thank Karen Carter, Nan Blitz, Susan Deysher, Elaine Benedetto and Jackie Caliman, and members of the computing staff Mark Foster, Mark-Jason Dominus and Alex Garthwaite.

There are also many people who helped in making my time at Penn enjoyable, and helping me to maintain a semblance of sanity. I would like to thank the Old Quaker Computer Scientists for some very bizarre and amusing times, the Penn Magic play-testers, the Saturday-morning Reading Terminal crowd, and all at Bicycle Therapy for keeping my bikes running nicely.

Finally, but most importantly of all, I would like to thank my parents and my family. Their love, support and encouragement have been a constant comfort to me in spite of the long distances between us, and I could not have achieved any of this without them.

ABSTRACT

TRANSFORMING DATABASES WITH RECURSIVE DATA STRUCTURES

Anthony Kosky
Advisors: Susan Davidson and Peter Buneman.

This thesis examines the problems of performing structural transformations on databases involving complex data-structures and object-identities, and proposes an approach to specifying and implementing such transformations.

We start by looking at various applications of such *database transformations*, and at some of the more significant work in these areas. In particular we will look at work on transformations in the area of *database integration*, which has been one of the major motivating areas for this work. We will also look at various notions of correctness that have been proposed for database transformations, and show that the utility of such notions is limited by the dependence of transformations on certain implicit database constraints. We draw attention to the limitations of existing work on transformations, and argue that there is a need for a more general formalism for reasoning about database transformations and constraints.

We will also argue that, in order to ensure that database transformations are well-defined and meaningful, it is necessary to understand the information capacity of the data-models being transformed. To this end we give a thorough analysis of the information capacity of data-models supporting object identity, and will show that this is dependent on the operations supported by a query language for comparing object identities.

We introduce a declarative language, *WOL*, based on Horn-clause logic, for specifying database transformations and constraints. We also propose a method of implementing transformations specified in this language, by manipulating their clauses into a *normal form* which can then be translated into an underlying database programming language.

Finally we will present a number of optimizations and techniques necessary in order to build a practical implementation based on these proposals, and will discuss the results of some of the trials that were carried out using a prototype of such a system.

Contents

Foreword

The theme of this thesis is to study the problems associated with performing structural transformations on databases involving complex data-structures and object identity, and to propose an approach to specifying and implementing such transformations based on a declarative language. Such transformations arise from a number of areas including database integration, schema evolution and implementing user views and data entry applications.

The language we propose, called *WOL* (Well-founded Object Language), is a declarative language based on Horn-clause logic, which allows us to express database transformations and constraints in a single framework, and thus allows us to reason about interactions between the two. *WOL* differs from established Horn-clause based languages, such as *Prolog* or *Datalog*, in that it is designed to deal with partial descriptions of database objects: a necessary facility when dealing with very large and complex data-structures. We will propose a method of implementing database transformations specified using *WOL* by first converting the clauses of a transformation program to a *normal form*, and then translating the normal form clauses to an underlying database programming language.

We will also look at a variety of related issues, including *database constraints*, and the *information capacity* of data-models involving object identity. In the case of the first, we will argue that there is an important interaction between database transformations and constraints, in that constraints play a part in determining transformations, but also in that, in order for a transformation program to be correct, various constraints may be implied on the source and and target databases of the program.

We are concerned with transformations of databases involving complex, arbitrarily deeply nested and possibly self-referential data-structures. In order to represent such data-structures we require a data-model equipped with a rich variety of type constructors, and also some sort of reference mechanism such as pointers, object identities or variables defined via systems of equations. We will pursue a model similar to that of [2], based on the notion of object-identity and incorporating set, variant and record type constructors. However our choice of using object-identities is largely a stylistic one, and models based on any alternative reference mechanism would yield similar results: indeed, in some sense, this point could be considered the theme of part II of this thesis. In addition a central feature of our data-model is the notion of *finite* *extents*: that is that a database instance consists of a collection of known, finite extents, and all values in the instance arise from those extents. This point, though it may seem trivial, is in fact a major distinguishing factor between databases and other areas of computation, in that allows us to compute a variety of functions on instances which would not be decidable in a more general computational framework.

1.1 A Roadmap

This thesis covers a variety of aspects of database transformations, and related issues, from a variety of perspectives. Since the interests of readers may vary, I will attempt to provide a guide allowing the reader to concentrate on those parts of the thesis of interest to him or her, while avoiding those parts which are not. The thesis is divided into four Parts, each of which is equipped with its own introductory section, which will motivate the Part and also include a more detailed roadmap. Here I will provide a brief overview of each Part, and also pointers to sections that are of a particular importance to the remainder of the thesis.

Part I serves as a general introduction to the field of database transformations, and a survey of some of the more significant work in this field. We start by describing the various different areas in which database transformations arise, and the different possible approaches to implementing them. In section 3 we look specificly at work done in the area of *database integration*. Database integration is the area from which the most significant work on transformations has arisen, and also the main motivating area for my work in this subject. In section 4 we discuss the requirements on data-models for use in database transformations. In section 5 we examine the various notions of correctness that have been proposed for database transformations. In particular we describe various notions of *information dominance*, and show that in order to ensure the correctness of a database transformation, it is necessary to consider constraints on the source database which may not be explicit in the database schemas.

In Part II we will give a formal examination of the information capacity of data-models supporting object identity. It is essential to understand these issues in order to provide a formal treatment of database transformations: transformations must respect the information represented by instances, and should map equivalent instances to equivalent instances. We will argue that the information represented by a database instance coincides precisely with its observable properties, and show that the observational equivalence of instances is determined by our assumptions about what operations are available in a query language for comparing object identities.

In section 7 we will introduce our data-model, and in section 10 we will extend this model with the notion of *keys*. It is recommended that a reader who is not concerned with the discussions of Part II should nevertheless read section 7 and the first part of section 10, since these will be necessary in order to understand the remaining parts of the thesis.

In Part III we will present an approach to specifying and implementing transformations based on the *WOL* language. In section 13 we will introduce the *WOL* language, defining its syntax and semantics, and giving examples to show how *WOL* can be used to express a variety of database constraints. In section 14 we will show how *WOL* can be used to express transformations, and describe the idea of *normal form* transformation programs which may be implemented efficiently using some underlying database programming language. In section 15 we will show how *WOL* may be extended to deal with alternative collection types such as bags and lists.

In Part IV we will present information necessary in order to implement transformations using *WOL*. We will present various optimizations necessary in order to make the algorithms constructed in Part III feasible, and will show how two-stage transformations can be used to avoid potential exponential blowups in the size of transformations programs. We will also describe our work on a prototype implementation of *WOL* and our experiences with trials using this implementation.

1.2 Some Comments on the Mathematical Approach and Assumptions

Though the work described in this thesis was driven largely by pragmatic problems and by intuitions about these problems, its foundations nevertheless lie in formal mathematics and logic. As such, each result in the thesis requires a formal proof, and each concept a formal definition, built on the proofs and definitions that have gone before. However many of the required proofs are similar or almost identical, and some proofs are repetitive and involve many similar cases. Writing out every detail of such proofs would be extremely time consuming, and the resulting document would lack continuity and be very difficult to read. Consequently, in the case of many similar proofs, I may choose to give the first proof in detail, and only outline later proofs, or concentrate on the details particular to each later proof. In the case of proofs involving large numbers of similar cases, I may only provide details of some representative cases, and concentrate on any cases which require special treatment. In all cases I will provide sufficient details for the reader to completely reconstruct a proof by comparison and adaption of other proofs in the thesis.

I have tried to make this work as general as possible, and to avoid simplifying assumptions and restrictions when ever it was feasible to do so. In some cases this generality may lead to excessive complexity, and simpler but slightly more restrictive approaches may be suitable for most pragmatic purposes. In such cases I have tried to first provide the most general versions of the work, and then to describe the simplified versions and explain the restrictions they entail.

There are a number of pervasive assumptions about the underlying mathematical models that I am dealing with. In particular I will assume a universe of *sets* which supports constructors for finite products, power sets and finite power sets, and function spaces. I will make the slightly non-standard assumption that function spaces and products are disjoint: if X and Y are sets, I will assume that $X \to Y$, the set of functions from X to Y, and $X \times Y$, the set of pairs with first element taken from X and second element from Y, are disjoint sets.

There are also a number of notational conventions and abbreviations which will occur sufficiently often that they warrant mentioning. *Finitely indexed families* of sets or functions will occur frequently. For example, $\mathcal{C}$ might be some finite set, and for each $C \in \mathcal{C}$ there might be a corresponding set σ^C. In such a case I will write $\sigma^{\mathcal{C}}$ for the *family* of sets σ^C for $C \in \mathcal{C}$.

I will also frequently make use of *ellipses*, "…", in order to represent sequences of similar terms or expressions. For example I would write $a_1 : \tau_1, \ldots, a_k : \tau_k$ to represent a series of k terms separated by commas, where the ith term has the form $a_i : \tau_i$. Such notational conventions will greatly simplify our presentation.

Part I
Database Transformations

2 Introduction

A *database transformation* is a mappings from the instances of one or more *source* database schemas to the instances of some *target* schema. The terms "schema" and "instance" are used here in a very general sense, to mean an abstract description of the structure of data in a database, and the data stored in a database at a particular moment in time respectively. The precise interpretation of these terms is dependent on the particular data-models being considered: the schemas involved may be expressed in a variety of different data-models, and implemented using different DBMSs.

The need to implement transformations between distinct, heterogeneous databases has become a major factor in information management in recent years. Such transformations arise from a number of different sources, including:

Database integration: where data from a number of distinct heterogeneous databases is mapped into a local database, or made available through a federated database system, to give the impression of a single unified database;

Schema evolution: where changes in the concepts being modeled and the tasks for which a database is used result in changes in the database schema, and it is necessary to transform existing data so that it conforms to a new, evolved schema; and

User views and data-entry applications: where the format in which data is entered into a system or viewed are substantially different from the format in which data is actually stored, and so it is necessary to transform data between these formats.

Incompatibilities between the sources and target exist at all levels – the choice of data-model, the representation of data within a model, and the data within a particular instance – and must be explicitly resolved within the transformations. The wide variety of data models in use, including those supporting complex data structures and object-identities, further complicate these problems.

Much of the existing work on transformations concentrates on the restructuring of source database schemas into a target schema, either by means of a series of simple manipulations or by a description in some abstract language, and the mappings of the underlying instances are determined by the restructurings of schemas. In some cases this emphasis is at the expense of a formal treatment of the effect of transformations on instances, which is stated informally or left to the intuition. However there are, in general, many possible interpretations of a particular schema manipulation. For example, in a data model supporting classes of objects and optional attributes of classes, suppose we changed an attribute of an existing class from being optional to being required. There are a number of ways that such a schema manipulation can be reflected on the underlying data: we could insert a default value for the attribute where ever it is omitted, compute a new value based on the other attributes, or we could simply delete any objects from the class for which the attribute is missing.

It is clear that there may be many transformations, with differing semantics, corresponding to the same schema manipulation, and that it is necessary to be able to distinguish between them. In contrast to existing work, our focus in this thesis is therefore on how transformations effect the underlying data itself. We will use the term "*database transformations*", as opposed to the more common "schema transformations", in order to emphasize this distinction.

In the remainder of this section we will describe the different methods used for implementating database transformations. In section 3 we will examine database transformations as they arise in the field of database integration. We will present an example and show how various existing approaches might be used to address this example, and where the limitations of such existing work lie. In section 4 we discuss the requirements on a data-model for database transformations, and motivate our choice of an object-identity based model with set, variant and record type constructors. In section 5 we will discuss the various notions of correctness that have been proposed for database transformations. In particular we will describe Hull's hierarchy of notions of information dominance [23], and more recent work linking information dominance to database constraints. We will argue that, in order to satisfy such correctness criteria, a transformation may imply constraints on the databases involved which have been left implicit, or may not be expressible in the datamodels used, and consequently the usefulness of such notions of correctness is limited. We conclude that there is a need for general formalism for expresing database transformations and constraints, and for reasoning about the interactions between the two.

2.1 Methods of Implementing Database Transformation

Implementations of database transformations fall into two camps: those in which the data is actually transformed into a format compatible with the target schema and then stored in a target database, and those in which the data remains stored in the source databases and queries against the target schema are translated into queries against the source databases. The first of these approaches can be thought of as performing a *one-time bulk transformation*, while the second approach evaluates transformations in a *call-by-need* manner. Which of these methods is most appropriate depends on the purpose of a particular transformation, and also the nature of the underlying databases: their size and complexity, how rapidly they change, how difficult it is to access them remotely, and so on.

For example, the most common approach adopted within federated database systems [22] is call-by-need [32, 17, 29]. This approach has the advantage that the source databases retain their autonomy, and updates to the various source databases are automatically reflected in the target database. However, in cases where accessing the component databases is costly and the databases are not frequently updated, actually merging the data into a local unified database may be more efficient. Furthermore, maintaining integrity constraints over a federated database system is a much more difficult task than checking data integrity for a single merged database [38, 48]. As a result, the approach of performing a one-time bulk transformation is taken in [47].

Some work on schema evolution also advocates implementing transformations in a call-by-need manner [7, 41, 44]. In this case multiple versions of a schema are maintained, and data is stored using the version for which it was originally entered. The advantage of this approach is that major database reorganizations can be avoided, and applications implemented for an earlier version of a schema can still be used. However, for applications built on old versions of a schema to be applied to new data, reverse transformations must also be implemented. Furthermore the cost of maintaining multiple views and computing compounded transformations may be prohibitively expensive. These problems are especially significant when schema evolutions are frequent, and it is not possible a priori to tell when old views or data cease to be relevant. Consequently some practical work on implementing schema evolutions has been based on performing bulk transformations of data [35].

It is clear that the implementation method appropriate for a particular transformation will depend on the application and on the databases involved. However, the semantics of a transformation, that is the effect of a transformation on the underlying data, should be independent of the implementation method chosen as well as of the application area itself. Unfortunately, for much of the work in the area of database transformations this is not the case, primarily due to the fact that there is no independent model or characterization for the semantics of transformations. One of the aims of this thesis is therefore to develop a semantics of database transformations, and examine various metrics for the "goodness" or "correctness" of such transformations.

3 Transformations in Database Integration

In this section we will look at some examples of database transformations, particularly in the context of database integration, and show how some parts of these examples are addressed by existing work, while others require more general transformation techniques. The context of database integration is particularly appropriate since much of the most significant work in database transformations stems from this field. In contrast, transformations proposed in say the area of schema evolution are comparatively simple [36, 44, 35, 7, 41], normally being based a single model and a small set of basic schema modifications (such as introducing specialization and generalization classes, adding or removing attributes, and so on). It is not clear whether the reason for this is historical, since database integration became a significant problem earlier in terms of needing formal tools and techniques, or because the transformations involved in database integration are inherently more difficult than those arising in other areas.

3.1 Database Integration: An Example

The objective of *database integration* is to make data distributed over a number of distinct, heterogeneous databases accessible via a single database interface, either by constructing a (virtual) view of the component databases to give them the appearance of a single database, or by actually mapping data from the component databases into a single unified database. In either case, the problem from the perspective of database transformations is how to transform data from the various formats and structures in which it is represented in the component databases into a form compatible with the integrated database schema.

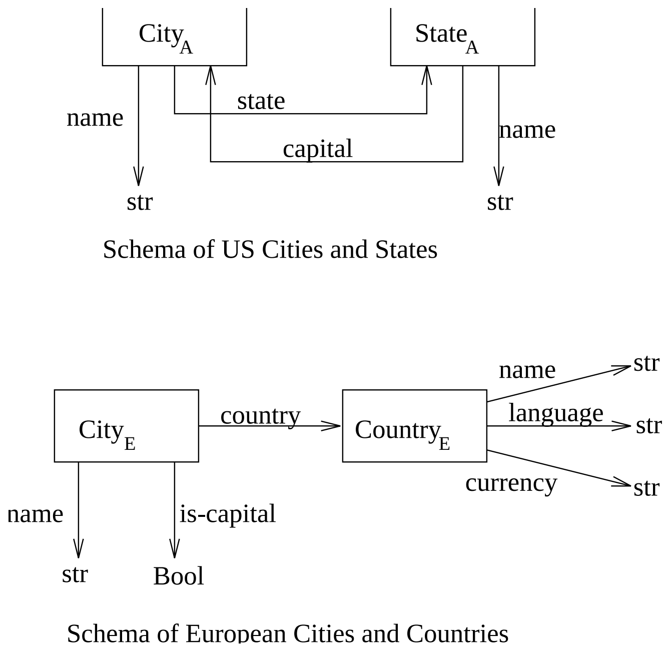


Figure 1: Schemas for US Cities and European Cities databases

Example 3.1: Figure 1 shows the schemas of two databases representing US Cities and States, and European Cities and Countries respectively. The graphical notation used here is inspired by [5]: the boxes represent *classes* which are finite sets of objects; the arrows represent *attributes*, or functions on classes; and *str* and *Bool* represent sets of *base values*. An instance of such a schema consists of an assignment of finite sets of objects to each class, and of functions on these sets to each attribute. The details of this model will be made precise in section 7.

The first schema has two classes: *City* and *State*. The City class has two attributes: *name*, representing the name of a city, and *state*, which points to the state to which a city belongs. The *State* class also has two attributes, representing its name and its capital city.

The second schema also has two classes, this time *City* and *Country*. The *City* class has attributes representing its name and its country, but in addition has a Boolean-valued attribute *capital* which represents whether or not it is the capital city of a country. The *Country* class has attributes representing its name, currency and the language spoken.

Suppose we wanted to combine these two databases into a single database containing information about both US and European cities. A suitable schema is shown in figure 2, where the "plus" node indicates a variant. Here the *City* classes from both the source databases are mapped to a single class *City* in the target database. The *state* and *country* attributes of the *City* classes are mapped to a single attribute *place* which take a value that is either a *State* or a *Country*, depending on whether the *City* is a US or a European city. A more difficult mapping is between the representations of capital cities of European countries. Instead of representing whether a city is a capital or not by means of a Boolean attribute, the *Country* class in our target database has an attribute *capital* which points to the capital city of a country. To resolve this difference in representation a straightforward embedding of data will not be sufficient; we will need to do some more sophisticated structural transformations on the data(see section 3.2). Further constraints on the source database, ensuring that each *Country* has exactly one *City* for which the *is_capital* attribute is true, are necessary in order for the transformation to be well defined. (The interaction between constraints and transformations will be explored in section 5.2). ■

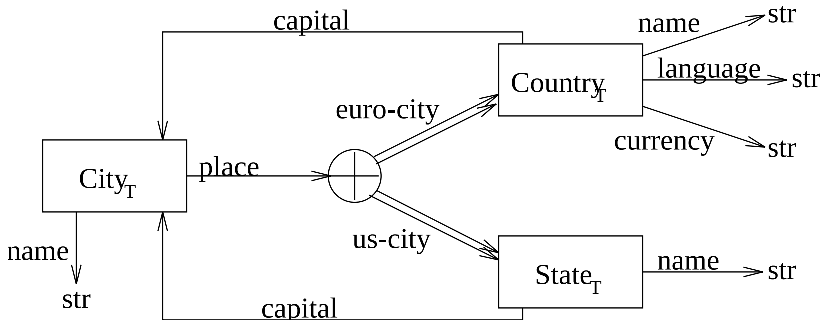


Figure 2: An integrated schema of European and US Cities

The problem of database integration is therefore to define an *integrated schema*, which represents the relevant information in the component source databases, together with transformations from the source databases to this integrated schema.

3.2 Resolving Structural Conflicts in Database Integration

In [9] Batini et al noted that schema integration techniques generally have two phases: conflict resolution and merging or unioning of schemas. Although schema merging has received a great deal of attention, it is only a small (and usually the last) step in the process of database integration. The more significant part of the process is manipulating the component databases so that they represent data in a compatible way. In order to do this it is necessary to resolve naming conflicts between the schemas (both homonyms and synonyms), and also to perform structural manipulations on data to resolve conflicts in the way data are represented. An example of such a structural manipulation was given by how the capital attribute was represented in the European Cities schema.

The order in which the conflict resolution and schema merging phases are carried out varies between different database integration methods. For example, in Motro[32] component schemas are first unioned to form disjoint components of a "superview" schema, and the superview is then manipulated in order to combine concepts and resolve conflicts between the component schemas. In contrast, [33, 13, 40] assume that conflicts between schemas are resolved prior to the schema merging process, and [8] interleaves the two parts of this process.

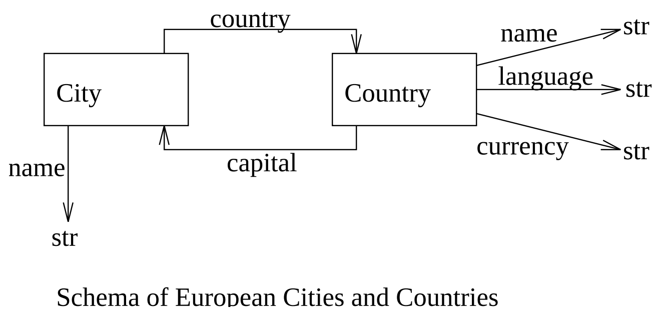


Figure 3: A modified schema for a European cities and countries database

Example 3.2: Returning to example 3.1, it is necessary to perform a structural modification on the database of European Cities and Countries to replace the Boolean *is_capital* attribute of the *City* class with a *capital* attribute of class *Country* going to the class *City*. This yields an intermediate database with the schema shown in figure 3. It is then necessary to associate the classes and attributes of the two source databases, so that the *City* classes and *name* attributes, and also the *state* and *country* attributes, are associated, and the remainder of the transformation could be implemented by means of an automated schema-merging tool. ■

There are two basic approaches to systems for implementing transformations to resolve such structural conflicts: using a small set of simple transformations or heuristics that can be applied in series [32, 7, 31, 40], or using some high-level language to describe the transformation [1, 17]. Examples of these two approaches will be given in section 3.3 (examples 3.3 and 3.4).

The advantage of using a small set of pre-defined atomic transformations is that they are simple to reason about and prove correctness for (notions of correctness for database transformations is the subject of section 5). For instance, one could prove that each transformation was information preserving [37, 31], or if necessary associate constraints with each transformation in order for it to be information capacity preserving, and deduce that a series of applications of the transformations was information preserving. The disadvantage of this approach is that the expressivity of such a family of transformations is inherently limited. For example the family of transformations proposed in [32] are insufficient to describe the transformation between an attribute of a class and a binary relation between classes: that is, one cannot transform from a class *Person* with an attribute *spouse* of class *Person* to a binary relation *Marriage* on the *Person* class. Although it might be easy to extend the family of atomic transformations to allow this case, which is a common source of incompatibility between databases, there would still be other important transformations that could not be expressed. The restructuring described in example 3.2 can also not be expressed using any of the families of transformations mentioned above.

A potentially much more flexible approach is to use some high-level language for expressing structural transformations on data. However each transformation expressed in such a language must be programmed and checked individually. Further, if it is necessary to ensure that a transformation is information preserving then additional constraints may be needed on the source databases, and in general these constraints will not be expressible in any standard constraint language. This point will be taken up again in more detail in section 5. We therefore believe that there is a need for a declarative language for expressing such transformations and constraints, which allows one to formally reason about the interaction between transformations and constraints and which is sufficiently simple to allow transformations to be programmed easily. Such a language will be presented in section 13.

3.3 Schema Integration Techniques

In [9] Batini et al. survey existing work on *schema integration.* They observe that schema integration arises from two tasks: *database integration*, which we have already discussed, and *integration of user views*, which occurs during the design phase of a database when constructing a schema that satisfies the individual needs of each of a set of user groups. However they fail to note that these two kinds of schema integration are fundamentally different. The reason for this can be seen by considering the direction in which data is transformed in each case. For database integration, instances of each of the source databases are transformed into instances of the merged schema. On the other hand, when integrating multiple user views, instances of the merged schema must be transformed back into instances of the user views (see figure 4). The intuition is that when integrating user views *all* of the underlying information must be represented; no objects or attributes can be missing since some user may want the information. However, when integrating pre-existing databases the best that can be hoped for is that attributes of objects that are present in every underlying database will definitely be present in the integration; attributes that are present in some but not all of the underlying databases may be absent in the integration. In [13] it was observed that integrating user views corresponds to the "least upper bound" of the component schemas in some information ordering on schemas, while in database integration what is required is the "greatest lower bound" of the component schemas in some information ordering on schemas. A good schema-integration method should therefore take account of its intended purpose and include a semantics for the underlying transformations of instances.

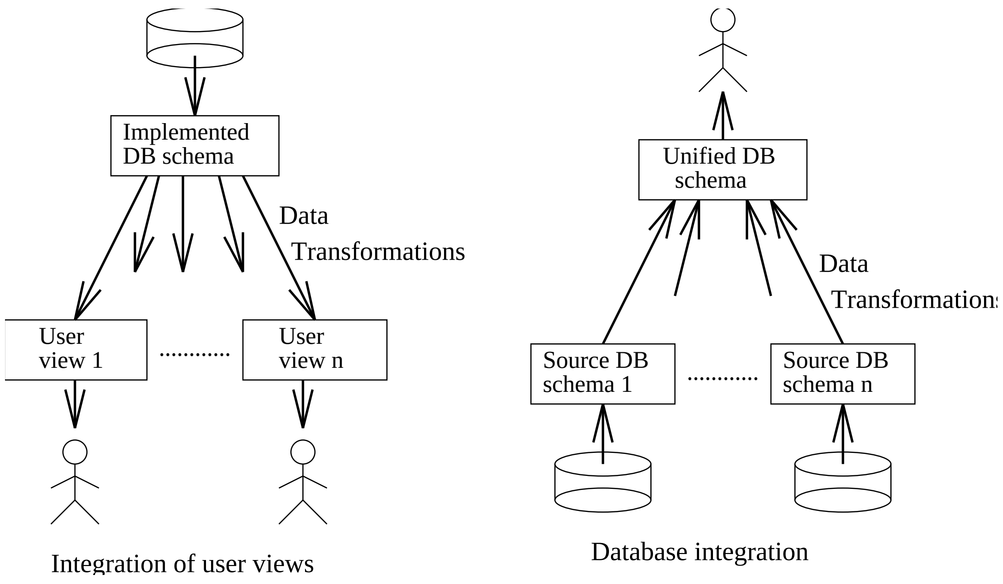


Figure 4: Data transformations in applications of schema integration

In this section we will concentrate on methodologies intended for database-integration, and look at some representative examples of the various approaches to this problem.

Example 3.3: Continuing with our example of database integration, we can use the technique of Motro[32] to integrate the *Cities* and *States* database of figure 1 with the restructured *Cities* and *Countries* database of figure 3.[1] The process is illustrated in figure 5. First, a disjoint union of the two schemas is formed (a), and then a series of "macro" transformations are applied to form the desired integrated schema.[2] The transformations applied include introducing generalizations (b), deriving new attributes as compositions or combinations of existing attributes (c), and combining classes (d). ■

In this particular integration method, the semantics of the transformations are strongly linked to the implementation method. The intention is that the integrated database be implemented as a *view* of the component databases, and that queries against the integrated database be executed by translating them into queries against the component databases and then combining the results: the semantics of the individual transformations are given by their effects on queries. However the lack of any independent characterization of their semantics makes it difficult to reason about or prove properties of the transformations, or to use any alternative implementation of the methodology.

[1]Recall that this methodology is not expressive enough to express the transformation from the *Cities* and *Countries* database of figure 1 to that of figure 3.

[2]In the model of [32] generalizations are represented by classes with *isa* edges, though for consistency we present this example using variants instead.

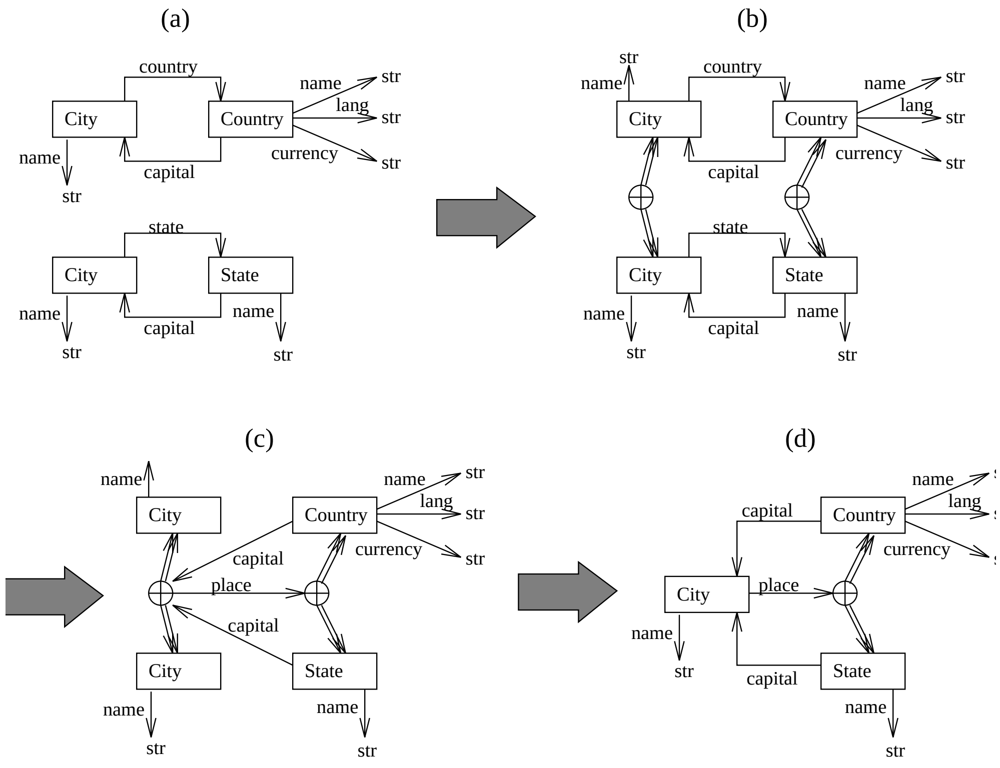


Figure 5: A schema integration using the methodology of Motro

A more expressive and flexible way of specifying transformations is to use some sort of high-level transformation language. An example of such an approach is the system of rewrite rules for nested relational structures proposed by Abiteboul and Hull in [1].

Example 3.4: We will show how the rewrite rules of [1] can be used to represent the mapping from the European Cities and Countries database of figure 1 to the integrated schema of figure 2. The transformation is defined by a series of *rewrite rules*:

$$\begin{aligned}
\rho_{capitals} &\equiv \text{rew}((\mathit{name} : X,\ \mathit{is\_capital} : \mathit{True},\ \mathit{country} : Y) \rightarrow X) \\
\rho_{country'} &\equiv \text{rew}((\mathit{name} : N, \mathit{language} : L, \mathit{currency} : C) \\
&\quad \rightarrow (\mathit{name} : N,\ \mathit{language} : L,\ \mathit{currency} : C,\ \mathit{capitals} : \rho_{capitals}(Z))) \\
\rho_{city} &\equiv \text{rew}((\mathit{name} : N,\ \mathit{is\_capital} : \_,\ \mathit{country} : Y) \\
&\quad \rightarrow (\mathit{name} : N,\ \mathit{place} : \langle\!|\mathit{euro\_city} : Y|\!\rangle)) \\
\rho &\equiv \text{rew}((\mathit{City} : Z,\ \mathit{Country} : W) \rightarrow (\mathit{City} : \rho_{city}(Z),\ \mathit{Country}' : \rho_{country'}(W)))
\end{aligned}$$

Each rule has the form "rew(*pattern* → *expression*)". When a rewrite rule is applied to a set of values of appropriate type it attempts to bind the pattern to each value in the set. The output of the rule is the set of values of the expression corresponding to succesful bindings of the pattern.

For example the first rewrite rule

$$\text{rew}((\mathit{name} : X,\ \mathit{is\_capital} : \mathit{True},\ \mathit{country} : Y) \rightarrow X)$$

can be applied to a set of records each of which has attributes *name* and *country*, and a Boolean valued attribute *is_capital*. The pattern matches those records for which the *is_capital* attribute is equal to *True*, and the rewrite rule returns the set of values of the name attribute X of those records matched by the pattern.

The rewrite rule ρ takes records with set-valued attributes *City* and *Country* and applies the rewrite rules ρ_{city} and $\rho_{country'}$ to the values associated with these attributes. The rewrite rule $\rho_{country'}$ takes a record with attributes *name*, *language* and *currency*, and forms a new record with an additional attribute *capitals* which is formed by applying the rewrite rule $\rho_{capitals}$ to the free variable X. This kind of nesting of rewrite rules is necessary in [1] to deal with nested sets. In section 13 we will see that such nested rules can be avoided in the language *WOL* by requiring some sort of identity on the elements of any set.

The model of [1] is purely *value-based*: there is no concept of object identity. Consequently it is necessary to use some notion of *keys* in order to represent recursive structures such as those of figure 1, and to reference values in one table or class from values in another (see section 10). Here we are using the *name* attributes of *Cities* and *States* as keys.

Note that the structure formed here does not coincide precisely with that of figure 2 because the class *Country'* has a set valued attribute *capitals* rather than a single valued attribute *capital*. To rectify this we would have to compose the transformation with a second transformation given by the rewrite rule

$$\begin{aligned}\rho' \quad \equiv \quad & \mathrm{rew}((\mathit{name} : N,\ \mathit{language} : L,\ \mathit{currency} : C,\ \mathit{capitals} : \{X\}) \\ & \to (\mathit{name} : N,\ \mathit{language} : L,\ \mathit{currency} : C,\ \mathit{capital} : X))\end{aligned}$$

The pattern of this rewrite rule matches records for which the attribute *capitals* is singleton valued, and maps them to similar records with a singleton attribute *capital* instead.

It is not possible to express this transformation using a single rewrite rule: in [1] rewrite rules are not closed under composition. A subclass of *simple rewrite rules* are also defined, which are closed under composition, but these are not sufficiently expressive to represent our example transformation. ∎

A number of other approaches to schema merging [33, 13, 40] take component schemas, such as those of figures 1 and 3, together with constraints relating the elements of the schemas, for example saying that the *City* classes of the two schemas and the *state* and *country* attributes correspond, and apply an algorithm which returns a unified schema. In these approaches the transformations are generally simple embeddings of data and type coercions.

For most schema integration methodologies the outcome is dependent on the order in which schemas are integrated: that is, they are not associative. Intuitively this should not be the case, since the integration of a set of schemas should depend only on the schemas and the relations between them; the semantics of the integration should be independent of the algorithm used. As a consequence of this non-associativity, a schema integration method will specify an ordering in which schema integrations take place, such as a binary tree or ladder, or all at once, and possibly a way of ordering the particular schemas. For example [8] states that schemas should be ranked and then integrated in order of relevance, although no justification for this ordering is given: why shouldn't it be appropriate to integrate the most relevant schemas last, or in the middle, rather than first? Further enforcing such an ordering is not acceptable in a system in which new databases may be added at a later date: if a database is added to an established federation the result should be the same as if the database had been present in the federation at the outset.

In [13] it was shown that the non-associativity of schema integration methodologies is due to new "implicit" nodes of classes that are introduced during the merging process. The variant of the *State* and *Country* classes in example 3.1 is an example of such an implicit node. By taking account of these implicit nodes and how they are introduced, an independent semantics can be given to the merge of a set of schemas and the relations between them, and an associative schema merging algorithm defined [13].

3.4 Merging Data

Once transformed into a suitable form, data from the component databases must be merged. In a value based model without nested sets or additional constraints, this is simply a matter of taking the union of the relevant data. However, when more complex data models are used, such as those supporting object identity or inter-database constraints, this task becomes more difficult since it necessary to resolve conflicts and equate objects arising from different databases [42, 26].

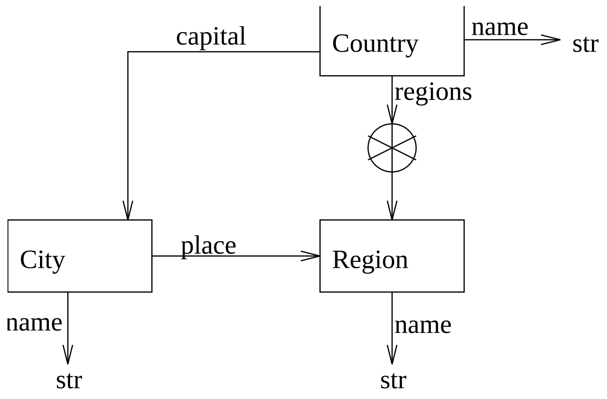


Figure 6: A schema for a international database of Cities and Countries

This problem is not apparent in our running example because the databases of *Cities* and *States* and *Cities* and *Countries* represent disjoint sets of objects. However suppose we were also interested in integrating a third database including international information about Cities and Countries with the schema shown in figure 6. This schema has three classes: *City*, *State* and *Region*. Each *City* is in a *Region*, and each *Country* has a set (indicated by a "star" node) of *Regions*. The exact meaning of *Region* depends on the country to which it belongs. For example, in the United States Regions would correspond to States (or Districts), while in Great Britain Regions might be counties. This database might contain data which overlaps with the other two databases. For example there might be objects representing the city *Philadelphia* in both the International Cities and Countries database and in the Cities and States database, in which case it would be necessary to map both objects to the same object in the integrated database. Equally there might be objects representing the same City or Country in both the International and the European Cities databases, which would need to be combined in the target database.

An important point to note here is that transformations from the various source databases to an integrated database are not independent: it is not sufficient to merely write a transformation from each individual database to the target database. Instead, we may have to write a transformation that takes a *set* of database instances, one for each source schema, and transforms them into a target instance.

The problem of resolving object identity over multiple databases with constraints is examined in [26, 19, 46]. In part II we will give an analysis of the more general problem of how to compare and equate object identities, and conclude by recommending a system of *external keys* for identifying object identities.

4 Data Models for Database Transformations

Works on transformations between heterogeneous databases are usually based around some sufficiently expressive data-model, or *meta*-data-model, which naturally subsumes the models used for the component databases. Various data models have been used, ranging from relational and extended entity-relationship models to semantic and object-oriented models. The main requirements on such a meta-data-model are that the models of component databases being considered should be embeddable in it in a natural way: that is, each constructor or data-type of the component database models should have a natural analogue in the meta-data-model. Further, such a model should be sufficiently simple and expressive as to allow data to be represented in multiple ways so that conflicts between alternative representations of data can be resolved. In [39] the requirements on a model for transforming heterogeneous databases are examined, and the authors conclude that a model supporting complex data-structures (sets, records and variants), object-identity and specialization and generalization relations between object classes is desirable.

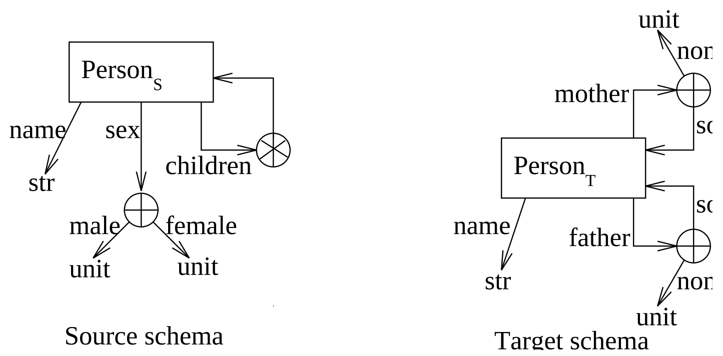


Figure 7: A transformation between recursive data structures

Some notion of referencing, such as *object-identities* or *keys* is essential in order to represent recursive data-structures such as those of figures 1 and 2. However, in order to transform databases involving such recursive structures, it is also necessary to have a notion of *extents* or *classes* in which all objects of a database must occur. To see this, let us look at another example, namely the transformation between the two schemas shown in figure 7. Suppose we considered the first schema merely to define a recursive type $Person_S$. A value of type $Person_S$ would be a record with attributes *name*, *sex* and *children*, such that the *children* attribute would be a set of records of type $Person_S$. In order to transform a source database consisting of a set of values of type $Person_S$, we would have to recursively apply a restructuring transformation to each set of children of each person in the database. This recursion could be arbitrarily deeply nested, and, in the case of cyclic data, non-terminating.

Fortunately the source schema of figure 7 conveys some important information in addition to describing a recursive type: namely it tells us that our database consists of a *finite extent* or *class* $Person_S$, and that all the people represented in the database are reachable as members of this extent. In particular it tells us that, if X is an object in the class Person_S and $Y \in X.\mathit{children}$ (Y is a child of X), then Y is also in the class Person_S. Consequently, when transforming the database, we can iterate our transformation over the elements of the class Person_S, and not have to worry about recursively applying the transformation to the children of a person.

Note that in performing a transformation, it may be necessary to create and reference an object-identity before it has a value associated with it. In this example, if we perform the transformation by iterating over the class Person_S, it may be necessary to create an object in the target class Person_T, with *father* and *mother* attributes both set to *some* person, before the objects corresponding to the parents of the person being transformed have been encountered in the class Person_S. In this case it is necessary to create and reference object identities for the two parents, even though the corresponding values have not yet been formed. Keys provide a mechanism for such early creation and referencing of object identities (see section 10.1).

The data-model we will use in this thesis supports object-identities, classes and complex data-structures. Specialization and generalization relations will be viewed as particular examples of constraints which can be expressed separately using a general constraint language. The model is basically the same as that of [2] and is equivalent to the models implemented in various object-oriented databases [6], except for the omission of direct support for inheritance. Formal details of the model will be defered until sections 7 and 10.

5 Information Dominance in Transformations

One of the important questions of database systems is that of *data-relativism*, or when one schema or data-structure can represent the same data as another. From the perspective of database transformations this can be thought of as asking when there is a transformation from instances of one schema to another such that all the information in the source database is preserved by the transformation. Such a transformation would be said to be *information preserving.*

There are a number of situations when dealing with database transformations where we might want to ensure that a transformation is information preserving. For example when performing a schema evolution, we might want to ensure that none of the information stored in the initial database is lost in the new evolved database, or when integrating databases, we might wish to ensure that all the information stored in one of the component databases is reflected in the integrated database.

Example 5.1: For the schema integration described in example 3.1 the transformation from the database of US Cities and States to the schema of figure 2 is information preserving, in that all the information stored in an instance of the first schema will be reflected in the transformed instance. Equally the transformation from the restructured European Cities and Countries schema of figure 3 to the schema of figure 2 is information preserving.

However the transformation from the first European Cities and Countries schema in figure 1 to the restructured schema of figure 3, and hence to the schema of figure 2, is not information preserving. This is because the transformation to the restructured schema assumes that, for each *Country* in the original schema, there is exactly one *City* of that *Country* with its *is_capital* attribute set to *True*. However the original schema allows a country to have multiple capitals: there may be many *Cities* with their *is_capital* attribute set to *True*. If we were able to associate an additional constraint with European Cities and Countries schema of figure 1 stating that each there can be at most one capital City in each Country, then the transformation would be information preserving, and we could say that the schema of figure 2 *dominates* both of the schemas of figure 1. ■

In section 5.1 we will look at the work of Hull in [23], which defines a series of progressively more restrictive concepts of information dominance, and see how they can be related to transformations using data models such as those described in section 4. In section 5.2 we consider the recent work of Miller in [30, 31] which studies various applications of database transformations, and the need for transformations to be information preserving in these situations.

5.1 Hull's Hierarchy of Information Dominance Measures

In [23] Hull defined four progressively more restrictive notions of information dominance between schemas, each determined by some reversible transformation between the schemas subject to various restrictions. Although [23] dealt only with simply keyed flat-relational schemas, the definitions and some of the results can be easily generalized to other data-models supporting the concepts of schemas and instances of schemas.

If $\mathcal{S}$ is a schema in some data-model, we will write $\mathit{Inst}(\mathcal{S})$ for the set of *instances* of $\mathcal{S}$.

Given two schemas, $\mathcal{S}$ and $\mathcal{T}$, a transformation from $\mathcal{S}$ to $\mathcal{T}$ is a partial map $\sigma : \mathit{Inst}(\mathcal{S}) \rightarrow \mathit{Inst}(\mathcal{T})$. Intuitively the transformation is information preserving iff there is a second transformation from $\mathcal{T}$ back to $\mathcal{S}$, say $\rho : \mathit{Inst}(\mathcal{T}) \rightarrow \mathit{Inst}(\mathcal{S})$ such that ρ recovers the instances of $\mathcal{S}$. That is, for any $\mathcal{I} \in \mathit{Inst}(\mathcal{S})$, the instances $\mathcal{I}$ and $(\rho \circ \sigma)(\mathcal{I})$ are equivalent[3]. In such a situation we say that $\mathcal{T}$ **dominates** $\mathcal{S}$ **via** (σ, ρ).

We say that a schema $\mathcal{T}$ **dominates** $\mathcal{S}$ **absolutely**, $\mathcal{S} \preceq \mathcal{T}(\mathit{abs})$, iff there exist transformations σ and ρ such that $\mathcal{T}$ dominates $\mathcal{S}$ via (σ, ρ).

However a problem with this definition is that there may be many functions from $\mathit{Inst}(\mathcal{S})$ to $\mathit{Inst}(\mathcal{T})$ which do not correspond to any reasonable or definable transformations, so that, although there may be an information preserving map from instances of $\mathcal{S}$ to those of $\mathcal{T}$, there is no way of realizing this map.

Suppose that $\mathcal{B}$ is the set of base types supported by a data-model (*integers, strings, reals* and so on), and for each base type, $\underline{b} \in \mathcal{B}$, $\mathbf{D}^{\underline{b}}$ is the set of values of type $\underline{b}$ that may occur in an

[3]Note that, in order for this to make sense, we need a method of comparing the instances of a data-model, and deciding when they are equivalent. The issues of equivalence of instances will be explored at some length in part II. For now it will suffice to assume that any data-model has a decidable notion of equivalence associated with it, such that equivalent instances are indistinguishable

instance of the model.

For any instance $\mathcal{I}$ the **support** of $\mathcal{I}$, $Supp(\mathcal{I})$ is the set of values from $\bigcup_{\underline{b}\in\mathcal{B}}\mathbf{D}^{\underline{b}}$ that occur in $\mathcal{I}$.

Suppose $Z \subseteq \bigcup_{\underline{b}\in\mathcal{B}}\mathbf{D}^{\underline{b}}$ is a finite set. A transformation σ from $\mathcal{S}$ to $\mathcal{T}$ is said to be **Z-internal** iff, for every instance $\mathcal{I} \in Inst(\mathcal{S})$, $Supp(\sigma(\mathcal{I})) \subseteq Supp(\mathcal{I}) \cup Z$. Intuitively a transformation is Z-internal if it doesn't invent any new values, beyond some finite set of constants represented by Z.

A schema $\mathcal{T}$ dominates a schema $\mathcal{S}$ **internally**, $\mathcal{S} \preceq \mathcal{T}(int)$ iff there is a finite set $Z \subseteq \bigcup_{\underline{b}\in\mathcal{B}}\mathbf{D}^{\underline{b}}$ and Z-internal transformations σ and ρ such that $\mathcal{T}$ dominates $\mathcal{S}$ via (σ,ρ).

The next concept of dominance attempts to capture the idea that base values are "essentially uninterpreted". Suppose $Z \subseteq \bigcup_{\underline{b}\in\mathcal{B}} D^{\underline{b}}$ is a finite set. A **Z-permutation** $g^{\mathcal{B}}$ is a family of *bijections* $g^{\underline{b}} : \mathbf{D}^{\underline{b}} \to \mathbf{D}^{\underline{b}}$ such that $g^{\underline{b}}$ restricted to $\mathbf{D}^{\underline{b}} \cap Z$ is the identity function for each $\underline{b} \in \mathcal{B}$. Given a Z-permutation $g^{\mathcal{B}}$ and an instance $\mathcal{I}$, we can form the instance $g^{\mathcal{B}}(\mathcal{I})$ by replacing every base value v of type $\underline{b}$ occuring in $\mathcal{I}$ with $g^{\underline{b}}(v)$.

A transformation σ from $\mathcal{S}$ to $\mathcal{T}$ is said to be **Z-generic** iff for any Z-permutation $g^{\mathcal{B}}$ and any instance $\mathcal{I} \in Inst(\mathcal{S})$, $\sigma(g^{\mathcal{B}}(\mathcal{I})) \cong g^{\mathcal{B}}(\sigma(\mathcal{I}))$. Intuitively a transformation is Z-generic if all base values other than those in the finite set Z are "essentially uninterpreted values". This fits with the assumption common in database query and constraint languages that no computations are performed on values themselves beyond simple comparisons.

The following lemma follows simply:

Lemma 5.1: If σ is a transformation from $\mathcal{S}$ to $\mathcal{T}$ and σ is Z-generic for some finite set Z, then there is a finite set Z' such that $Z \subseteq Z'$ and σ is Z'-internal. ■

Proof: Suppose that $\mathcal{S}$, $\mathcal{T}$, Z and σ are such that σ is a Z-generic transformation from $\mathcal{S}$ to $\mathcal{T}$, but there is no Z' as described in the lemma. Then there exists a base type $\underline{b} \in \mathcal{B}$ and an instance $\mathcal{I}$ of $\mathcal{S}$ such that the cardinality of $\mathbf{D}^{\underline{b}}$ is not finite and there exists a $v \in \mathbf{D}^{\underline{b}}$ such that $v \in Supp(\sigma(\mathcal{I}))$ and $v \notin Supp(\mathcal{I}) \cup Z$, since otherwise we could take $Z' = Z \cup \bigcup\{\mathbf{D}^{\underline{b}} | \mathbf{D}^{\underline{b}} \text{ finite}, \underline{b} \in \mathcal{B}\}$. Choose $v' \in \mathbf{D}^{\underline{b}} \setminus (Z \cup Supp(\mathcal{I}))$ such that $v' \neq v$. Let $g^{\mathcal{B}}$ be such that $g^{\underline{b}}(v) = v'$, $g^{\underline{b}}(v') = v$, $g^{\underline{b}}$ is the identity elsewhere on $\mathbf{D}^{\underline{b}}$, and $g^{\underline{b}'}$ is the identity on $D^{\underline{b}'}$ for $\underline{b}' \in \mathcal{B}$, $\underline{b}' \neq \underline{b}$. Clearly $g^{\mathcal{B}}$ is a Z-permutation, and $\mathcal{I} = g^{\mathcal{B}}(\mathcal{I})$. But $\sigma(\mathcal{I}) \neq g^{\mathcal{B}}(\sigma(\mathcal{I}))$, contradicting our initial assumption. ■

The converse, however, does not hold. Suppose, for example, we had a schema with a class *Person* which had an attribute *age*. Consider a transformation from the schema to itself which is the identity transformation except that it replaces the *age* of each *Person* in an instance with the *minimum* value of the *age* attribute occuring in the instance: if there was an object of class *Person* with *age* 4 say, and no objects in *Person* with *age* less than 4, then every *Person* in the transformed instance would have *age* 4. Such a transformation would be internal, since no new values are introduced, but is not Z-generic for any Z.

A schema $\mathcal{T}$ dominates $\mathcal{S}$ **generically**, $\mathcal{S} \preceq \mathcal{T}(gen)$, iff there is a finite set Z and Z-generic transformations σ and ρ such that $\mathcal{T}$ dominates $\mathcal{S}$ via (σ,ρ).

The final concept of information dominance captures the idea of having transformations expressible in some implicit calculus. To formalize this definition and realize the following results it is necessary to actually fix some underlying calculus for expressing transformations, and to show that for any transformation expressed in the calculus there will be a finite set Z such that the transformation is Z-generic. Later we will be using the language *WOL*, which does satisfy these properties, for expression transformations. For the time being it will suffice to assume some implicit calculus.

Suppose $\mathcal{T}$ and $\mathcal{S}$ are schemas. Then $\mathcal{T}$ **dominates** $\mathcal{S}$ **calculously**, $\mathcal{S} \preceq \mathcal{T}(calc)$, iff there are calculus expressions representing transformations σ and ρ such that $\mathcal{T}$ dominates $\mathcal{S}$ via (σ, ρ).

It is clear that calculus dominance is more restrictive than the other three concepts of dominance. However it has the disadvantage of depending on a particular calculus for its definition, while the other definitions of dominance are more abstract.

The following proposition is due to Hull ([23]) and follows easily from the previous lemma:

Proposition 5.2:Hull '86. Let $\mathcal{S}$ and $\mathcal{T}$ be schemas. Then $\mathcal{S} \preceq \mathcal{T}(calc)$ implies $\mathcal{S} \preceq \mathcal{T}(gen)$, $\mathcal{S} \preceq \mathcal{T}(gen)$ implies $\mathcal{S} \preceq \mathcal{T}(int)$ and $\mathcal{S} \preceq \mathcal{T}(int)$ implies $\mathcal{S} \preceq \mathcal{T}(abs)$. ■

A more significant result, also shown in [23] is the following:

Proposition 5.3:Hull '86 Let $\mathcal{S}$ and $\mathcal{T}$ be schemas. Then $\mathcal{S} \preceq \mathcal{T}(int)$ does not imply $\mathcal{S} \preceq \mathcal{T}(gen)$. ■

In particular, in [23], it was shown that there are flat relational schemas $\mathcal{S}$ and $\mathcal{T}$ for which $\mathcal{S} \preceq \mathcal{T}(int)$ and $\mathcal{S} \not\preceq \mathcal{T}(gen)$. It follows that we cannot hope to construct a calculus which is complete with respect to expressing internal dominance. The questions of whether absolute dominance implies internal dominance, or generic dominance implies calculus dominance for some calculus are left open however.

An important conclusion of [23] is that none of these criteria capture an adequate notion of semantic dominance, that is, whether there is a semantically meaningful interpretation of instances of one schema as instances of another. Consequently the various concepts of information dominance can be used in order to test whether semantic dominance between schemas is plausible, or to verify that a proposed transformation is information preserving, but the task of finding a semantically meaningful transformation still requires a knowledge and understanding of the databases involved.

Another significant problem with this analysis is that it assumes that all possible instances of a source schema should be reflected by distinct corresponding instances of a target schema. However, in practice only a small number of instances of a source schema may actually correspond to real world data sets. That is, there may be implicit constraints on the source database which are not included in the source schema, either because they are not expressible in the data-model being used or simply because they were forgotten or not anticipated at the time of initial schema design. An alternative approach, pursued in [18], is to attempt to define information preserving transformations and valid schemas with respect to some underlying *"universe of discourse"*. However such characterizations are impossible or impractical to represent and verify in practice.

5.2 Information Capacity and Constraints

In [30, 31] Miller et al. analyse the information requirements that need to be imposed on transformations in various applications. The restrictions on transformations that they consider are somewhat simpler than those of [23] in that they examine only whether transformations are injective (one-to-one) or surjective (onto) mappings on the underlying sets of instances. For example they claim that if a transformation is to be used to view and query an entire source database then it must be a total injective function, while if a database is to be updated via a view then the transformation to the view must also be surjective. Having derived necessary conditions for various applications of transformations, they then go on to evaluate existing work on database integration and translations in the light of these conditions.

An important observation in [30] is that database transformations can fail to be information capacity preserving, not because there is anything wrong with the definition of the transformations themselves, but because certain constraints which hold on the source database are not expressed in the source database schema. However the full significance of this observation is not properly appreciated: in fact it is frequently the case that the constraints that must be taken into account in order to validate a transformation have not merely been omited from the source schema, but are not expressible in any standard constraint language.

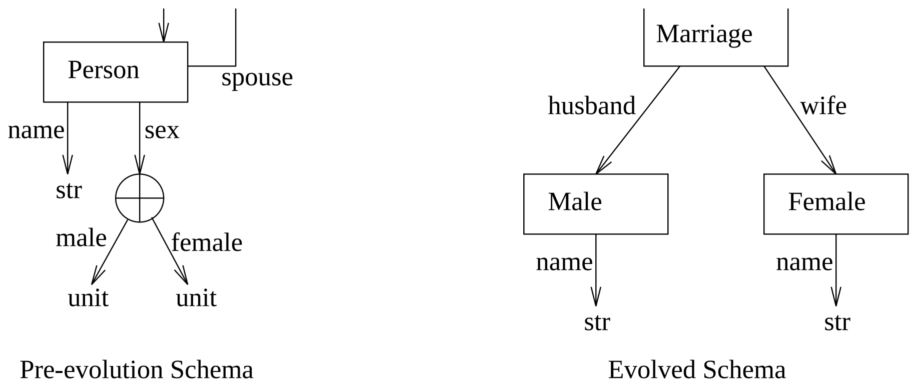


Figure 8: An example schema evolution

Example 5.2: Consider the schema evolution illustrated in figure 8. The first schema has only one class, *Person*, with attributes representing a person's *name*, *sex* (a variant of *male* and *female*) and *spouse*. In our second (evolved) schema the *Person* class has been split into two distinct classes, *Male* and *Female*, perhaps because we wished to start storing some different information for men and women. Further the *spouse* attribute is replaced by a new class, *Marriage*, perhaps because we wished to start recording additional information such as dates of marriages, or allow un-married people to be represented in the database.

It seems clear that there is a meaningful transformation from instances of the first database to instances of the second. The transformation can be described by the following transformation program, written in the language *WOL*:

$$
\begin{array}{l}
X \in \mathit{Male},\ X.\mathit{name} = N \ \Longleftarrow\ Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{male}}();\\
X \in \mathit{Female},\ X.\mathit{name} = N \ \Longleftarrow\ Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}();\\
M \in \mathit{Marriage},\ M.\mathit{husband} = X,\ M.\mathit{wife} = Y\\
\qquad\qquad \Longleftarrow\ X \in \mathit{Male},\ Y \in \mathit{Female}, Z \in \mathit{Person},\ W \in \mathit{Person},\\
\qquad\qquad\quad\ X.\mathit{name} = Z.\mathit{name},\ Y.\mathit{name} = W.\mathit{name},\ W = Z.\mathit{spouse};
\end{array}
$$

This program consists of three *clauses* which are logical statements describing the transformation. The first clause states that for each object Y in the source *Person* class, such that the *name* attribute of Y is equal to the string N ("Y.*name*= N"), and the *sex* attribute of Y is the male choice of the variant ("$Y.\mathit{sex} = \mathit{ins}_{\mathit{male}}()$"), there is a corresponding object X in the target *Male* class with *name*attribute equal to N. Similarly the second clause states that for every Y in the source class *Person* with *name* attribute N and *sex* attribute equal to the *female*choice of the variant, there is a corresponding object X in the target *Female* class. The third clause describes how a object M in the target *Marriage* class arises between to objects X and Y in the target *Male* and *Female* classes. Details of the language *WOL* will be given in section 13.

Although this transformation intuitively appears to preserve the information of the first database, in practice it is not information preserving. The reason is that there are instances of the *spouse* attribute that are allowed by the first schema that will not be reflected by the second schema. In particular the first schema does not require that the *spouse* attribute of a man goes to a woman, or that for each *spouse* attribute in one direction there is a corresponding *spouse* attribute going the other way. To assert these things we would need to augment the first schema with additional constraints, such as:

$$
\begin{array}{l}
X.\mathit{sex} = \mathit{ins}_{\mathit{male}}() \ \Longleftarrow\ Y \in \mathit{Person},\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}(),\ X = Y.\mathit{spouse};\\
Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}() \ \Longleftarrow\ X \in \mathit{Person},\ X.\mathit{sex} = \mathit{ins}_{\mathit{male}}(),\ Y = X.\mathit{spouse};\\
Y = X.\mathit{spouse} \ \Longleftarrow\ Y \in \mathit{Person},\ X = Y.\mathit{spouse};
\end{array}
$$

These are also clauses of the language *WOL*, but this time are interpreted as *constraints* on the source database. The first clause states that for any Y in the class *Person* such that the *sex* attribute of Y is the *female* choice of the variant, the *spouse* attribute of Y points to an object for which the *sex* attribute is the *male* choice of the variant: that is, the spouse of a female person is always a male person. Similarly the second clause states that the spouse of a male is always a female. The third clause states that, for any object Y of class *Person*, if X is the *spouse* of Y then Y is the *spouse* of X.

We can then show that the transformation is information preserving on those instances of the first schema that satisfy these constraints. Notice however, that these constraints are very general, and deal with values at the instance level of the database, rather than just being expressible at the schema level. They could not be expressed with the standard constraint languages associated with most data-models (functional dependencies, inclusion dependencies, cardinality constraints and so on). ■

This highlights one of the basic problems with information capacity analysis of transformations: Such an analysis assumes that schemas give a complete description of the set of possible instances of a database. In practice schemas are seldom complete, either because certain constraints were forgotten or were not known at the time of schema design, or because the data-model being used simply isn't sufficiently expressive. When dealing with schema evolutions, where information capacity preserving transformations are normally required, it is frequently the case that the transformation implementing a schema evolution appears to discard information, while in fact this is because the new schema is a better fit for the data, expressing and taking advantage of various constraints that have become apparent since the initial schema design.

Further, when dealing with transformations involving multiple source databases, even if the transformations from individual source databases to a target database are information preserving, it is unlikely that the transformations will be jointly information preserving. This is in part due to the fact that the source databases may represent overlapping information, and inter-database constraints are necessary to ensure that the individual databases do not contain conflicting information. It may also be due to the fact that information describing the source of a particular item of data may be lost.

An additional limitation of the information capacity analysis of transformations is that it is very much an all-or-nothing property, and does not help us to establish other less restrictive correctness criteria on transformations. When dealing with database integration, we might only be interested in a small part of the information stored in one of the source databases, but wish to ensure that the information in this subpart of the database is preserved by the transformation. For example, we might be integrating our database of US Cities and States with a database of European Cities or towns and Countries, and only be interested in those Cities or towns with a population greater than a hundred thousand. However we would still like to ensure that our transformation does not lose any information about European Cities and towns with population greater than one hundred thousand.

It therefore seems that a more general and problem specific correctness criteria for transformations is needed, such as relative information capacity. In addition, a formalism in which transformations and constraints can be jointly be expressed is needed in which to test these more general correctness criteria. The language *WOL* provides a uniform framework for specifying database transformations and constraints, and so provides a first step towards these goals.

Part II

Observable Properties of Models for Recursive Data-Structures

6 Introduction

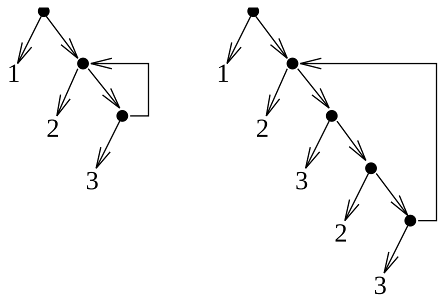
The purpose of a database transformation is to transform the information stored in one or more source databases into an instance of a target database. In particular, the results of a transformation should not be affected by implementation details such as the locations of source databases or details of disk or memory allocation: if a transformation is well-defined then when it is applied to two source instances representing the same information then the resulting target instances should be the same, or at least should contain the same information. In order to ensure that transformations are well-defined it is therefore necessary to have a precise understanding of the information capacity of the data-models involved, and to be able to test whether two instances represent the same information.

The information represented by a database instance is precisely that which can be observed using the available query mechanisms: if some query distinguishes between two instances then they represent different information, while if it is not possible to distinguish between them using any queries then they represent the same information. Equally, two values or objects occuring in a database represent the same data iff it is not possible to distinguish between them using any available query mechanism. It follows that the semantics and expressive power of any data-model is dependent on the assumptions about what operations are available for examining the data.

Given two instances of a database schema, suppose we wished to determine whether or not the instances were different. 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, one could find a fixed set of queries, dependent only on the schema, 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 [6, 25], 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.

Suppose, for example, we are using a data-model which supports object identities and are comparing the two instances shown bellow: object identities are represented by •, and each identity has a value associated with it consisting of an integer and another object identity.



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 correspond to an infinite sequence $1, 2, 3, 2, 3, 2, 3, \ldots$, lthough their representations are clearly different.

Though this example is simple, it 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 or some other kind of reference mechanism 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.* 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, as well as for reasoning about the well-definedness of operations such as database transformations. Recent independent work by Abiteboul and Van den Bussche[4] overlaps the results presented in this Part, albeit a with an emphasis on values rather than entire instances.

In section 7 we will present the object-identity based model that we will use throughout the remainder of this thesis. In section 8 we will present a variant of the query language *SRI* [10] adapted for our data-model and equiped with an equality test on object identities. We will show that, in this query language, instances are observationally indistinguishable iff they are isomorphic. In section 9 we will consider another variant of this query language, this time with no means of comparing object identities, and show that in this case instances are indistinguishable iff they are bisimilar. In section 10 we will extend our model with systems of *keys*, and will show that such systems of keys give rise to observational equivalence relations which lie inbetween the two extremes of bisimulation and isomorphism of instances. In section 11 we will present an alternative model based on *regular trees*, and show that instances of this model are equivalent to bisimulation classes of instances in the regular tree model. Section 11 is, in a sense, orthogonal to the other sections of this Part, and need not be included in a reading. However it examines ideas that have been proposed in a variety of places, most notably in [2], but have not been thoroughly examined elsewhere, and shows that, despite their intuitive nature, there are considerable problems in providing a formal treatment of data-models based on un-ordered regular trees.

7 A Data-Model with Object Identities and Extents

As has already been pointed out, in order to allow for the representation of recursive or arbitarily deeply nested data-structures, a data-model must support support some kind of reference mechanism. Various such mechanisms have been proposed, including the use of pointers, surrogate keys or object identities ([27]). The choice between these different approaches is largely a stylistic one. We will focus our attention on models based on object identities, since they offer advantages of locational and data independence, and afford efficient implementation techniques [27], and because the author finds the use of object identities to be the most intuitive of these choices. However the results of these sections should apply equally well to other reference mechanisms with minor adaptions.

An important distinction between databases and other fields of computation is that all the values that may occur in a database are accessable through some top level finite extents. For example, in a relational database all the data in an instance is contained in one of a fixed set of relations, while in an object-oriented database every object belongs to some *class* [6]. In section 4 it was observed that it is the presense of these finite extents that enables us to perform transformations on recursive data-structures. In the following sections we will see that these finite extents also allow us to compute comparisons on recursive data-structures that we would not expect be decidable in a more general model of computation.

In this section we will describe a data-model supporting complex data-structures and object identities, which we will use to investigate the problems of observational indistinguishability raised in section 6. The same model will be used in part III as the basis of the transformation and constraint language *WOL*. The model is basically equivalent to that of [2], and could also be considered to be a simplification of the models of [5, 25].

The model starts with a *type system* which is similar to the types of the nested relational model [3] with the addition of *class types* which are used to represent the finite extents of object identities present in a database. In order to describe a particular database system, it it necessary to state what classes are present and also the types of (the values associated with) the objects of each class.

7.1 Types and Schemas

Definition 7.1: Assume a fixed countable set of *attribute labels*, $\mathcal{A}$, ranged over by $a, a', \ldots$, and a fixed finite set of *base types*, $\mathcal{B}$, ranged over by $\underline{b}, \ldots$.

Let $\mathcal{C}$ be some finite set of *classes*, ranged over by $C, C', \ldots$. Then the set $\mathit{Types}^{\mathcal{C}}$ of **types** over $\mathcal{C}$ is then given by the following abstract syntax:

$$\begin{array}{rcll} \tau & ::= & \{\tau\} & \text{— set type} \\ & | & (a : \tau, \ldots, a : \tau) & \text{— record type} \\ & | & \langle\!| a : \tau, \ldots, a : \tau |\!\rangle & \text{— variant type} \\ & | & \underline{b} & \text{— base type} \\ & | & C & \text{— class type} \end{array}$$

The notation $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ represents a *record type* with attributes $a_1, \ldots, a_k$ of types $\tau_1, \ldots, \tau_k$ respectively, and similarly $\langle\!| a_1, \tau_1, \ldots, a_k : \tau_k |\!\rangle$ represents a *variant type* with attributes $a_1, \ldots, a_k$.

A **schema** consists of a finite set of classes, $\mathcal{C}$, and a mapping $\mathcal{S} : \mathcal{C} \to \mathit{Types}^{\mathcal{C}}$, such that

$$\mathcal{S} : C \mapsto \tau^C$$

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). ∎

Example 7.1: As an example let us consider the database of Cities and States shown in figure 1. Our set of classes is

$$\mathcal{C}_A \equiv \{\mathit{City}_A, \mathit{State}_A\}$$

and the schema mapping, $\mathcal{S}_A$, is given by

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

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).

For the combined schema of figure 2 the classes would be

$$\mathcal{C}_T \equiv \{\mathit{City}_T, \mathit{State}_T, \mathit{Country}_T\}$$

and the schema mapping $\mathcal{S}_T$ would be

$$\begin{array}{rcl} \mathcal{S}_T(\mathit{City}_T) & \equiv & (\mathit{name} : \mathit{str}, \mathit{place} : \langle\!| \mathit{us\_city} : \mathit{State}_T, \mathit{euro\_city} : \mathit{Country}_T |\!\rangle) \\ \mathcal{S}_T(\mathit{State}_T) & \equiv & (\mathit{name} : \mathit{str}, \mathit{capital} : \mathit{City}_T) \\ \mathcal{S}_T(\mathit{Country}_T) & \equiv & (\mathit{name} : \mathit{str}, \mathit{language} : \mathit{str}, \mathit{currency} : \mathit{str}, \mathit{capital} : \mathit{City}_T) \end{array}$$

■

We can view schemas as directed graphs, with classes and other type constructors as their nodes. Note that, if we take this view, any loops in the graph must go through a class node. This means that any recursion in a schema must be via a class. Consequently, we will see in section 7.2, any recursive data-structures in an instance must have a finite representation via the object-identifiers of these classes.

7.2 Database Instances

The instances of our data-model will be based on *object identities*. This could be thought of as providing an abstract model of the internal representation of a database instance, rather than a representation of the observable properties of an instance.

For each base type $\underline{b}$, assume a fixed domain $\mathbf{D}^{\underline{b}}$ associated with $\underline{b}$.

The values that may occur in a particular database instance depend on the object identities in that instance. Consequently we will 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.[4] We define the *domain* of our model for the sets of object identities $\sigma^{\mathcal{C}}$, $\mathbf{D}(\sigma^{\mathcal{C}})$, to be the smallest set satisfying

$$\begin{aligned}\mathbf{D}(\sigma^{\mathcal{C}}) \quad &\equiv \quad \left(\bigcup_{C\in\mathcal{C}} \sigma^C\right) \cup \left(\bigcup_{\underline{b}\text{ a base type}} \mathbf{D}^{\underline{b}}\right) \cup \\ &\qquad (\mathcal{A} \overset{\sim}{\to} \mathbf{D}(\sigma^{\mathcal{C}})) \cup (\mathcal{A} \times \mathbf{D}(\sigma^{\mathcal{C}})) \cup \mathcal{P}_{fin}(\mathbf{D}(\sigma^{\mathcal{C}}))\end{aligned}$$

where $X \overset{\sim}{\to} Y$ represents the set of partial functions from X to Y with finite domains.

Definition 7.2: For each type τ define $[\![\tau]\!]\sigma^{\mathcal{C}}$ by

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

■

[4]Alternatively *oids* of class C, or *references*, or *pointers* or *place-holders* or *gumballs*, or whatever else you like best.

Definition 7.3: 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

$$\mathcal{V}^C : \sigma^C \to [\![\tau^C]\!]\sigma^{\mathcal{C}}$$

We write $\mathit{Inst}(\mathcal{S})$ for the set of instances of a schema $\mathcal{S}$. ■

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 $[\![\tau]\!]\sigma^{\mathcal{C}}$.

Example 7.2: We will describe an instance of the schema introduced in example 7.1.

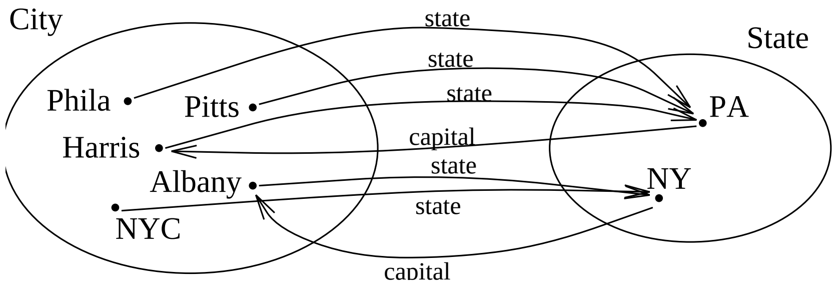


Figure 9: A database instance

Our object identities are:

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

and the mappings are

$$\begin{aligned} \mathcal{V}^{City}(\mathit{Phila}) &\equiv (\mathit{name} \mapsto \text{“Philadelphia”}, \mathit{state} \mapsto \mathit{PA}) \\ \mathcal{V}^{City}(\mathit{Pitts}) &\equiv (\mathit{name} \mapsto \text{“Pittsburg”}, \mathit{state} \mapsto \mathit{PA}) \\ \mathcal{V}^{City}(\mathit{Harris}) &\equiv (\mathit{name} \mapsto \text{“Harrisburg”}, \mathit{state} \mapsto \mathit{PA}) \\ \mathcal{V}^{City}(\mathit{NYC}) &\equiv (\mathit{name} \mapsto \text{“New York City”}, \mathit{state} \mapsto \mathit{NY}) \\ \mathcal{V}^{City}(\mathit{Albany}) &\equiv (\mathit{name} \mapsto \text{“Albany”}, \mathit{state} \mapsto \mathit{NY}) \end{aligned}$$

and

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

This defines the instance illustrated in figure 9. ■

Homomorphisms and Isomorphisms 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 or transformation when applied to two isomorphic instances should return isomorphic results. Isomorphism therefore provides the finest level of distinction that we might hope to be able to observe.

In this section we will formalize this notion.

If $\mathcal{I}$ and $\mathcal{I}'$ are two instances of a schema $\mathcal{S}$, and $f^{\mathcal{C}}$ is a family of mappings, $f^C : \sigma^C \to \sigma'^C$, $C \in \mathcal{C}$, then we can extend $f^{\mathcal{C}}$ to mappings $f^\tau : [\![\tau]\!]\mathcal{I} \to [\![\tau]\!]\mathcal{I}'$ 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 **homomorphism** 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 mappings, $f^{\mathcal{C}}$, such that for each $C \in \mathcal{C}$ and each $o \in \sigma^C$

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

Definition 7.4: An **isomorphism** of two instances, $\mathcal{I}$ and $\mathcal{I}'$, consists of a homomorphism, $f^{\mathcal{C}}$ from $\mathcal{I}$ to $\mathcal{I}'$ and a homomorphism $g^{\mathcal{C}}$ from $\mathcal{I}'$ to $\mathcal{I}$, such that, for $C \in \mathcal{C}$, $g^C \circ f^C$ is the identity mapping on σ^C and $f^C \circ g^C$ is the identity mapping on σ'^C. $\mathcal{I}$ and $\mathcal{I}'$, are said to be **isomorphic** iff there exists an isomorphism $f^{\mathcal{C}}$ between $\mathcal{I}$ and $\mathcal{I}'$.

We write $\mathcal{I} \cong \mathcal{I}'$ to mean $\mathcal{I}$ is isomorphic to $\mathcal{I}'$. ∎

In section 8 we will show that, given a query language equipped with an equality test on object identities, isomorphism of instances coincides exactly with observational indistinguishability.

8 A Query Language Based on Structural Recursion

In this section we will present an adaption of the query language *SRI* ([10, 11]) to the model of definition 7.3. The language is based on the mechanism of *structural recursion* over sets which was described in [10] as a basis for a query language on the nested relational data-model. Our choice of this mechanism is because it is semantically well understood and because it is known to be more expressive than other formally developed query languages for nested relational model, such as the algebra and calculus of [3]. Consequently most of the results on the expresivity of various operators in this language paradigm will automatically carry over to other query language paradigms.

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

In section 8.1 we introduce the language *SRI*(=) and describe *queries* to be closed expressions of ground type in this language. A denotational semantics for *SRI*(=) is given, and certain useful shorthand notations or extensions are introduced. In section 8.2 we show that two instances are *indistinguishable* in *SRI*(=) if and only if they are isomorphic, but that there is no *generic* test for isomorphism of instances: that is, there is no *SRI*(=) query which, when evaluated for any two instances, will return the same result if and only if the instances are isomorphic.

In section 9.2 we will present the language *SRI* with no comparison operator on object identities. We will show that distinguishability of instances in this language coincides with the bisimulation correspondence on instances defined in 9.2, and that it is possible to test values in an instance for bisimilarity using only *SRI*. However we also show that testing for bisimilarity using *SRI* requires the ability to recurse over the entire extents of a database instance, from which we conclude that a more efficient means of comparing values is necessary.

In section 10 we will consider a third variant of the language *SRI*, this time taking account of a key-specification on the schema. We show that distinguishability in such a language lies between isomorphism and bisimulation of instances, and that such a language, together with an acyclic key-specification, provides us with an efficient way of comparing databases and values in a database.

8.1 Queries and the Language *SRI*(=)

The query language is described for a schema $\mathcal{S}$, with classes $\mathcal{C}$, such that $\mathcal{S} : C \mapsto \tau^C$ for each $C \in \mathcal{C}$. The schema will be considered to be implicit in the remainder of this section, and indeed in most of this document.

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 to a finite set of base types, $\mathcal{B}$, ranged over by $\underline{b}, \ldots$, 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). We expand our type system to include first-order function types (ranged over by $T, T', \ldots$), as well as object/data types (ranged over by $\tau, \ldots$):

Object types

$$\begin{array}{rcl} \tau & ::= & (a : \tau, \ldots, a : \tau) \mid \langle\!| a : \tau, \ldots, a : \tau |\!\rangle \mid \{\tau\} \mid \\ & & unit \mid Bool \mid \underline{b} \mid C \end{array}$$

General types

$$T \quad ::= \quad \tau \mid \tau \to T$$

Definition 8.1: 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.

For example the type $(\mathit{name} : \mathit{str},\ \mathit{state\_name} : \mathit{str})$ is a ground type, while $(\mathit{name} : \mathit{str},\ \mathit{state} : \mathit{State})$ is not. When comparing two instances of the schema of example 7.1, it makes no sense to compare values of the second type since they will contain object identities taken from two distinct instances.

For each class $C \in \mathcal{C}$ we will assume there is a binary predicate $=^C$ in the language which tests whether two terms evaluate to the same object identity. Also for each non-finite base type, $\underline{b}$ we will assume a binary predicate $=^{\underline{b}}$. Consequently, we will see later this section that we can define equality predicates $=^\tau$ on each object type τ, and also set inclusion predicates, using the language.

For each type τ we assume a countably infinite set of variables $x^\tau, y^\tau, \ldots$, and for each base type $\underline{b} \in \mathcal{B}$ and base value $c \in \mathbf{D}^{\underline{b}}$ we assume a corresponding *constant symbol* $\bar{c}$. The syntax and typing rules are given in figure 10.

Products

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

Variants

$$\frac{\vdash e : \tau_i}{\vdash \mathit{ins}_{a_i}^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle} e : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle}$$

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

Sets

$$\frac{}{\vdash \emptyset^\tau : \{\tau\}} \qquad \frac{\vdash e_1 : \tau \ \ \vdash e_2 : \{\tau\}}{\vdash \mathit{add}(e_1, e_2) : \{\tau\}} \qquad \frac{\vdash e_1 : \tau_1 \to \tau_2 \to \tau_2 \ \ \vdash e_2 : \tau_2 \ \ \vdash e_3 : \{t_1\}}{\vdash \mathit{sri}(e_1, e_2, e_3) : \tau_2}$$

Functions

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

Booleans

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

Base values

$$\frac{c \in \mathbf{D}^{\underline{b}}}{\vdash \bar{c} : \underline{b}} \qquad \frac{\vdash e : \underline{b} \ \ \vdash e' : \underline{b}}{\vdash e =^{\underline{b}} e' : \mathit{Bool}}$$

Others

$$\frac{}{\vdash x^\tau : \tau} \qquad \frac{}{\vdash () : \mathit{unit}} \qquad \frac{}{\vdash C : \{C\}} \qquad \frac{\vdash e : C}{\vdash !e : \tau^C} \qquad \frac{\vdash e_1 : C \ \ \vdash e_2 : C}{\vdash e_1 =^C e_2 : \mathit{Bool}}$$

Figure 10: Typing rules for query language

The operator *sri* is the only part of this language to really require an explanation. *sri* takes three arguments: a function, a starting value and a set. It then *iterates* the function over the set starting with the starting value. So, for example, the expression $\mathit{sri}(f, p, S)$ would be equivalent to $f(s_1, f(s_2, \ldots, f(s_k, p) \ldots))$, if S denoted the set $\{s_1, \ldots, s_k\}$ (allowing for rather a lot of notational abuse).

Definition 8.2: An $\mathit{SRI}(=)$ expression e is said to be *closed* iff there are no free variables occuring in e. A **query** is a closed expression of ground type. ■

A query represents a question one can ask of a database: since they are closed expressions they can be evaluated without recourse to any sort of an environment, and since they are of ground type the results my be observed directly, and the results of evaluating the same query in different database instances can be compared.

Example 8.1: For the schema of example 7.1, the following query will return the set of all names of *States* in an instance:

$$\mathit{sri}(\lambda x \cdot \lambda y \cdot \mathit{add}(x.\mathit{name}, y), \emptyset, \mathit{State})$$

where we use $e.a$ as shorthand for $\pi_a(!e)$.

The following expression returns the set of all *Cities* in states named "Pennsylvania":

$$\mathit{sri}(\lambda x \cdot \lambda y \cdot \mathit{if}(x.\mathit{state}.\mathit{name} = \text{“Pennsylvania”}, \mathit{add}(x, y), y), \emptyset, \mathit{City})$$

However this expression does not count as a query, since its type is $\{\mathit{City}\}$. A query which returns the names of all Cities that have a state named "Pennsylvania" would be

$$\mathit{sri}(\lambda x \cdot \lambda y \cdot \mathit{if}(x.\mathit{state}.\mathit{name} = \text{“Pennsylvania”}, \mathit{add}(x.\mathit{name}, y), y), \emptyset, \mathit{City})$$

■

Semantics of *SRI*(=)

Let *Var* be the set of variables of $\mathit{SRI}(=)$. An *environment* for instance $\mathcal{I}$ is a mapping $\rho : \mathit{Var} \overset{\sim}{\rightarrow} \mathbf{D}(\mathcal{I})$ such that $\rho(x^\tau) \in [\![\tau]\!]\mathcal{I}$ for each variable x^τ of type τ.

If ρ is an environment, x^τ a variable and $v \in [\![\tau]\!]\mathcal{I}$ a value then $\rho[x^\tau \mapsto v]$ denotes an environment such that $\mathit{dom}(\rho[x^\tau \mapsto v]) = \mathit{dom}(\rho) \cup \{x^\tau\}$ and

$$(\rho[x^\tau \mapsto v])(y) \equiv \begin{cases} v & \text{if } y = x^\tau \\ \rho(y) & \text{if } y \in \mathit{dom}(\rho) \setminus \{x^\tau\} \end{cases}$$

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

by

$$
\begin{array}{l}
V[\![\pi_a e]\!]\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[\![\mathit{ins}_a e]\!]\mathcal{I}\rho \;\equiv\; (a, V[\![e]\!]\mathcal{I}\rho) \\
V[\![\mathit{case}\, e\, \mathit{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[\![\mathit{add}(e_1, e_2)]\!]\mathcal{I}\rho \;\equiv\; \{V[\![e_1]\!]\mathcal{I}\rho\} \cup V[\![e_2]\!]\mathcal{I}\rho \\
V[\![\mathit{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[\![\mathit{tt}]\!]\mathcal{I}\rho \;\equiv\; \mathbf{T} \\
V[\![\mathit{ff}]\!]\mathcal{I}\rho \;\equiv\; \mathbf{F} \\
V[\![\mathit{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[\![\bar{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}
$$

Note that, for the semantics of an *sri* expression to be well defined, its function argument must be idempotent and commutative in its first argument. In any of our uses of *sri* we will assume that this is the case.

Note also that, if an expression e contains no free variables then its semantics does not depend on the environment ρ. In this case we can write $V[\![e]\!]\mathcal{I}$ for the semantics of e in instance $\mathcal{I}$.

Example 8.2: For the instance described in example 7.2, and the first query of example 8.1,

$$V[\![\mathit{sri}(\lambda x \cdot \lambda y \cdot \mathit{add}(x.\mathit{name}, y), \emptyset, \mathit{State})]\!]\mathcal{I} \;=\; \{\text{“Pennsylvania”}, \text{“New York”}\}$$

and for the second query

$$\begin{array}{l} V[\![\mathit{sri}(\lambda x \cdot \lambda y \cdot if(x.\mathit{state.name} = \text{“Pennsylvania”}, \mathit{add}(x.\mathit{name}, y), y), \emptyset, \mathit{City})]\!]\mathcal{I} \\ \quad = \{\text{“Philadelphia”}, \text{“Pittsburgh”}, \text{“Harrisburg”}\} \end{array}$$

■

Extending *SRI(=)*

In order to make the language *SRI*(=) more usable we will add some additional predicates and logical operators. These do not actually add to the expressive power of the language, but may be thought of as *macros* or short-hand notations for more complicated *SRI* expressions. The typing rules for the extensions are shown in figure 11.

Logical Operators

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

Predicates

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

Figure 11: Extensions to *SRI*(=)

The logical predicates can be defined in terms of the minimal *SRI*(=) as follows:

$$\begin{aligned} e \wedge e' &\equiv if(e, e', ff) \\ e \vee e' &\equiv if(e, tt, e') \\ \neg e &\equiv if(e, ff, tt) \end{aligned}$$

and the predicates $=^{\tau}$ and $\in^{\tau}$, for general object types τ, can be defined by the following induction on types:

$$\begin{aligned} e =^{Bool} e' &\equiv if(e, e', \neg e') \\ e =^{unit} e' &\equiv tt \\ e =^{(a_1:\tau_1,\ldots,a_k:\tau_k)} e' &\equiv (\pi_{a_1} e =^{\tau_1} \pi_{a_1} e') \wedge \ldots \wedge (\pi_{a_k} e =^{\tau_k} \pi_{a_k} e') \\ e =^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle} e' &\equiv case\, e\; of\, a_1(x_1) \Rightarrow (case\, e'\; of\, a_1(y_1) \Rightarrow x_1 =^{\tau_1} y_1, \\ &\qquad\quad a_2(y_2) \Rightarrow ff, \ldots, a_k(y_k) \Rightarrow ff), \\ &\quad \ldots,\; a_k(x_k) \Rightarrow (case\, e'\; of\, a_1(y_1) \Rightarrow ff, \ldots, a_{k-1}(y_{k-1}) \Rightarrow ff, \\ &\qquad\quad a_k(y_k) \Rightarrow x_k =^{\tau_k} y_k) \\ e \in^{\tau} e' &\equiv sri(\lambda x \cdot \lambda u \cdot (x =^{\tau} e) \vee u, ff, e') \\ e =^{\{\tau\}} e' &\equiv sri(\lambda x \cdot \lambda u \cdot (x \in^{\tau} e') \wedge u, tt, e) \wedge \\ &\quad\; sri(\lambda y \cdot \lambda u \cdot (y \in^{\tau} e) \wedge u, tt, e') \end{aligned}$$

In addition we use the shorthand notations

$$\exists x \in e \cdot e' \equiv sri(\lambda x \cdot \lambda u \cdot e' \vee u, ff, e)$$

and

$$\forall x \in e \cdot e' \;\equiv\; \mathit{sri}(\lambda x \cdot \lambda u \cdot e' \wedge u, tt, e)$$

where u does not occur in e, e'.

8.2 Indistinguishable Instances in *SRI*(=)

Two instances $\mathcal{I}$ and $\mathcal{I}'$ are said to be **indistinguishable** in *SRI*(=) iff, for every ground type τ and closed expression e such that $\vdash e : \tau$, $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$.

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

The first lemma is expected, and tells us merely that the semantics of the *SRI*(=) query language is not dependent the partcular choice of object identities in an instance. The substance of the result is in the proof of theorem 8.2.

Lemma 8.1: If $\mathcal{I}$ and $\mathcal{I}'$ are two isomorphic instances of a schema $\mathcal{S}$, say $f^{\mathcal{C}}$ is an isomorphism from $\mathcal{I}$ to $\mathcal{I}'$, then for any *SRI*(=) query e, $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$. ∎

Proof: Recall that in section 7.2 we showed how to extend a family of fuctions on object identity sets, $f^{\mathcal{C}}$, to functions for general object types f^{τ}. We must also extend isomorphisms to cover first rank function types of the form $\tau \to T$. Suppose $f^{\mathcal{C}}$ is an isomorphism from $\mathcal{I}$ to $\mathcal{I}'$, and $g^{\mathcal{C}}$ is the family of inverse functions to the f^{C}'s, so $g^{\mathcal{C}}$ is an isomorphism from $\mathcal{I}'$ to $\mathcal{I}$. For type $T \equiv \tau \to T'$ and $v \in [\![T]\!]\mathcal{I}$, define $f^{T}(v) \in [\![T]\!]\mathcal{I}'$ by

$$(f^{T}(v))(u) \equiv f^{T'}(v(g^{\tau}(u)))$$

If ρ is an in $\mathcal{I}$-environment then we define the $\mathcal{I}'$-environment, $f^{\mathcal{C}}(\rho)$ to be such that $\mathit{dom}(f^{\mathcal{C}}(\rho)) = \mathit{dom}(\rho)$ and, for each $x^{\tau} \in \mathit{dom}(\rho)$, $f^{\mathcal{C}}(\rho)(x^{\tau}) \equiv f^{\tau}(\rho(x^{\tau}))$.

We can now show by induction on *SRI*(=) expressions that, for any expression e such that $\vdash e : \tau$, and any suitable environment ρ,

$$f^{\tau}(V[\![e]\!]\mathcal{I}\rho) \;=\; V[\![e]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho))$$

We will provide some sample cases of the induction only:

1. If e is a constant symbol, say $e \equiv \overline{c}$ for some $c \in \mathbf{D}^{\underline{b}}$, then

$$\begin{aligned} f^{\underline{b}}(V[\![\overline{c}]\!]\mathcal{I}\rho) &= f^{\underline{b}}(c) \\ &= c \\ &= V[\![\overline{c}]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) \end{aligned}$$

2. If $e \equiv x^\tau$, a variable, then

$$\begin{aligned} f^\tau(V[\![x^\tau]\!]\mathcal{I}\rho) &= f^\tau(\rho(x^\tau)) \\ &= (f^{\mathcal{C}}(\rho))(x^\tau) \\ &= V[\![x^\tau]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) \end{aligned}$$

3. If $e \equiv C$, for some class $C \in \mathcal{C}$, then

$$\begin{aligned} f^{\{C\}}(V[\![C]\!]\mathcal{I}\rho) &= f^{\{C\}}(\sigma^C) \\ &= \{f^C(o) | o \in \sigma^C\} \\ &= \sigma'^C \\ &= V[\![C]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) \end{aligned}$$

4. If $e \equiv \pi_{a_i} e'$, where $\vdash e' : (a_1 : \tau_1, \ldots, a_k : \tau_k)$, then

$$\begin{aligned} f^{\tau_i}(V[\![\pi_{a_i} e']\!]\mathcal{I}\rho) &= f^{\tau_i}((V[\![e']\!]\mathcal{I}\rho)(a_i)) \\ &= (f^{(a_1:\tau_1,\ldots,a_k:\tau_k)}(V[\![e']\!]\mathcal{I}\rho))(a_i) \\ &= (V[\![e']\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)))(a_i) \\ &= V[\![\pi_{a_i} e']\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) \end{aligned}$$

5. If $e \equiv \lambda x^t \cdot e'$, then for any $u \in [\![\tau]\!]\mathcal{I}'$

$$\begin{aligned} (f^{\tau \to T}(V[\![\lambda x^t \cdot e']\!]\mathcal{I}\rho))(u) &= f^T((V[\![\lambda x^t \cdot e']\!]\mathcal{I}\rho)(g^\tau(u))) \\ &= f^T(V[\![e']\!]\mathcal{I}(\rho[x^\tau \mapsto g^\tau(u)])) \\ &= V[\![e']\!]\mathcal{I}'(f^{\mathcal{C}}(\rho[x^\tau \mapsto g^\tau(u)]) \\ &= V[\![e']\!]\mathcal{I}'((f^{\mathcal{C}}(\rho))[x^\tau \mapsto f^\tau(g^\tau(u))]) \\ &= V[\![e']\!]\mathcal{I}'((f^{\mathcal{C}}(\rho))[x^\tau \mapsto u]) \\ &= (V[\![\lambda x^\tau \cdot e']\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)))(u) \end{aligned}$$

   where $g^{\mathcal{C}}$ is the inverse of $f^{\mathcal{C}}$. Hence $f^{\tau \to T}(V[\![\lambda x^t \cdot e']\!]\mathcal{I}\rho = V[\![\lambda x^\tau \cdot e']\!]\mathcal{I}'(f^{\mathcal{C}}(\rho))$.

6. Suppose $e \equiv \mathit{sri}(e_1, e_2, e_3)$, and $V[\![e_1]\!]\mathcal{I}\rho = h$, $V[\![e_2]\!]\mathcal{I}\rho = u$, $V[\![e_3]\!]\mathcal{I}\rho = \{v_1, \ldots, \nu_n\}$. By induction hypothesis $V[\![e_3]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) = f^{\{\tau_1\}}(\{v_1, \ldots, v_n\}) = \{f^{\tau_1}(v_1), \ldots, f^{\tau_1}(v_n)\}$, $V[\![e_2]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho)) = f^{\tau_2}(u)$ and for any $v' \in [\![\tau_1]\!]\mathcal{I}'$ and $u' \in [\![\tau_2]\!]\mathcal{I}'$, $V[\![e_1]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho))(v')(u') = f^{\tau_2}(h(g^{\tau_1}(v'))(g^{\tau_2}(u')))$ where $g^{\mathcal{C}}$ is the inverse of $f^{\mathcal{C}}$.

   We prove result by induction on n, the size of $V[\![e_3]\!]\mathcal{I}\rho$. If $n = 0$, then

$$f^{\tau_2}(V[\![\mathit{sri}(e_1, e_2, e_3)]\!]\mathcal{I}\rho) = f^{\tau_2}(u)$$

   If $n \geq 1$ then

$$f^{\tau_2}(V[\![\mathit{sri}(e_1, e_2, e_3)]\!]\mathcal{I}\rho)$$

$$\begin{aligned}
&= f^{\tau_2}(h(v_1, h(v_2, \ldots h(v_n, u) \ldots)) \ldots) \\
&= f^{\tau_2}(h(g^{\tau_1}(f^{\tau_1}(v_1), g^{t_2}(f^{\tau_2}(h(v_2, \ldots h(v_n, u)) \ldots)) \\
&= f^{\tau_1 \to \tau_2 \to \tau_2}(h)(f^{\tau_1}(v_1), f^{\tau_2}(h(v_2, \ldots, h(v_n, u) \ldots))) \\
&= f^{\tau_1 \to \tau_2 \to \tau_2}(h)(f^{\tau_1}(v_1), \ldots f^{\tau_1 \to \tau_2 \to \tau_2}(h)(f^{\tau_1}(v_n), f^{\tau_2}(u)) \ldots)
\end{aligned}$$

Hence $f^{\tau_2}(V[\![sri(e_1, e_2, e_3)]\!]\mathcal{I}\rho) = V[\![sri(e_1, e_2, e_3)]\!]\mathcal{I}'(f^{\mathcal{C}}(\rho))$.

The other cases follow in a similar manner.

Consequently, since for any *ground* type τ, $[\![\tau]\!]\mathcal{I} = [\![\tau]\!]\mathcal{I}'$ and f^τ is the identity on $[\![\tau]\!]\mathcal{I}$, we have

$$V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$$

for any *query* e. ■

Theorem 8.2: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI*(=) if and only if they are isomorphic. ■

Proof: The if part follows from lemma 8.1.

For the only-if part, it suffices to show that, for any instance $\mathcal{I}$, we can construct an expression $e_{\mathcal{I}}$ such that $\vdash e_{\mathcal{I}} : Bool$ and $V[\![e_{\mathcal{I}}]\!]\mathcal{I}'$ is true iff $\mathcal{I}' \cong \mathcal{I}$.

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

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

and

$$\mathcal{V}^C(o_i) = p_i$$

where $p^i \in [\![\tau^C]\!]\mathcal{I}$.

We also assume an implicit ordering on the object identities $o_1, \ldots, o_k$, and fix a sequence of variables, $x_1^C, \ldots, x_k^C$.

For any value $p \in [\![\tau]\!]\mathcal{I}$ we inductively define *SRI*(=) expressions $\widetilde{p}$ with free variables $x_1^C, \ldots, x_k^C$, as follows:

1. If $p \in [\![unit]\!]\mathcal{I}$ then $\widetilde{p} \equiv ()$
2. If $p \in [\![Bool]\!]\mathcal{I}$ then if $p = \mathbf{T}$ then $\widetilde{p} \equiv tt$ otherwise $\widetilde{p} \equiv f\!f$
3. If $p \in [\![\underline{b}]\!]\mathcal{I}$, so $p \in \mathbf{D}^{\underline{b}}$, then $\widetilde{p} \equiv \overline{p}$
4. If $p \in [\![C]\!]\mathcal{I}$, say $p = o_i \in \sigma^C$, then $\widetilde{p} \equiv x_i^C$

5. If $p \in [\![\{\tau\}]\!]\mathcal{I}$, say $p = \{q_1, \ldots, q_n\}$, then $\widetilde{p} \equiv \mathit{add}(\widetilde{q_1}, \ldots \mathit{add}(\widetilde{q_n}, \emptyset) \ldots)$
6. If $p \in [\![(a_1 : \tau_1, \ldots, a_n : \tau_n)]\!]\mathcal{I}$ then $\widetilde{p} \equiv (a_1 = \widetilde{p(a_1)}, \ldots, a_n = \widetilde{p(a_n)})$
7. If $p \in [\![\langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle]\!]\mathcal{I}$, say $p = (a_i, q)$, then $\widetilde{p} \equiv \mathit{ins}_{a_i}\widetilde{q}$.

Then for any type τ and any $p \in [\![\tau]\!]\mathcal{I}$, we have $\vdash \widetilde{p} : \tau$ and

$$V[\![\widetilde{p}]\!]\mathcal{I}(x_1 \mapsto o_1, \ldots, x_k \mapsto o_k) \;=\; p$$

We will use the shorthand expression $\mathit{Dist}(e_1, \ldots, e_n)$ defined by

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

So $V[\![\mathit{Dist}(e_1, \ldots, e_n)]\!]\mathcal{I}\rho = \mathbf{T}$ 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:

$$\begin{aligned} e_{\mathcal{I}} \;\equiv\; & \exists x_1 \in C \cdot \ldots \exists x_k \in C \cdot \\ & \mathit{Dist}(x_1, \ldots, x_k) \wedge \\ & (\forall y \in C \cdot (y = x_1 \vee y = x_2 \vee \ldots \vee y = x_k)) \wedge \\ & (!x_1 = \widetilde{p_1}) \wedge \ldots \wedge (!x_k = \widetilde{p_k}) \end{aligned}$$

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 $V[\![e_{\mathcal{I}}]\!]\mathcal{I}' = \mathbf{T}$ iff $\mathcal{I}' \cong \mathcal{I}$. ■

Example 8.3: Returning to the instance $\mathcal{I}$ described in example 7.2, we construct $e_{\mathcal{I}}$ as

$$\begin{aligned} e_{\mathcal{I}} \;\equiv\; & \exists x_1 \in \mathit{City} \cdot \exists x_2 \in \mathit{City} \cdot \exists x_3 \in \mathit{City} \cdot \exists x_4 \in \mathit{City} \cdot \exists x_5 \in \mathit{City} \cdot \\ & \exists y_1 \in \mathit{State} \cdot \exists y_2 \in \mathit{State} \cdot \\ & (\mathit{Dist}(x_1, x_2, x_3, x_4, x_5) \wedge \mathit{Dist}(y_1, y_2) \wedge \\ & (\forall z \in \mathit{City} \cdot (z = x_1 \vee z = x_2 \vee z = x_3 \vee z = x_4 \vee z = x_5)) \wedge \\ & (\forall w \in \mathit{State} \cdot (w = y_1 \vee w = y_2)) \wedge \\ & x_1 = (\mathit{name} = \text{“Philadelphia”}, \mathit{state} = y_1) \wedge \\ & x_2 = (\mathit{name} = \text{“Pittsburgh”}, \mathit{state} = y_1) \wedge \\ & x_3 = (\mathit{name} = \text{“Harrisburg”}, \mathit{state} = y_1) \wedge \\ & x_4 = (\mathit{name} = \text{“New York City”}, \mathit{state} = y_2) \wedge \\ & x_5 = (\mathit{name} = \text{“Albany”}, \mathit{state} = y_2) \wedge \\ & y_1 = (\mathit{name} = \text{“Pennsylvania”}, \mathit{capital} = x_3) \wedge \\ & y_2 = (\mathit{name} = \text{“New York”}, \mathit{capital} = x_5)) \end{aligned}$$

Then $V[\![e_{\mathcal{I}}]\!]\mathcal{I} = \mathbf{T}$, and for any other instance $\mathcal{I}'$, $V[\![e_{\mathcal{I}}]\!]\mathcal{I}' = \mathbf{T}$ iff $\mathcal{I}' \cong \mathcal{I}$. ■

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: To see that this is true we need to demonstrate that, in any natural query language we can think of, with extensions for handling object identity dereferencing and classes, it is possible to construct an expression $e_{\mathcal{I}}$ equivalent to the one from theorem 8.2. For example the constructors used in the proof of theorem 8.2 do not go beyond those found in the nested relational algebra of [12] or the calculus of [3] without the powerset operator. ■

The 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.

First we must recall our definition of Z-internal transformations from section 5.1 and provide a similar definition for general functions on instances.

Definition 8.3: Suppose that σ is a function from instances of a schema $\mathcal{S}$ to some set D, $\sigma : \mathit{Inst}(\mathcal{S}) \to D$, and Z is a finite set of base values, $Z \subseteq \bigcup_{\underline{b}\in\mathcal{B}} \mathbf{D}^{\underline{b}}$. For each $v \in D$ write $\mathit{Supp}(v)$ for the set of values from $\bigcup_{\underline{b}\in\mathcal{B}} \mathbf{D}^{\underline{b}}$ occuring in v. σ is said to be Z**-internal** iff for any instance $\mathcal{I}$, $\mathit{Supp}(\sigma(\mathcal{I})) \subseteq \mathit{Supp}(\mathcal{I}) \cup Z$. That is σ does not introduce any new base values, other than those in Z. ■

Lemma 8.3: For any closed *SRI*(=) expression, e, there exists a finite set Z such that the mapping $V[\![e]\!]$ is Z-internal. ■

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

Proposition 8.4: For any non-trivial schema (i.e. a schema containing at least one class and admitting a non-trivial instance), it is not possible to build a generic expression in *SRI*(=) which tests whether tests whether two instances are isomorphic. In other words, given a schema $\mathcal{S}$, it is not possible to construct a query $e_{\mathcal{S}}$, 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 $W \subseteq \bigcup_{\underline{b}\in\mathcal{B}} \mathbf{D}^{\underline{b}}$ such that there exist instances $\mathcal{I}$ with $\mathit{Supp}(\mathcal{I}) \subseteq W$. So, for any instance $\mathcal{I}$ with $\mathit{Supp}(\mathcal{I}) \subseteq W$, $V[\![e]\!]\mathcal{I} \in T$

and $Supp(V[\![e]\!]\mathcal{I}) \subseteq W \cup Z$. The set $\{v \in \mathcal{T} \mid Supp(v) \subseteq W \cup Z\}$ 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 of them by introducing duplicates of object identities. It follows by a simple cardinality argument that e can not distinguish between these instances. ■

Claim: For any non-trivial schema $\mathcal{S}$, it is not possible to build a generic expression $e_{\mathcal{S}}$ in any *pure* query language, L, such that $e_{\mathcal{S}}$ distinguishes between all non-isomorphic instances of $\mathcal{S}$.

Justification: By saying that a language L is a "pure" query language we mean that it can express operations which extract, manipulate and compare data from an instance, but cannot perform general computations, such as arithmetic. In particular any query expressible in such a language should not create any new base values, other possible than those belonging to a finite set of constants that occur in the query. Consequently any query expressible in a pure query language should be Z-internal for some finite set Z, and so the proof of proposition 8.4 can be applied.

9 Bisimulation and Observational Equivalence without Equality

It seems clear that the object-identity based model of definition 7.3 captures our intuition about how databases with recursive values and extents are represented. However, since object identities are not normally considered to be directly observable, or to have meaning outside the internal representation of a database, it follows that there can be many indistinguishable instances in this model. Any particular query system or set of assumptions about what queries may be asked of an instance will lead to an observational equivalence relation on instances: two instances being equivalent when they are indistinguishable under that query system.

In section 7.2 we introduced the concept of *isomorphism* of instances to represent when two instances differed only their choice of object identities, and commented that, intuitively, any two isomorphic instances should not be distinguishable by any query mechanism. As such, isomorphism should be at least as fine as any possible observational equivalence relation. In section 8 we showed that, given a reasonably expressive query language with an equality test on object-identities, observational equivalence coincides precisely with isomorphism. However it is questionable whether including a direct equality test on object identities in a query language is reasonable: there may be situations where multiple objects represent the same data, and we do not wish to distinguish between distinct object identities if they have the same observable values associated with them. This is particularly true, for example, when dealing with federated database systems where data may be replicated over several distinct instances. We would therefore like a notion of equivalence of schemas which captures the idea that two instances have the same printable data, though the number and interconnections of the object-identities used to represent that data may differ.

In section 9.1 we will define *bisimulation equivalence* on instances and values, and argue that intuitively bisimulation is at least as coarse as any observable equivalence relation on instances for any reasonable query system. In section 9.2 we will show that bisimulation actually coincides with observational equivalence for a query language with no means of derectly comparing object-identities.

9.1 Bisimulation and Corespondence Relations

The definition of bisimulation relations will proceed in stages: we start by defining *correspondence relations* between the object-identities of two instances, and then define bisimulation in terms correspondence relations.

Definition 9.1: A **correspondence** between two families of object identifiers $\sigma^{\mathcal{C}}$ and $\sigma'^{\mathcal{C}}$ is a family of binary relations $\sim^C \subseteq \sigma^C \times \sigma'^C$, $C \in \mathcal{C}$.

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

1. $c \sim^{\underline{b}} c$ 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 $a_i = a_j$ 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 $\mathcal{V}^C(o) \sim^{\tau^C} \mathcal{V}'^C(o')$.

Intuitively a correspondence relation is some possible correspondence between the object identities of two instances. A correspondence relation is consistent iff it is not at odds with the printable values associated with the objects of that instance.

Lemma 9.1: Given any set of consistent correspondences between the instances $\mathcal{I}$ and $\mathcal{I}'$ the correspondence formed by taking the union of the relations for each class $C \in \mathcal{C}$ from each of the set of correspondences is also consistent for $\mathcal{I}$ and $\mathcal{I}'$. ■

Proof: Suppose we have a family of correspondence relations, $\{\sim^{\mathcal{C}}_a | a \in A\}$ between object-identities $\sigma^{\mathcal{C}}$ and $\sigma'^{\mathcal{C}}$. Consider the correspondence relation $\sim^{\mathcal{C}}$ defined by $(\sim^C) \equiv \bigcup_{a\in A}(\sim^C_a)$ for each $C \in \mathcal{C}$. Now $\sim^{\mathcal{C}}$ can be extended to binary relations on general types $\sim^\tau$, and for any type τ and any $a \in A$, $(\sim^\tau_a) \subseteq (\sim^\tau)$.

Suppose instances $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and $\mathcal{I}' = (\sigma'^{\mathcal{C}}, \mathcal{V}'^{\mathcal{C}})$ are such that, for each $a \in A$, $\sim^{\mathcal{C}}_a$ is consistent with $\mathcal{I}$ and $\mathcal{I}'$. Suppose that $C \in \mathcal{C}$, $o \in \sigma^C$, $o' \in \sigma'^C$ are such that $o \sim^C o'$. Then there is an $a \in A$ such that $o \sim^C_a o'$. Hence $\mathcal{V}^C(o) \sim^{\tau^C}_a \mathcal{V}'^C(o')$, and so $\mathcal{V}^C(o) \sim^{\tau^C} \mathcal{V}'^C(o')$. Hence $\sim^{\mathcal{C}}$ is also consistent with $\mathcal{I}$ and $\mathcal{I}'$. ■

Definition 9.2: Let $\mathcal{I}$, $\mathcal{I}'$ be instances of a schema S. Then we define the **bisimulation** correspondence, ${}_{\mathcal{I}}{\approx}^{\mathcal{C}}_{\mathcal{I}'}$, between $\mathcal{I}$ and $\mathcal{I}'$ to be the *largest* consistent correspondence between $\mathcal{I}$ and $\mathcal{I}'$ (the existence of which follows from the previous lemma). Since the subscripts are rather cumbersome here, and are usually clear from context, we will frequently omit them.

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 $o\,{}_{\mathcal{I}}{\approx}_{\mathcal{I}'}\,o'$,

■

The bisimulation correspondence between two instances therefore links any two object-identities that can be linked by some consistent correspondence between the two instances, and therefore cannot be distinguished by the non-object-identity (printable) part of the values associated with the objects. Two instances are bisimilar iff for each object-identity occuring in one instance there is a corresponding object-identity in the other instance such that the two objects can not be distinguished by looking at the associated values.

Proposition 9.2: The relation $\approx$ is an equivalence on the set $\mathit{Inst}(\mathcal{S})$ of instances of a schema $\mathcal{S}$. ■

Proof: It is clear that $\approx$ is a reflexsive and symmetric relation on instances. We shall prove that it is transitive.

Suppose $\mathcal{I}, \mathcal{I}', \mathcal{I}'' \in \mathit{Inst}(\mathcal{S})$ are such that $\mathcal{I} \approx \mathcal{I}'$ and $\mathcal{I}' \approx \mathcal{I}''$, where $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, $\mathcal{I}' = (\sigma'^{\mathcal{C}}, \mathcal{V}'^{\mathcal{C}})$ and $\mathcal{I}'' = (\sigma''^{\mathcal{C}}, \mathcal{V}''^{\mathcal{C}})$. Define the corresponce relation $\sim^{\mathcal{C}}$ between $\sigma^{\mathcal{C}}$ and $\sigma''^{\mathcal{C}}$ to be such that, for any $C \in \mathcal{C}$, $o \in \sigma^C$ and $o'' \in \sigma''^C$, $o \sim^C o''$ iff there exists a $o' \in \sigma'^C$ such that $o \approx^C o'$ and $o' \approx^C o''$.

Then for any type τ, $v \in [\![\tau]\!]\sigma^{\mathcal{C}}$, $v' \in [\![\tau]\!]\sigma'^{\mathcal{C}}$ and $\nu'' \in [\![\tau]\!]\sigma''^{\mathcal{C}}$, if $v \approx^\tau v'$ and $v' \approx^\tau v''$ then $v \sim^\tau v''$.

Suppose $C \in \mathcal{C}$, $o \in \sigma^C$, $o'' \in \sigma''^C$ are such that $o \sim^C o''$. Then $o \approx^C o'$ and $o' \approx^C o''$ for some $o' \in \sigma'^C$. Hence $\mathcal{V}^C(o) \approx^{\tau^C} \mathcal{V}'^C(o')$ and $\mathcal{V}'^C(o') \approx^{\tau^C} \mathcal{V}''^C(o'')$. Hence $\mathcal{V}^C(o) \sim^{\tau^C} \mathcal{V}''^C(o'')$. Hence $\sim^{\mathcal{C}}$ is a consistent correspondence between $\mathcal{I}$ and $\mathcal{I}''$.

But, for every $o \in \sigma^C$ there is an $o' \in \sigma'^C$ such that $o \approx^C o'$, and for every $o' \in \sigma'^C$ there is an $o'' \in \sigma''^C$ such that $o' \approx^C o''$. So for every $o \in \sigma^C$ there is an $o'' \in \sigma''^C$ such that $o \sim^C o'$, and, since $\sim^{\mathcal{C}}$ is a consistent correspondence, $o \approx^C o''$. Similarly for every $o'' \in \sigma''^C$ there is an $o \in \sigma^C$ such that $o \approx^C o''$.

Hence $\mathcal{I} \approx \mathcal{I}''$. ■

Proposition 9.3: For each equivalence class $[\mathcal{I}]_{\approx}$ there is an $\mathcal{I}' \in [\mathcal{I}]_{\approx}$, unique up to isomorphism, such that, for any $\mathcal{I}'' \in [\mathcal{I}]_{\approx}$ there is a unique homomorphism, $f^{\mathcal{C}}$ from $\mathcal{I}''$ to $\mathcal{I}'$. Such an $\mathcal{I}'$ is said to be a *canonical representative* of $[\mathcal{I}]_{\approx}$. ■

Proof: Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance of $\mathcal{S}$. We must first build an $\mathcal{I}'$ such that $\mathcal{I} \approx \mathcal{I}'$, and for any $\mathcal{I}''$ such that $\mathcal{I}'' \approx \mathcal{I}'$, there is a unique homomorphism from $\mathcal{I}''$ to $\mathcal{I}'$.

Consider the bisimulation correspondence ${}_{\mathcal{I}}\!\approx^{\mathcal{C}}_{\mathcal{I}}$ (which we will simplify to $\approx^{\mathcal{C}}$). Note that, for each C, $\approx^C$ is an equivalence relation on σ^C: to see this observe that the equivalence relations generated by the binary relations $\approx^C$, $\approx^{C^*}$ say, themselves form a consistent correspondence on $\mathcal{I}$, and hence $(\approx^{C^*}) \subseteq (\approx^C)$ for each $C \in \mathcal{C}$.

For each $C \in \mathcal{C}$ define $\sigma'^C \equiv \sigma^C/_{\approx^C}$: the $\approx^C$-equivalence classes of σ^C. For any value $v \in [\![\tau]\!]\sigma^{\mathcal{C}}$ we define an object $\widetilde{v} \in [\![\tau]\!]\sigma' C$ using the following induction on types:

1. If $v \in [\![\underline{b}]\!]\sigma^{\mathcal{C}}$, $\underline{b} \in \mathcal{B}$, then $\widetilde{v} \equiv v$
2. If $v \in [\![C]\!]\sigma^{\mathcal{C}}$, $C \in \mathcal{C}$, that is $v \in \sigma^C$, then $\widetilde{v} \equiv [v]_{\approx^C}$, where $[v]_{\approx^C} \in \sigma'^C$ is such that $v \in [v]_{\approx^C}$.
3. If $v \in [\![(a_1 : \tau_1, \ldots, a_n : \tau_n)]\!]\sigma^{\mathcal{C}}$ then $\widetilde{v}(a_i) \equiv \widetilde{v(a_i)}$ for $i = 1, \ldots, n$.
4. If $v \in [\![\langle\!|a_1 : \tau_1, \ldots, a_n : \tau_n|\!\rangle]\!]\sigma^{\mathcal{C}}$, say $v = (a_i, w)$, then $\widetilde{v} = (a_i, \widetilde{w})$.
5. If $v \in [\![\{\tau\}]\!]\sigma^{\mathcal{C}}$ then $\widetilde{v} = \{\widetilde{w} | w \in v\}$.

Then for any type τ and $u, v \in [\![\tau]\!]\sigma^{\mathcal{C}}$, $u \approx^{\tau} v$ iff $\widetilde{u} = \widetilde{v}$.

For each $C \in \mathcal{C}$ define $\mathcal{V}'^C$ by $\mathcal{V}'^C([o]_{\approx^C}) \equiv \widetilde{\mathcal{V}^C(o)}$. From the above $\mathcal{V}'^C$ is well defined since $\approx^{\mathcal{C}}$ is consistent.

Let $\mathcal{I}'$ be the instance $\mathcal{I}' = (\sigma'^C, \mathcal{V}'^C)$. Then define the correspondece $\sim^{\mathcal{C}}$ between $\mathcal{I}$ and $\mathcal{I}'$ such that $o \sim^C o'$ iff $o \in o'$. Then $\sim^{\mathcal{C}}$ is a consistent correspondence. Further, for each $o \in \sigma^C$ there is an $o' \in \sigma'^C$ such that $o \sim^C o'$, and for each $o' \in \sigma'^C$ there is an $o \in \sigma^C$ such that $o \sim^C o'$. Hence $\mathcal{I} \approx \mathcal{I}'$. Further for any $o'_1, o'_2 \in \sigma'^C$, if $o'_1 \approx^C o'_2$ then $o'_1 = o'_2$.

Suppose $\mathcal{I}'' = (\sigma''^{\mathcal{C}}, \mathcal{V}''^{\mathcal{C}})$ is such that $\mathcal{I}'' \approx \mathcal{I}$. Then $\mathcal{I}'' \approx \mathcal{I}'$. For each $C \in \mathcal{C}$ define the function $f^C : \sigma''^C \to \sigma'^C$ as follows: for any $o'' \in \sigma''^C$ choose $o \in \sigma'^C$ such that $o'' \approx^C o'$ and let $f^C(o'') = o'$. f^C is well defined since, if $o'' \approx^C o_1$ and $o'' \approx^C o_2$ then $o_1 \approx^C o_2$ and hence $o_1 = o_2$. Further f^C is a homomorphism from $\mathcal{I}''$ to $\mathcal{I}'$.

Suppose $h^{\mathcal{C}}$ is a homomorphism from $\mathcal{I}''$ to $\mathcal{I}'$. Then we can form a consistent correspondence $\sim^{\mathcal{C}}$ between $\mathcal{I}''$ and $\mathcal{I}'$ by taking $o'' \sim^C o'$ iff $o' = h^C(o'')$. Hence for each $C \in \mathcal{C}$, each $o'' \in \sigma''^C$, $o'' \approx^C h^C(o'')$, and so $h^C(o'') = f^C(o'')$. Hence $h^{\mathcal{C}} = f^{\mathcal{C}}$ and $f^{\mathcal{C}}$ is the unique homomorphism from $\mathcal{I}''$ to $\mathcal{I}'$.

It remains to show that $\mathcal{I}'$ is the unique-up-to-isomorphism instance satisfying these properties. Suppose that $\mathcal{I}''$ is another instance satisfying these properties. Then there exists a homomorphism $f^{\mathcal{C}}$ from $\mathcal{I}''$ to $\mathcal{I}'$, and a homomorphism $g^{\mathcal{C}}$ from $\mathcal{I}'$ to $\mathcal{I}''$. We can form a homomorphism, $h^{\mathcal{C}}$ from $\mathcal{I}'$ to itself by taking $h^C = f^C \circ g^C$ for $C \in \mathcal{C}$. But the family of identity maps $Id^{\mathcal{C}}$ is a homomorphism from $\mathcal{I}'$ to itself, so we get $h^{\mathcal{C}} = Id^{\mathcal{C}}$, and hence $f^C \circ g^C = Id_{\sigma'^C}$ for each $C \in \mathcal{C}$, where $Id_{\sigma' C}$ denotes the identity function on $\sigma' C$. Similarly $g^C \circ f^C = Id_{\sigma'' C}$ for $C \in \mathcal{C}$. Hence $\mathcal{I}'$ and $\mathcal{I}''$ are isomorphic. ∎

The canonical representative of an equivalence class $[\mathcal{I}]_\approx$ is therefore an instance $\mathcal{I}'$ in which any bisimilar object identities are coalesced into a single object identity: for every object identity in $\mathcal{I}$, $o \in \sigma^C$ say, there is a *unique* object identity in $\mathcal{I}'$, $o' \in \sigma'^C$, such that $o\,_{\mathcal{I}}\!\approx_{\mathcal{I}'} o'$.

9.2 Distinguishing Instances without Equality on Identities

We now introduce a variant on the language *SRI*(=), which we will call simply *SRI*. This is the same as the language *SRI*(=) only without the $=^C$ predicates on object identities. So *SRI* gives us no way of directly comparing object identities.

We will show that observational indistinguishability of instances in *SRI* coincides with the bisimulation correspondence on instances, $\approx$, defined in definition 9.2, and, further, that the relation $\approx_{\mathcal{I}}$ on values of an instance $\mathcal{I}$ can be computed using *SRI*.

Indistinguishable Instances in *SRI*

Lemma 9.4: If $\mathcal{I}$ and $\mathcal{I}'$ are two instances of a schema $\mathcal{S}$ such that $\mathcal{I} \approx \mathcal{I}'$ then $\mathcal{I}$ and $\mathcal{I}'$ are indistinguishable in *SRI*. ∎

Proof: Assume that $\mathcal{I}$ and $\mathcal{I}'$ are such that $\mathcal{I} \approx \mathcal{I}'$. We need to show that, for any *SRI* query e, $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$. Note that, for a *ground* type τ, $\approx^\tau$ coincides with equality. We will show that for any closed expression e such that $\vdash e : \tau$, $V[\![e]\!]\mathcal{I} \approx^\tau V[\![e]\!]\mathcal{I}'$.

We must first expand the definition of $\approx^\tau$ to function types. Suppose $f \in [\![\tau \to T]\!]\mathcal{I}$ and $g \in [\![\tau \to T]\!]\mathcal{I}'$. We say $f \approx^{\tau \to T} g$ iff for any $u \in [\![\tau]\!]_{\mathcal{I}}$ and $v \in [\![\tau]\!]_{\mathcal{I}'}$ if $u \approx^\tau v$ then $fu \approx^T gv$.

Now it suffices to show that, if e is any *SRI* expression, $\vdash e : T$, and $\rho \in Env(\mathcal{I})$ and $\rho' \in Env(\mathcal{I}')$ are environments such that $dom(\rho) = dom(\rho')$ and, for each variable $x^\tau \in dom(\rho)$, $\rho(x^\tau) \approx^\tau \rho'(x^\tau)$, then

$$V[\![e]\!]\mathcal{I}\rho \approx^T V[\![e]\!]\mathcal{I}'\rho'$$

The proof proceeds by induction on the structure of *SRI* expressions and is similar to the proof of lemma 8.1. ∎

Lemma 9.5: If $\mathcal{I}$ and $\mathcal{I}'$ are indistinguishable in *SRI* then they are bisimilar, $\mathcal{I} \approx \mathcal{I}'$. ∎

Proof: Suppose otherwise: that is there exist instances $\mathcal{I}$ and $\mathcal{I}'$ such that $\mathcal{I} \not\approx \mathcal{I}'$ but, for every query e, $V[\![e]\!]\mathcal{I} = V[\![e]\!]\mathcal{I}'$. We can assume without loss of generallity that for some class $C \in \mathcal{C}$ there is an $o \in \sigma^C$ such that there is no $o' \in \sigma'^C$ for which $o \approx o'$.

Given any type, τ, value $v \in [\![\tau]\!]\mathcal{I}$ and expression e such that $\vdash e : \tau$, we define a series of expressions $Test^i(v)[e]$, for $i \in \mathbb{N}$, such that $\vdash Test^i(v)[e] : Bool$. We define $Test^i(v)[e]$ by co-induction on i and on the type of v:

For any value v and expression e, $\mathit{Test}^0(v)[e] \equiv tt$.

For $i \geq 1$:

1. If $v \in [\![\mathit{int}]\!]\mathcal{I} = \mathbb{N}$ then $\mathit{Test}^i(v)[e] \equiv (e = \overline{v})$.
2. If $v \in [\![\mathit{Bool}]\!]\mathcal{I}$ then if $v = \mathbf{T}$ then $\mathit{Test}^i(v)[e] \equiv (e)$ otherwise $\mathit{Test}^i(v)[e] \equiv (\neg e)$.
3. If $v \in [\![\mathit{unit}]\!]\mathcal{I}$ then $\mathit{Test}^i(v)[e] \equiv (tt)$.
4. If $v \in [\![C]\!]\mathcal{I}$ then $\mathit{Test}^i(v)[e] \equiv \mathit{Test}^{i-1}(\mathcal{V}^C(v))[!e]$
5. If $v \in [\![(a_1 : \tau_1, \ldots, a_n : \tau_n)]\!]\mathcal{I}$ then $\mathit{Test}^i(v)[e] \equiv (\mathit{Test}^i(v(a_1))[e.a_1] \wedge \ldots \wedge \mathit{Test}^i(v(a_n))[e.a_n])$
6. If $v \in [\![\langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle]\!]\mathcal{I}$, say $v = (a_l, u)$ then $\mathit{Test}^i(v)[e] \equiv (\mathit{case}\ e\ \mathit{of}\ a_1(x_1) \Rightarrow \mathit{ff}, \ldots, a_l(x_l) \Rightarrow \mathit{Test}^i(u)[x_l], \ldots, a_k(x_k) \Rightarrow \mathit{ff})$
7. If $v \in [\![\{\tau\}]\!]\mathcal{I}$, say $v = \{u_1, \ldots, u_n\}$, then $\mathit{Test}^i(v)[e] \equiv ((\exists x \in e \cdot \mathit{Test}^i(u_1)[x]) \wedge \ldots \wedge (\exists x \in e \cdot \mathit{Test}^i(u_n)[x]) \wedge (\forall x \in e \cdot (\mathit{Test}^i(u_1)[x] \vee \ldots \vee \mathit{Test}^i(u_n)[x])))$.

Intuitively, for any expression e and value v, $V[\![\mathit{Test}^i(v)[e]]\!]\mathcal{I}\rho = \mathbf{T}$ iff the value $V[\![e]\!]\mathcal{I}\rho$ cannot be distinguished from v using $i-1$ levels of dereferencing of object-identities.

Then for each $C \in \mathcal{C}$, each $o \in \sigma^C$ and any $i \in \mathbb{N}$ we have $V[\![\exists x \in C \cdot \mathit{Test}^i[o](x)]\!]\mathcal{I} = \mathbf{T}$. Hence, since $\mathcal{I}$ and $\mathcal{I}'$ are indistinguishable, $V[\![\exists x \in C \cdot \mathit{Test}^i[o](x)]\!]\mathcal{I}' = \mathbf{T}$ for all i. And so for each $C \in \mathcal{C}$, $o \in \sigma^C$, we can form a decreasing series of finite *non-empty* sets $E^i_o \subseteq \sigma'^C$ defined by

$$E^i_o \equiv \{o' \in \sigma'^C \mid V[\![\mathit{Test}^i(o)[x]]\!]\mathcal{I}'(x \mapsto o') = \mathbf{T}\}$$

That is $E^{i+1}_o \subseteq E^i_o$ and $E^i_o \neq \emptyset$ for all $i \in \mathbb{N}$.

We can then build a series of correspondence relations $\sim^{\mathcal{C}}_i$, $i \in \mathbb{N}$ such that, for any $C \in \mathcal{C}$, $o \in \sigma^C$ and $o' \in \sigma'^C$, $o \sim^C_i o'$ iff $o' \in E^i_o$.

Then, for each $C \in \mathcal{C}$, $o \in \sigma^C$ it follows that $\{o' \in \sigma'^C | o \sim^C_{i+1} o'\} \subseteq \{o' \in \sigma'^C | o \sim^C_i o'\} \neq \emptyset$. So we can form a correspondence $\sim^{\mathcal{C}}_\infty$, defined by $\sim^C_\infty \equiv \bigcap_{i=1}^\infty \sim^C_i$, such that for each $o \in \sigma^C$ the set $\{o' \in \sigma'^C | o \sim^C_\infty o'\}$ is non-empty. It remains to show that this correspondence relation is consistent.

Observe that, for any $C \in \mathcal{C}$, $o \in \sigma^C$, $o' \in \sigma'^C$, if $i \geq 1$ and $o \sim^C_i o'$ then $\mathcal{V}^C(o) \sim^{\tau^C}_{i-1} \mathcal{V}'^C(o')$.

Also observe that each of the operators used in raising correspondence relations to general types in definition 9.1 was a continuous operator with respect to the set-inclusion ordering on correspondence relations. Hence, for any general object type τ, $(\sim^\tau_\infty) = \bigcap_{i=1}^\infty (\sim^\tau_i)$.

Hence, if $o \in \sigma^C$, $o' \in \sigma'^C$ are such that $o \sim^C_\infty o'$, then $o \sim^C_i o'$ for all $i \in \mathbb{N}$, so $\mathcal{V}^C(o) \sim^{\tau^C}_{i-1} \mathcal{V}^C(o')$ for all $i \geq 1$, and so $\mathcal{V}^C(o) \sim^{\tau^C}_\infty \mathcal{V}^C(o')$.

Hence $\sim^{\mathcal{C}}_\infty$ is a consistent correspondence relation: a contridiction. Hence result. ■

Proposition 9.6: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI* if and only if $\mathcal{I} \approx \mathcal{I}'$. ■

Proof: This follows directly from lemmas 9.4 and 9.5. ■

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. Lemma 9.4 automatically holds for any query languages less expressive than SRI.

The proof of lemma 9.5 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. ■

Testing for Correspondence in *SRI*

Proposition 9.7: Using *SRI* we can test for the bisimulation correspondence relation described in section 9.1: that is, for any type τ, and any values u and v, $u, v \in [\![\tau]\!]\mathcal{I}$, we can form a function expression $Cor^\tau : (\tau \times \tau) \to Bool$ such that $V[\![Cor]\!]_{\mathcal{I}}(u, v) = \mathbf{T}$ iff $u \approx^\tau v$.

(The notation $\tau \times \tau'$ represents a Cartesian product and is not actually a type constructor in our model, but can be considered to be a notational abbreviation for a record constructor with two attributes: $(\#1 : \tau, \#2 : \tau')$). ■

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

Proof: We will show how to build such a function in the case where the schema, $\mathcal{S}$, contains only one class, C, though again the definition can be easily extended for the case when there are many classes.

First we will define some more "macros" for *SRI*:

$$\begin{aligned}
Map(f, X) &\equiv sri(\lambda x \cdot \lambda Y \cdot add(fx, Y), \emptyset, X) \\
Flatten(X) &\equiv sri(\lambda x \cdot \lambda Y \cdot x \cup Y, \emptyset, X) \\
Prod(X, Y) &\equiv sri(\lambda x \cdot \lambda z \cdot sri(\lambda y \cdot \lambda w \cdot add((x, y), w), z, Y), \emptyset, X) \\
UnionProd(X, Y) &\equiv Map(\lambda x \cdot x.\#1 \cup x.\#2, Prod(X, Y))
\end{aligned}$$

Here *Map* and *Flatten* are the standard operators. *Prod* is the cartesian product operator, and *UnionProd* maps the union operator over the cartesian product of two sets.

For each object type τ we construct a function $Check^\tau : (\tau \times \tau) \to \{\{C \times C\}\}$ such that, if $Check^\tau(e, e') = X$, then $[\![e]\!] \approx^\tau [\![e']\!]$ iff, for some set $Y \in X$, $o \approx^C o'$ for each pair $(o, o') \in Y$.

Note that, if $\mathit{Check}^\tau(e,e') = \emptyset$ then $[\![e]\!] \not\approx [\![e']\!]$, and if $\mathit{Check}^\tau(e,e') = \{\emptyset\}$ then $[\![e]\!] \approx [\![e']\!]$. We will give some of the cases in the definition of Check^τ.

$$\begin{aligned}
\mathit{Check}^{\mathit{unit}}(e,e') &\equiv \{\emptyset\}\\
\mathit{Check}^{\mathit{Bool}}(e,e') &\equiv \mathit{if}(e \doteq e', \{\emptyset\}, \emptyset)\\
\mathit{Check}^{C}(e,e') &\equiv \{\{(e,e')\}\}\\
\mathit{Check}^{(a_1:\tau_1,\ldots,a_k:\tau_k)}(e,e') &\equiv \mathit{UnionProd}(\mathit{Check}^{\tau_1}(e.a_1, e'.a_1), \mathit{UnionProd}(\ldots,\\
&\qquad \mathit{Check}^{\tau_k}(e.a_k, e'.a_k))\ldots)\\
\mathit{Check}^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle}(e,e') &\equiv \mathit{case}\ e \mathit{of}\ a_1(x_1) \Rightarrow (\mathit{case}\ e' \mathit{of}\ a_1(y_1) \Rightarrow \mathit{Check}^{\tau_1}(x_1,y_1),\\
&\qquad\quad a_2(y_2) \Rightarrow \emptyset, \ldots, a_k(y_k) \Rightarrow \emptyset),\\
&\quad a_k(x_k) \Rightarrow (\mathit{case}\ e' \mathit{of}\ a_1(y_1) \Rightarrow \emptyset, \ldots, a_{k-1}(y_{k-1}) \Rightarrow \emptyset,\\
&\qquad\quad a_k(y_k) \Rightarrow \mathit{Check}^{\tau_k}(x_k,y_k))\\
\mathit{Check}^{\{\tau\}}(e,e') &\equiv \mathit{sri}(\lambda x \cdot \lambda Z \cdot \mathit{Map}(\lambda y \cdot \mathit{Map}(\lambda W \cdot\\
&\qquad \mathit{UnionProd}(\mathit{Check}^{\tau}(x,y), W), Z), e'), \emptyset, e)
\end{aligned}$$

The next step is a function $\mathit{IterChk} : \{C \times C\} \to \{\{C \times C\}\}$ which iterates the function Check^{τ^C} over a set.

$$\mathit{IterChk}(Y) \ \equiv\ \mathit{sri}(\lambda x \cdot \lambda Z \cdot \mathit{UnionProd}(\mathit{Check}^{\tau^C}(!(x.\#1), !(x.\#2)), Z), \emptyset, Y)$$

The function $\mathit{Unfold} : \{\{\mathcal{C} \times C\}\} \to \{\{C \times C\}\}$ applies $\mathit{IterChk}$ to each element in a set and flattens the result.

$$\mathit{Unfold}(Z) \ \equiv\ \mathit{Flatten}(\mathit{Map}(\lambda x \cdot \mathit{IterChk}(x), Z))$$

So a pair of expressions, e and e', can be shown not to be bisimilar using N levels of dereferencing iff the result of applying *Unfold* to $\mathit{Check}(e,e')$ N times is the empty set. But we know that, if e and e' are not bisimilar then they can be shown not to be bisimilar in less than $|\sigma^C|$ steps. So we define

$$\mathit{TestCor}^\tau(e,e') \ \equiv\ \mathit{sri}(\lambda x \cdot \lambda Z \cdot \mathit{Unfold}(Z), \mathit{Check}^\tau(e,e'), C)$$

Finally we can use this set in testing for $\approx$-equivalence of values:

$$\mathit{Cor}^\tau(e,e') \quad\equiv\quad (\exists x \in \mathit{TestCor}^\tau(e,e') \cdot\ )$$

Then $[\![\mathit{Cor}^\tau(e,e')]\!] = \mathbf{T}$ iff $[\![e]\!] \approx^\tau [\![e']\!]$. ■

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, and we would like to have algorithms to compare values which take time dependent on the size of the values being compared only. We would therefore like to know if we can test for $\approx$ without iterating over the extents of an instance. In the following subsection we will show that this is not possible in general.

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 $o \in \sigma^C$,$\mathcal{V}(o)$ 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 9.8: 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 schemas $\mathcal{S}$ and type τ such that, for any expression *Cor* with $\vdash Cor : \tau \times \tau \to Bool$, either there is an N-bounded instance $\mathcal{I}$ and values u and v, $u, v \in [\![\tau]\!]\mathcal{I}$, such that $V[\![Cor]\!]_{\mathcal{I}}(u, v) = \mathbf{T}$ and $u \not\approx^{\tau} v$, or there is an N-bounded instance $\mathcal{I}$ and values $u, v \in [\![\tau]\!]\mathcal{I}$ such that $V[\![Cor]\!]_{\mathcal{I}}(u, v) = \mathbf{F}$ and $u \approx^{\tau} v$. ■

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 levels of dereferencing 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.

For example one can take the schema $\mathcal{S}$ with one class C, such that $\mathcal{S}(C) = (\#1 : Bool, \#2 : C)$. Then we can take $\mathcal{I}$ and $\mathcal{I}'$ both to have the same set of $k + 1$ object identities, $\sigma^C = \sigma'^C = \{o_0, \ldots, o_k\}$, and $\mathcal{V}^C(o_i) = \mathcal{V}'^C(o_i) = (\#1 \mapsto \mathbf{T}, \#2 \mapsto o_{i+1})$ for $i = 0, \ldots, k - 1$, $\mathcal{V}^C(o_k) = (\#1 \mapsto \mathbf{T}, \#2 \mapsto o_0)$ and $\mathcal{V}'^C(o_k) = (\#1 \mapsto \mathbf{F}, \#2 \mapsto o_0)$. Then the object identity o_0 can not be distinguished with less than $k + 1$ dereferences. ■

This tells us that we can not hope to test if two values are equivalent without making use of recursion over classes, from which we conclude that a more efficient way of comparing values is needed.

Claim: It is not possible to construct an expression to test whether two values of general type are bisimilar in any reasonable query language without using recursion over the entire extents of a database. ■

Justification: Once again it is sufficient to note that the result automatically holds for any language less expressive than *SRI* and to argue that *SRI* is as expressive as any other established query language pradigm. ■

10 Observable Properties of Object Identities with Keys

In sections 8.1 and 9.2 we presented two different query languages, based on different assumptions about the predicates available for comparing values. The first, $SRI(=)$, assumed that it was possible to directly compare any two object identities for equality, and we showed that such a predicate, together with a simple query language over complex objects, allowed us to compute the equivalence relation $=$ over all types, and was sufficient to distinguish between any two non-isomorphic instances.

However object identities are abstract entities that do not directly represent data, and so we would like to ensure that they can only be compared by means of their associated values. Our second query language, SRI, was based on the idea that only base values could be directly compared, and that other complex values and objects could be compared only by comparing their component parts or associated values. In proposition 9.6 we showed that distinguishability under such a language coincided precisely with the bisimulation relation on instances, $\approx$, defined in 9.2. In proposition 9.7 we saw that such a bisimulation equivalence relation, $\approx$, on values in an instance could be computed, but proposition 9.8 showed that doing so required a level of unfolding bounded by the size of the instance, and therefore the ability to recurse over all object identities in the instance. Allowing such a computation seems at odds with our premise, that object identities, and hence the cardinality of a particular class of object identities, could not be observed. From a more pragmatic perspective, it is clear that such an equivalence relation is too expensive to use in a query language over databases, and a more efficient way of comparing values and object identities is required. In particular, we want to be able to compare values in time dependent on the values themselves and independent of the size of the database. In this section we will propose a solution to this problem based on the systems of *keys*.

10.1 A Data-Model with Keys

In the model described so far, object identities represent abstract entities which can not be directly observed, but which can only be viewed by examining the values associated with them. However, in section 9 we saw that comparing and referencing object-idenitities only on the basis of the printable values associated with them is not a practical proposition. It is necessary to have some efficient means of uniquely identifying and referencing an object identity. One possible solution, which is adopted by many practical database systems, is to use *keys*: simple values that are associated with object identities, and are used to compare object identities. Two object identities are taken to be the same iff their keys are the same.

Key Specifications

Definition 10.1: 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 $C \in \mathcal{C}$, and a mapping $\mathcal{K}^{\mathcal{C}}$ from instances, $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, of $\mathcal{S}$ to families of functions

$$\mathcal{K}^C_{\mathcal{I}} : \sigma^C \to [\![\kappa^C]\!]\sigma^{\mathcal{C}}$$

for each $C \in \mathcal{C}$. ■

Example 10.1: Consider the first schema described in example 7.1. 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

$$\begin{aligned}\mathcal{K}^{City}_{\mathcal{I}}(o) &\equiv \mathcal{V}^{City}(o)\\ \mathcal{K}^{State}_{\mathcal{I}}(o) &\equiv (\mathcal{V}^{State}(o))(name)\end{aligned}$$

If we take $\mathcal{I}$ to be the instance described in example 7.2 then we have

$$\begin{aligned}\mathcal{K}^{City}_{\mathcal{I}}(Phila) &= (name \mapsto \text{“Philadelphia”}, state \mapsto PA)\\ \mathcal{K}^{City}_{\mathcal{I}}(Pitts) &= (name \mapsto \text{“Pittsburg”}, state \mapsto PA)\\ \vdots\quad & \quad\vdots\\ \mathcal{K}^{City}_{\mathcal{I}}(Albany) &\equiv (name \mapsto \text{“Albany”}, state \mapsto NY)\end{aligned}$$

and

$$\begin{aligned}\mathcal{K}^{State}_{\mathcal{I}}(PA) &= \text{“Pennsylvania”}\\ \mathcal{K}^{State}_{\mathcal{I}}(NY) &= \text{“New York”}\end{aligned}$$

■

A key specification is said to be **well-defined** iff for any two instances, $\mathcal{I}$ and $\mathcal{I}'$, if $f^{\mathcal{C}}$ is a family of functions describing an isomorphism from $\mathcal{I}$ to $\mathcal{I}'$, then for each $C \in \mathcal{C}$ and each $o \in \sigma^C$,

$$f^{\kappa^C}(\mathcal{K}^C_{\mathcal{I}}(o)) = \mathcal{K}^C_{\mathcal{I}'}(f^C(o))$$

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. For the remainder 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 $o_1, o_2 \in \sigma^C$, $\mathcal{K}^C_{\mathcal{I}}(o_1) = \mathcal{K}^C_{\mathcal{I}}(o_2)$ if and only if $\mathcal{K}'^C_{\mathcal{I}}(o_1) = \mathcal{K}'^C_{\mathcal{I}}(o_2)$.

Definition 10.2: 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 of example 10.1 has nodes *City* and *State*, and a single edge $(\mathit{City}, \mathit{State})$.

Proposition 10.1: 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). ■

Proof: We proceed by induction on the maximum length of paths in $G(\mathcal{K}^{\mathcal{C}})$. If the maximum length of paths is 0 then each type κ^C, $C \in \mathcal{C}$, is ground and we are done.

Suppose that the maximum length of paths in $G(\mathcal{K}^{\mathcal{C}})$ is $n \geq 1$. Let $\mathcal{C}^* \subseteq \mathcal{C}$ be the set of classes C such that the maximum length of paths in $G(\mathcal{K}^{\mathcal{C}})$ starting at C is 1.

For each $C \in \mathcal{C}^*$ let κ^{*C} be the type formed by replacing C' with $\kappa^{C'}$ in κ^C for each class C' occuring in κ^C. For any instance $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, let $\mathcal{K}^{*C}_{\mathcal{I}} : \sigma^C \to [\![\kappa^{*C}]\!]\mathcal{I}$ be the function such that $\mathcal{K}^{*C}_{\mathcal{I}}(o)$ is formed by replacing each object identity $o' \in \sigma^{C'}$ occuring in $\mathcal{K}^C_{\mathcal{I}}(o)$ by $\mathcal{K}^{C'}_{\mathcal{I}}(o')$ in $\mathcal{K}^C_{\mathcal{I}}(o)$.

For each $C \in (\mathcal{C} \setminus \mathcal{C}^*)$ let $\kappa^{*C} = \kappa^C$ and $\mathcal{K}^{*C}_{\mathcal{I}} = \mathcal{K}^C_{\mathcal{I}}$. Then $\mathcal{K}^{*\mathcal{C}}$ is a key specification which is equivalent to $\mathcal{K}^{\mathcal{C}}$ and the maximum length of a path in $G(\mathcal{K}^{*\mathcal{C}})$ is $n-1$. Hence, by the induction assumption, there is a key specification $\mathcal{K}'^{\mathcal{C}}$, equivalent to $\mathcal{K}^{*\mathcal{C}}$, such that every type in $\mathcal{K}'^{\mathcal{C}}$ is ground. ■

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

Key Correspondences

Definition 10.3: Given a key specification, $\mathcal{K}^{\mathcal{C}}$ and two instances $\mathcal{I}$ and $\mathcal{I}'$, we define the family of relations $\approx^\tau_{\mathcal{K}} \subseteq [\![\tau]\!]\mathcal{I} \times [\![\tau]\!]\mathcal{I}'$ to be the largest relations such that

1. if $c^{\underline{b}} \approx^b_{\mathcal{K}} c'^{\underline{b}}$ for $c, c' \in \mathbf{D}^{\underline{b}}$ then $c \equiv c'$,

2. if $x \approx^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\mathcal{K}} y$ then $x(a_i) \approx^{\tau_i}_{\mathcal{K}} y(a_i)$ for $i = 1, \ldots, k$,

3. if $(a_i, x) \approx^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle}_{\mathcal{K}} (a_j, y)$ then $a_i = a_j$ and $x \approx^{\tau_i}_{\mathcal{K}} y$,

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 $\mathcal{K}_{\mathcal{I}}^{C}(o) \;{}_{\mathcal{I}\mathcal{I}'}{\approx}_{\mathcal{K}}^{\kappa^C}\; \mathcal{K}_{\mathcal{I}'}^{C}(o')$.

We will write ${}_{\mathcal{I}\mathcal{I}'}{\approx}_{\mathcal{K}}$ instead of $\approx_{\mathcal{K}}$ in cases where the instances are not clear from context. ■

Note that for any set of families of relations, $\sim_a^\tau$, $a \in A$ satisfying conditions 1–5 above, the family of unions of the relations, $(\sim^\tau) = \bigcup_{a \in A}(\sim_a^\tau)$, also satisfies conditions 1–5. Hence $\approx_{\mathcal{K}}^{\tau}$ is well-defined.

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.

Given any instance $\mathcal{I}$, we write ${}_{\mathcal{I}}{\approx}_{\mathcal{K}}^{\mathcal{C}}$ for ${}_{\mathcal{I}\mathcal{I}}{\approx}_{\mathcal{K}}^{\mathcal{C}}$, or omit the $\mathcal{I}$ altogether when it is clear from the context. $\approx_{\mathcal{K}}^{\mathcal{C}}$ is called the *correspondence on $\mathcal{I}$ generated by $\mathcal{K}^{\mathcal{C}}$*.

Proposition 10.2: 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 on $[\![\tau]\!]\mathcal{I}$. ■

Proof: It is sufficient to observe that for any family of relations $\sim^\tau \subseteq [\![\tau]\!]\mathcal{I} \times [\![\tau]\!]\mathcal{I}$ satisfying conditions 1–5 of definition 10.3, the family of smallest equivalence relations containing the relations $\sim^\tau$, $\sim^{\tau *}$ say, also satisfy conditions 1–5. The result follows since $\approx_{\mathcal{K}}$ is the largest relation satisfying these conditions. ■

Definition 10.4: 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 $\mathcal{V}^C(o) \approx_{\mathcal{K}}^{\tau^C} \mathcal{V}^C(o')$. ■

Keyed Schema

Definition 10.5: A **keyed schema** is a pair consisting of a schema $\mathcal{S}$ and a key specification $\mathcal{K}^C$ 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 $\mathcal{K}^{\mathcal{C}}$. ■

Given two instances of a simply keyed schema, $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$, say $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and $\mathcal{I}' = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, 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 $\mathcal{V}^C(o) \approx_{\mathcal{K}}^{\tau^C} \mathcal{V}'^C(o')$.

Lemma 10.3: 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 $\approx^{\mathcal{C}}_{\mathcal{K}}$ is a consistent correspondence between $\mathcal{I}$ and $\mathcal{I}'$. ■

Proof: Note that, the extension of the family of binary relations $\approx^{\mathcal{C}}_{\mathcal{K}}$ to a general type τ described in definition 9.1 is equal to the relation $\approx^{\tau}_{\mathcal{K}}$ The result then follows from the definition of $\mathcal{K}$-equivalences of instances. ■

Proposition 10.4: 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 $\mathcal{I} \approx \mathcal{I}'$. ■

Proof: Follows immediately from the definition of $\approx$. ■

Note, however, that the converse is not true: there are simply keyed schema for which the key equivalence is strictly finer than bisimulation of instances, as the following example demonstrates.

Example 10.2: Let us consider the schema from example 7.1 once again, and the key specification, $\mathcal{K}'^{\mathcal{C}}$ given by

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

and

$$\begin{aligned} \mathcal{K}^{City}_{\mathcal{I}}(o) &\equiv (name \mapsto (\mathcal{V}^{City} o).name,\ state\text{-}name \mapsto \mathcal{V}^{State}(\mathcal{V}^{City}(o)(state))(name)) \\ \mathcal{K}^{State}_{\mathcal{I}}(o) &\equiv (name \mapsto \mathcal{V}^{State}(o)(name),\ cities \mapsto \{o' \in \sigma^{City} | \mathcal{V}^{City}(o')(state) = o\}) \end{aligned}$$

So the key of a City is its name and the name of its State, and the key of a state is its name and the set of its Cities.

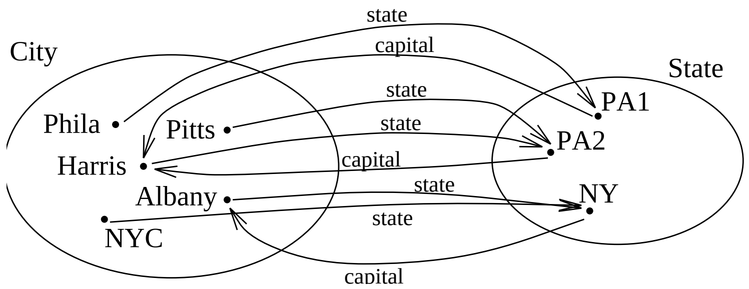


Figure 12: A database instance

Let us now construct a new instance, $\mathcal{I}'$, and compare it to the instance $\mathcal{I}$ defined in example 7.2. $\mathcal{I}'$ is defined by:

$$\begin{aligned} \sigma^{City} &\equiv \{Phila, Pitts, Harris, NYC, Albany\} \\ \sigma^{State} &\equiv \{PA1, PA2, NY\} \end{aligned}$$

and the mappings are

$$\begin{aligned}
\mathcal{V}^{City}(Phila) &\equiv (name \mapsto \text{“Philadelphia”}, state \mapsto PA1)\\
\mathcal{V}^{City}(Pitts) &\equiv (name \mapsto \text{“Pittsburg”}, state \mapsto PA2)\\
\mathcal{V}^{City}(Harris) &\equiv (name \mapsto \text{“Harrisburg”}, state \mapsto PA2)\\
\mathcal{V}^{City}(NYC) &\equiv (name \mapsto \text{“New York City”}, state \mapsto NY)\\
\mathcal{V}^{City}(Albany) &\equiv (name \mapsto \text{“Albany”}, state \mapsto NY)
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{V}^{State}(PA1) &\equiv (name \mapsto \text{“Pennsylvania”}, capital \mapsto Harris)\\
\mathcal{V}^{State}(PA2) &\equiv (name \mapsto \text{“Pennsylvania”}, capital \mapsto Harris)\\
\mathcal{V}^{State}(NY) &\equiv (name \mapsto \text{“New York”}, capital \mapsto Albany)
\end{aligned}$$

This instance is illustrated in figure 12.

Then $\mathcal{I}$ and $\mathcal{I}'$ are bisimilar, $\mathcal{I} \approx \mathcal{I}'$, but then are not equivalent under the key specification $\mathcal{K}'^{\mathcal{C}}$, $\mathcal{I} \not\approx_{\mathcal{K}} \mathcal{I}'$. ■

10.2 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 by

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

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.

Further, we can once again argue that these results also hold for any other reasonable query language supporting keys.

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

Proposition 10.5: 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 $\mathit{SRI}^N(\mathcal{K})$. That is, for each type τ, there is a formula $\mathit{Cor}^{\tau}_{\mathcal{K}}$ of $\mathit{SRI}^N(\mathcal{K})$ such than $\vdash \mathit{Cor}^{\tau}_{\mathcal{K}} : \tau \times \tau \to \mathit{Bool}$ and for any two N-bounded values $u, v \in [\![\tau]\!]\mathcal{I}$, $V[\![\mathit{Cor}^{\tau}_{\mathcal{K}}]\!]\mathcal{I}(u, v) = \mathbf{T}$ iff $u \approx^{\tau}_{\mathcal{K}} v$. ■

Proof: Suppose that k is the maximum length of a path in $G(\mathcal{K})$. For $i = 1, \ldots, k+1$ define $\mathcal{C}_i \subseteq \mathcal{C}$ to be the set of classes $C \in \mathcal{C}$ such that there are no paths in $G(\mathcal{K})$ of length i originating from C. So $\mathcal{C}_{k+1} = \mathcal{C}$, and for each $C \in \mathcal{C}_1$,

κ^C is ground. For convenience we take $\mathcal{C}_0 = \emptyset$. Then for each $C \in \mathcal{C}_i$, $i \geq 1$, every class C' occuring in κ^C is in $\mathcal{C}_{i-1}$.

First we show that, if for every $C \in \mathcal{C}_i$ we have an expression $\mathit{Cor}^{C}_{\mathcal{K}}$, then for any type τ involving only classes from $\mathcal{C}_i$, we can form a formula $\mathit{Cor}^{\tau}_{\mathcal{K}}$. $\mathit{Cor}^{\tau}_{\mathcal{K}}$ is defined by induction on τ. We only present some of the cases, since they are simlar to previous proofs.

1. If $\tau \equiv \underline{b}$ then $\mathit{Cor}^{\tau}_{\mathcal{K}} \equiv (\lambda(x, y) \cdot x =^{\underline{b}} y)$.
2. If $\tau \equiv C$ where $C \in \mathcal{C}_i$ then follows from out assumption.
3. If $\tau \equiv (a_1 : \tau_1, \ldots, a_n : \tau_n)$ then $\mathit{Cor}^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\mathcal{K}} \equiv (\lambda(x, y) \cdot \mathit{Cor}^{\tau_1}_{\mathcal{K}}(x.a_1, y.a_1) \wedge \ldots \wedge \mathit{Cor}^{\tau_k}_{\mathcal{K}}(x.a_1, y.a_1))$

Next we show by induction on i, that for any class $C \in \mathcal{C}_i$, we can form a formula $\mathit{Cor}^{C}_{\mathcal{K}}$. The base case, $i = 0$, is trivial. Suppose we have expressions $\mathit{Cor}^{C'}_{\mathcal{K}}$ for each $C' \in \mathcal{C}_{i-1}$, and $C \in \mathcal{C}_i$. Then we define

$$\mathit{Cor}^{C}_{\mathcal{K}} \equiv (\lambda(x, y) \cdot \mathit{Cor}^{\kappa^C}_{\mathcal{K}}(\mathit{key}^C x, \mathit{key}^C y))$$

Then for any values $u, v \in [\![\tau]\!]\mathcal{I}$, $V[\![\mathit{Cor}^{\tau}_{\mathcal{K}}]\!]\mathcal{I}(u, v) = \mathbf{T}$ iff $u \approx^{\tau}_{\mathcal{K}} v$. ■

Claim: If $(\mathcal{S}, \mathcal{K})$ is a simply keyed schema, and L is some reasonably expressive query language, then for any object type τ it is possible to construct an expression in L which tests whether any two values of type τ are $\mathcal{K}$-equivalent.

Justification: It suffices to convince ourselves that all the constructs and operators used to define the operators $\mathit{Cor}^{\tau}_{\mathcal{K}}$ in the proof of proposition 10.5 are things that one would expect to find in any reasonable query language. In particular we do not need to resort to iterating over the extents of a database.

10.3 A Summary of Observational Equivalence Relations

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 in that system. In section 8.2 we showed that, in a query language supporting an equality test on object identities, two instances are observationally indistinguishable if and only iff they are isomorphic, that is they differ only in their choice of object identities. Further, in such a language, it is straight forward to construct efficient equality tests on values of general data-types. However, although we know that any two non-isomorphic instances are distinguishable, proposition 8.4 tells us that it is not in general possible to find a query which distinguishes between them.

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	$\approx^\tau_\mathcal{K}$ — key correspondence	$\approx_\mathcal{K}$ — key correspondence
SRI($\mathcal{K}$) $\mathcal{K}$ a general key specification	$\approx^\tau_\mathcal{K}$ — 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 13: A summary of the operators considered and the resulting observational equivalences

Equality of object identities does not always coincide with equivalence of the data represented, since databases may contain duplicate data, or may represent the same data using a variety of different structures. For example, when integrating multiple databases, different objects representing the same data may arrise from distinct sources. Consequently we would like to be able to compare database instances and data occuring in an instance, by comparing the observable values in the database only. In section 9 we showed that, in a resonable query language without support for any direct comparisons of object identities, observational distinguishability of databases coincides with the bisimulation relation $\approx$, and, if a query language provides the ability to iterate over the entire extents of a database, it is possible to test whether any two values in a database are bisimilar.

In sections 8 and 9 we argued that isomorphism and bisimulation were respectively the *finest* and the *coarsest* equivalence relations on instances that one might hope to observe. In this section we showed that systems of *keys* generate various observational equivalences lying between these two. Further acyclic key specifications provide us with an efficient means of comparing and referencing recursive values which incorporate object identities, without having to examine the object identities directly. These results are summarized in figure 13.

11 A Data Model Based on Regular Trees

The concept of object-identities provides a useful abstraction of the reference mechanisms used in representing complex or recursive data-structures in a database. However such reference mechanisms are normally internal to a database system, and may not be directly accessed or observed by a user. In particular object identities do not represent part of the data being modeled in a database, and the data being modeled does not depend on the choice of object identities used. Consequently we would like to deal with data-models where object-identities are not considered to be directly visible.

We would like to construct a data-model which coincides precisely with the observable properties of a database: two instances or values in an instance should be equal in the model precisely when they can not be distinguished by any query in some underlying query language. Such a model would give us insight into the expressive power and richness of a database system, which an overly fine model of instances, such as that introduced in definition 7.3, fails to capture.

One approach to this problem is to develop an *observational equivalence* relation on instances, representing when two instances are observationally indistinguishable, and to consider equivalence classes of instances rather than individual instances. This approach was investigated in sections 8, 9 and 10 for various different assumptions about the available query language. However this approach still makes it difficult to divorce the information represented by an instance from the mechanisms used to represent the information, and does not provide us with a clear view of the observable information captured by a specific database.

In this section we will present a model sharing definition 7.1 of types and schemas, but in which instances are based on the idea that the only observable values are those of base-type (integers, strings and so on), and those constructed from other observable values using set, record and variant constructors. Such a value-based model was proposed in [2].

The model will use *regular trees* in order to represent values and instances. The idea of regular trees is that they capture those infinite trees with *finite* representations, and consequently can be thought of as finite trees with cycles ([15]). Though it is fairly easy to form an intuitive understanding of regular trees, based on diagrammatic representations, to formulate them rigorously requires a surprising amount of care. This is in part due to the use of *sets* in our data-model, and consequently having to deal with trees in which the branches may not have distinct labels. It's suggested that readers who feel comfortable with the concept of regular trees skip the technical details in the next couple of sub-sections.

11.1 Regular Trees

Definition 11.1: A *tree domain*, $D \subseteq \mathbb{N}^*$, is a set of non-empty strings of natural numbers, such that

1. if $\phi i \in D$, $i > 1$, then $\phi(i-1) \in D$, and

2. if $\phi i \in D$, ϕ a non-empty string, then $\phi \in D$.

Suppose $\mathbf{A}$ is some set. A *tree* over $\mathbf{A}$ is a function from some tree domain, D, to $\mathbf{A}$. ■

We write $\mathit{Tree}(\mathbf{A})$, for the set of trees over $\mathbf{A}$, and say that $\mathbf{A}$ is the *alphabet* of the trees in $\mathit{Tree}(\mathbf{A})$.

Suppose $t \in \mathit{Tree}(\mathbf{A})$ is a tree and $\phi \in \mathbb{N}^*$ is a string such that $\phi \in \mathit{dom}(t)$. We write $t.\phi$ for the tree

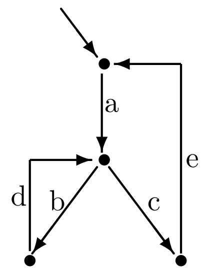
$$t.\phi(x) \equiv t(\phi x)$$

We say $t.\phi$ is the *projection* of t on ϕ.

Given two trees, t and t', we say t' is a *subtree* of t iff there is a $\phi \in \mathit{dom}(t)$ such that $t' = t.\phi$.

Definition 11.2: A **regular tree** is a tree with only finitely many subtrees. ■

There several ways of constructing regular trees: one can view them as directed graphs with edges marked by elements of the alphabet, together with some distinguished node, known as the *root*, such that there is a directed path from the root to each node in the graph. An example of such a graph is shown below:



Alternatively we can view a regular tree as being defined by a system of equations over string variables and patterns:

$$\begin{array}{rcl} T_1 & = & a.T_2 \\ T_2 & = & b.T_3 | c.T_4 \\ T_3 & = & d.T_2 \\ T_4 & = & e.T_3 \end{array}$$

In this case the previous regular tree is represented by the string variable T_1. The operator $|$ represents *choice*: either $T_2 = b.T_3$ or $T_2 = c.T_4$.

There are a number of other equivalent representations as well. In general we will use directed graphs in order to give an intuitive representation of regular trees. See [15] for further details.

Bisimulation of Regular Trees

We do not want to distinguish between two regular trees if they differ only in the ordering or multiplicity of their edges. We therefore need to construct an equivalence relation on trees which captures when we consider two regular trees to be the same.

We define the binary relation $\asymp$ on regular trees over $\mathbf{A}$ to be the largest relation such that, if $t_1 \asymp t_2$ then

1. if $i \in \mathit{dom}(t_1)$ (i a string of length 1), then there is a $j \in \mathit{dom}(t_2)$ such that $t_1(i) = t_2(j)$ and $t_1.i \asymp t_2.j$, and
2. if $j \in \mathit{dom}(t_2)$ then there is an $i \in \mathit{dom}(t_1)$ such that $t_1(i) = t_2(j)$ and $t_1.i \asymp t_2.j$.

If $t_1 \asymp t_2$ we say that t_1 and t_2 are *bisimilar* and we will be treating them as the same tree.

In some sense, the need to equate regular trees which are bisimilar here is bought about because we are dealing with data-models based on sets: if we considered regular trees to be distinct when they differed in the ordering or multiplicity of edges we would arrive at a similar data-model based on lists instead of sets. As such, bisimulation may be thought of as encoding the properties of finite sets at this point.

Constructors for Trees

Suppose $t_1, \ldots, t_k$ are trees, and $\alpha_1, \ldots, \alpha_k$ are elements of the alphabet. Then $\langle \alpha_1 t_1, \ldots, \alpha_k t_k \rangle^T$ is the tree given by

$$\mathit{dom}(\langle \alpha_1 t_1, \ldots, \alpha_k t_k \rangle^T) \equiv \bigcup_{i=1}^{k} \{ i\phi \mid \phi \in \mathit{dom}(t_i) \} \cup \bigcup_{i=1}^{k} \{ i \}$$

and

$$\langle \alpha_1 t_1, \ldots, \alpha_k t_k \rangle^T (i) \equiv \alpha_i$$

for $i = 1, \ldots, k$, and

$$\langle \alpha_1 t_1, \ldots, \alpha_k t_k \rangle^T (i\phi) \equiv t_i(\phi)$$

for $i = 1, \ldots, k$ and $\phi \in \mathit{dom}(t_i)$.

We write $\langle \alpha_1 t_1, \ldots, \alpha_k t_k \rangle$ for the $\asymp$-equivalence class containing $\langle \alpha_1 t'_1, \ldots, \alpha_k t'_k \rangle^T$, where t'_i is a representative of the $\asymp$-equivalence class t_i, for $i = 1, \ldots, k$. We write ϵ for the $\asymp$-class consisting of the tree with empty domain. For the remainder we will refer to these equivalence classes as regular trees, and, when dealing with regular trees, will consider them to be representatives of their $\asymp$-equivalence classes.

11.2 Trees of Types

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

We will be interested in regular trees over an alphabet with the elements

1. the symbol $\dot{\in}$,
2. for each attribute label $a \in \mathcal{A}$, the symbols π_a and ins_a,
3. for each $C \in \mathcal{C}$ a symbol St^C, and
4. for each base type $\underline{b} \in \mathcal{B}$ and each $c \in \mathbf{D}^{\underline{b}}$ a *constant symbol* $c^{\underline{b}}$.

Definition 11.3: We define the mapping $TTree_{\mathcal{S}}$ from types to sets of regular trees over this alphabet to be the largest such that:

1. if $t \in TTree_{\mathcal{S}}(\underline{b})$ then $t = \langle c^{\underline{b}} \epsilon \rangle$ for some constant symbol $c^{\underline{b}} \in Const(\underline{b})$,
2. if $t \in TTree_{\mathcal{S}}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ then $t = \langle \pi_{a_1} t_1, \ldots, \pi_{a_k} t_k \rangle$ where $t_i \in TTree_{\mathcal{S}}(\tau_i)$ for $i = 1, \ldots, k$.
3. if $t \in TTree_{\mathcal{S}}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ then $t = \langle ins_{a_i}\ t' \rangle$ where $t' \in TTree_{\mathcal{S}}(\tau_i)$ for some $i \in 1, \ldots, k$.
4. if $t \in TTree_{\mathcal{S}}(\{\tau\})$ then $t = \langle \dot{\in} t_1, \ldots, \dot{\in} t_k \rangle$ where $k \geq 0$ and $t_i \in TTree_{\mathcal{S}}(\tau)$ for $i = 1, \ldots, k$.
5. if $t \in TTree_{\mathcal{S}}(C)$ then $t = \langle \mathcal{S}^C_{Tgt} t' \rangle$ where $t' \in TTree_{\mathcal{S}}(\tau^C)$.

■

$TTree_{\mathcal{S}}(\tau)$ represents the set of all regular trees of type τ for the schema $\mathcal{S}$.

Informally these definitions can be interpreted as:

1. A tree of base type has one branch, labeled by a constant symbol, which goes to the empty tree;
2. A tree of record type $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ has k branches, labeled π_{a_1} to π_{a_k}, going to trees of types τ_1 to τ_n respectively;
3. A tree of variant type $\langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle$ has one branch, labeled ins_{a_i} for some i, going to a tree of type τ_i;
4. A tree of set type $\{\tau\}$ has finitely many branches, each labeled by $\dot{\in}$ and each going to a tree of type τ; and
5. A tree of class type C has one branch, labeled by $\mathcal{S}^C_{Tgt}$, going to a tree of type τ^C.

11.3 Instances

Definition 11.4: A (regular tree) **instance**, $\omega^{\mathcal{C}}$, of a schema $\mathcal{S}$ consists of a family of finite sets of regular trees, $\omega^{C} \subseteq \mathit{TTree}_{\mathcal{S}}(C)$, for each $C \in \mathcal{C}$.

A regular tree, t, is said to be a **value of type** τ iff $t \in \mathit{TTree}_{\mathcal{S}}(\tau)$.

A regular tree t is said to be **valid** for an instance $\omega^{\mathcal{C}}$ iff for each $C \in \mathcal{C}$ and each subtree t' of t of type C, $t' \in \omega^{C}$.

An instance $\omega^{\mathcal{C}}$ is said to be **valid** iff for each $C \in \mathcal{C}$ and each $t \in \omega^{C}$, t is valid for $\omega^{\mathcal{C}}$.

We write $\mathit{RInst}(\mathcal{S})$ for the set of valid regular tree instances of a schema $\mathcal{S}$. ■

Example 11.1: Let us consider an instance for the schema described in example 7.1. The

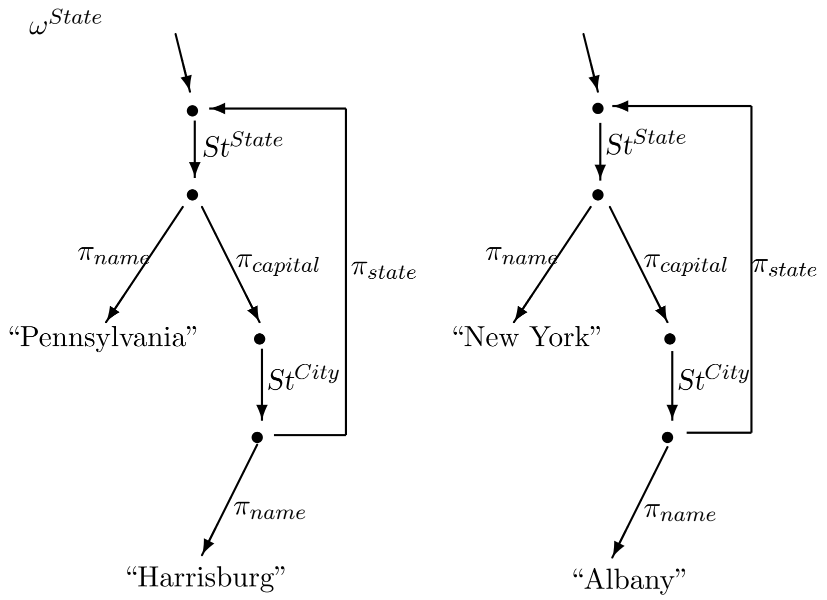


Figure 14: States from instance

instance consists of two sets, ω^{State} and ω^{City}. The set ω^{State} contains the regular trees shown in figure 14 representing the states *New York* and *Pennsylvania*. Pennsylvania has the string "Pennsylvania" as its name, and a tree representing the city *Harrisburg* as its capital, while New York has the string "New York" as its name, and a tree representing the city *Albany* as its capital. The tree representing Harrisburg in turn has the string "Harrisburg" as its name, and the tree representing Pennsylvania as its state. Note that there is a loop in the tree at this point: in fact this is a finite representation of an infinite regular tree. The set ω^{City} will also contain a number of regular trees, such as the one shown in figure 15 representing the city *Philadelphia*. However, in order to be a valid instance, ω^{City} must at least contain regular trees

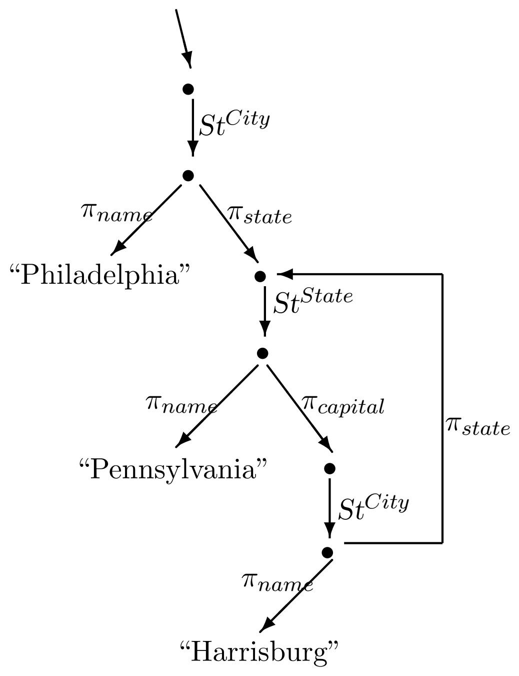


Figure 15: Cities from instance

for Harrisburg and Albany. ∎

11.4 Mapping Between Regular Tree and Object-Identity Based Models

In this section we will show that there is a one-to-one correspondence between the $\approx$-equivalence classes of the object-identity based instances of a schema of definition 7.3 and the regular tree based instances of definition 11.4.

Mapping from Object-Identities to Regular Trees

Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance of schema $\mathcal{S}$. For each type τ we define the mapping $tree^{\tau}_{\mathcal{I}}$ from $[\![\tau]\!]\sigma^{\mathcal{C}}$ to regular trees, by:

$$\begin{aligned}
tree^{b}_{\mathcal{I}}(c) &\equiv \langle c^{b}\epsilon\rangle \\
tree^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\mathcal{I}}(x) &\equiv \langle \pi_{a_1} tree^{\tau_1}_{\mathcal{I}}(x(a_1)),\ldots,\pi_{a_k} tree^{\tau_{a_k}}_{\mathcal{I}}(x(a_k))\rangle \\
tree^{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle}_{\mathcal{I}}(a_i,x) &\equiv \langle ins_{a_i}\, tree^{\tau_i}_{\mathcal{I}}(x)\rangle \\
tree^{\{\tau\}}_{\mathcal{I}}(\{x_1,\ldots,x_k\}) &\equiv \langle \dot{\in} tree^{\tau}_{\mathcal{I}}(x_1),\ldots,\dot{\in} tree^{\tau}_{\mathcal{I}}(x_n)\rangle \\
tree^{C}_{\mathcal{I}}(o) &\equiv \langle St^{C}\, tree^{\tau^{C}}_{\mathcal{I}}(\mathcal{V}^{C}(o))\rangle
\end{aligned}$$

Lemma 11.1: For any type τ if $v \in [\![\tau]\!]\sigma^{\mathcal{C}}$ then $tree^{\tau}_{\mathcal{I}}(v) \in TTree_{\mathcal{S}}(\tau)$. ■

Proof: For each type τ define the set of regular trees T^{τ} by

$$\begin{array}{rcl} T^{C} & \equiv & \{tree^{C}_{\mathcal{I}}(o) \mid o \in \sigma^{C}\} \\ T^{\underline{b}} & \equiv & \{\langle c^{\underline{b}}\epsilon\rangle \mid c \in \mathbf{D}^{\underline{b}}\} \\ T^{(a_1:\tau_1,\ldots,a_k:\tau_k)} & \equiv & \{\langle \pi_{a_1}t_1,\ldots,\pi_{a_k}t_k\rangle \mid t_i \in T^{\tau_i} \text{ for } i=1,\ldots,k\} \\ T^{\langle\!|a_1:\tau_1,\ldots,a_k:\tau_k|\!\rangle} & \equiv & \{\langle ins_{a_i}t_i\rangle \mid i \in 1,\ldots,k \text{ and } t_i \in T^{\tau_i}\} \\ T^{\{\tau\}} & \equiv & \{\langle \dot{\in}t_1,\ldots,\dot{\in}t_n\rangle \mid t_i \in T^{\tau} \text{ for } i=1,\ldots,n,\ n \in \mathbb{N}\} \end{array}$$

Then, for any $x \in [\![\tau]\!]\mathcal{I}$, $tree^{\tau}_{\mathcal{I}}(x) \in T^{\tau}$. Further the family of sets T^{τ} satisfy conditions 1–5 of definition 11.3, and so $T^{\tau} \subseteq TTree_{\mathcal{S}}(\tau)$ for every type τ. Hence result. ■

The mapping *tree* from object-identity based instances, *Inst*($\mathcal{S}$), to regular tree based instances, *RInst*($\mathcal{S}$), is given by

$$tree : \mathcal{I} \mapsto \omega^{\mathcal{C}}$$

where

$$\omega^{C} \equiv \{tree^{C}_{\mathcal{I}}(o) \mid o \in \sigma^{C}\}$$

for $C \in \mathcal{C}$.

From lemma 11.1 we can see that *tree*($\mathcal{I}$) is indeed a valid regular-tree instance of $\mathcal{S}$ for any $\mathcal{I} \in$ *Inst*($\mathcal{S}$).

Mapping from Regular Trees to Object-Identities

Suppose $\omega^{\mathcal{C}}$ is a regular tree instance of a schema $\mathcal{S}$.

Assume we have some ordered, countably infinite set from which to pick object identifiers.[5] For each $C \in \mathcal{C}$, and each $\nu \in \omega^{C}$, pick an object identifier o^{ν} to associate with ν. (If we use a lexicographical ordering on the elements of $\omega^{\mathcal{C}}$ and the ordering on our set of potential object-identifiers then this can be done in a deterministic way).

We can form a family of object identifier sets, $\sigma^{\mathcal{C}}$ by

$$\sigma^{C} \equiv \{o^{\nu} \mid \nu \in \omega^{C}\}$$

Then for each type τ we define the mapping $inst^{\tau}_{\omega^{\mathcal{C}}}$ from $TTree_{\mathcal{S}}(\tau)$ to values from $[\![\tau]\!]\sigma^{\mathcal{C}}$ by

$$\begin{array}{rcl} inst^{\underline{b}}_{\omega^{\mathcal{C}}}(\langle c^{\underline{b}}\epsilon\rangle) & \equiv & c \\ inst^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\omega^{\mathcal{C}}}(\langle \pi_{a_1}\nu_1,\ldots,\pi a_k\nu_k\rangle) & \equiv & (a_1 \mapsto inst^{\tau_1}_{\omega^{\mathcal{C}}}(\nu_1),\ldots,a_k \mapsto inst^{\tau_k}_{\omega^{\mathcal{C}}}(\nu_k)) \\ inst^{\langle\!|a_1:\tau_1,\ldots,a_k:\tau_k|\!\rangle}_{\omega^{\mathcal{C}}}(\langle ins_{a_i}\nu\rangle) & \equiv & (a_i, inst^{\tau_i}_{\omega^{\mathcal{C}}}(\nu)) \\ inst^{\{\tau\}}_{\omega^{\mathcal{C}}}(\langle \dot{\in}\nu_1,\ldots,\dot{\in}\nu_k\rangle) & \equiv & \{inst^{\tau}_{\omega^{\mathcal{C}}}(\nu_i) \mid i=1,\ldots,k\} \\ inst^{C}_{\omega^{\mathcal{C}}}(\nu) & \equiv & o^{\nu} \end{array}$$

[5] a gumball machine

For each $C \in \mathcal{C}$ define the mapping $\mathcal{V}^C : \sigma^C \to [\![\tau^C]\!]\sigma^{\mathcal{C}}$ by

$$\mathcal{V}^C(o^\nu) \equiv \mathit{inst}^{\tau^C}_{\omega^{\mathcal{C}}}(\nu') \quad \text{where } \nu = \langle St^C \nu' \rangle$$

For any regular tree instance, $\omega^{\mathcal{C}}$, we define the object-identity instance $\mathit{inst}'(\omega^{\mathcal{C}})$ by

$$\mathit{inst}'(\omega^{\mathcal{C}}) \equiv (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$$

where $\sigma^{\mathcal{C}}$ and $\mathcal{V}^{\mathcal{C}}$ are as described above.

The mapping *inst* from regular tree instances $\mathit{RInst}(\mathcal{S})$ to equivalence classes of object-identity instances, $\mathit{Inst}(\mathcal{S})/\approx$ is given by

$$\mathit{inst}(\omega^{\mathcal{C}}) \equiv [\mathit{inst}'(\omega^{\mathcal{C}})]_{\approx}$$

Lemma 11.2: If $\mathcal{I} \equiv (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an object-identity instance, $\mathcal{I} \in \mathit{Inst}(\mathcal{S})$, then $\mathit{inst}'(\mathit{tree}(\mathcal{I})) \approx \mathcal{I}$. ■

Proof: Define $\sim^{\mathcal{C}}$ to be the smallest correspondence relation between $\mathit{inst}'(\mathit{tree}(\mathcal{I}))$ and $\mathcal{I}$ such that $\mathit{inst}^C(\mathit{tree}^C_{\mathcal{I}}(o)) \sim^C o$ for each $C \in \mathcal{C}$, $o \in \sigma^C$.

Suppose $\mathit{tree}(\mathcal{I}) = \omega^{\mathcal{C}}$ and $\mathcal{I}' = \mathit{inst}'(\omega^{\mathcal{C}})$. We prove that, for any type τ and $u \in [\![\tau]\!]\sigma^{\mathcal{C}}$, $\mathit{inst}^\tau_{\omega^{\mathcal{C}}}(\mathit{tree}^\tau_{\mathcal{I}}(u)) \sim^\tau u$. We will present some sample induction cases here:

1. If $\tau \equiv \underline{b}$, a base type, $c \in \mathbf{D}^{\underline{b}}$, then

$$\begin{aligned} \mathit{inst}^{\underline{b}}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\underline{b}}_{\mathcal{I}}(c)) &= \mathit{inst}^{\underline{b}}_{\omega^{\mathcal{C}}}(\langle c^{\underline{b}} \epsilon \rangle) \\ &= c \end{aligned}$$

2. If $\tau \equiv (a_1 : \tau_1, \ldots, a_n : \tau_n)$ and $u \in [\![(a_1 : \tau_1, \ldots, a_n : \tau_n)]\!]\sigma^{\mathcal{C}}$, then

$$\begin{aligned} \mathit{inst}^\tau_{\omega^{\mathcal{C}}}(\mathit{tree}^\tau_{\mathcal{I}}(u)) &= \mathit{inst}^\tau_{\omega^{\mathcal{C}}}(\langle \pi_{a_1} \mathit{tree}^{\tau_1}_{\mathcal{I}}(u(a_1)), \ldots, \pi_{a_n} \mathit{tree}^{\tau_n}_{\mathcal{I}}(u(a_n)) \rangle) \\ &= (a_1 \mapsto \mathit{inst}^{\tau_1}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\tau_1}_{\mathcal{I}}(u(a_1))), \ldots, a_n \mapsto \mathit{inst}^{\tau_n}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\tau_n}_{\mathcal{I}}(u(a_n)))) \\ &\sim^\tau (a_1 \mapsto u(a_1), \ldots, a_n \mapsto u(a_n)) \\ &= u \end{aligned}$$

3. If $\tau \equiv \{\tau'\}$ and $u \in [\![\{\tau'\}]\!]\sigma^{\mathcal{C}}$, say $u = \{u_1, \ldots, u_k\}$, then

$$\begin{aligned} \mathit{inst}^\tau_{\omega^{\mathcal{C}}}(\mathit{tree}^\tau_{\mathcal{I}}(u)) &= \mathit{inst}^\tau_{\omega^{\mathcal{C}}}(\langle \in \mathit{tree}^{\tau'}_{\mathcal{I}}(u_1), \ldots, \in \mathit{tree}^{\tau'}_{\mathcal{I}}(u_k) \rangle) \\ &= \{\mathit{inst}^{\tau'}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\tau'}_{\mathcal{I}}(u_1), \ldots, \mathit{inst}^{\tau'}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\tau'}_{\mathcal{I}}(u_n)\} \\ &\sim^\tau \{u_1, \ldots, u_n\} \\ &= u \end{aligned}$$

Suppose $C \in \mathcal{C}$ and $o \in \sigma^C$. Then $\mathit{tree}^C_{\mathcal{I}}(o) = \langle \mathcal{S}^C_{\mathrm{Tgt}} \mathit{tree}^{\tau^C}_{\mathcal{I}}(\mathcal{V}^C(o))\rangle$.

Hence $\mathcal{V}'^C(\mathit{inst}^C_{\omega^{\mathcal{C}}}(\mathit{tree}^C_{\mathcal{I}}(o))) = \mathit{inst}^{\tau^C}_{\omega^{\mathcal{C}}}(\mathit{tree}^{\tau^C}_{\mathcal{I}}(\mathcal{V}^C(o)))$.

Hence $\mathcal{V}^C(o) \sim^{\tau^C} \mathcal{V}'^C(\mathit{inst}^C_{\omega^{\mathcal{C}}}(\mathit{tree}^C_{\mathcal{I}}(o)))$.

Hence $\sim^{\mathcal{C}}$ is a consistent correspondence, and $\mathcal{I} \approx \mathcal{I}'$. ■

Lemma 11.3: If $\omega^{\mathcal{C}} \in \mathit{RInst}(\mathcal{S})$ is a regular tree instance, and $\mathcal{I} = \mathit{inst}'(\omega^{\mathcal{C}})$, then $\omega^{\mathcal{C}} = \mathit{tree}(\mathit{inst}'(\omega^{\mathcal{C}}))$. ■

Proof: Suppose $\mathcal{I} = \mathit{inst}'(\omega^{\mathcal{C}})$. We will show that, for any type τ and $\nu \in \mathit{TTree}_{\mathcal{S}}(\tau)$ such that ν is valid for $\omega^{\mathcal{C}}$, $\mathit{tree}^{\tau}_{\mathcal{I}}(\mathit{oinstO}^{\tau}(\nu)) = \nu$.

Consider the relations R defined by

$$R^{\tau} \equiv \{(\mathit{tree}^{\tau}_{\mathcal{I}}(\mathit{inst}^{\tau}_{\omega^{\mathcal{C}}}(\nu)), \nu) | \nu \in \mathit{TTree}_{\mathcal{S}}(\tau),\ \tau \in \mathit{Types}(\mathcal{S}),\ \nu \text{ valid for } \omega^{\mathcal{C}}\}$$

for any type τ. From the definition of regular trees (section 11.1) it is enough to show that, for any $(\nu', \nu) \in R$, if $i \in \mathit{dom}\nu'$ then there is a $j \in \mathit{dom}(\nu)$ such that $\nu'(i) = \nu(j)$ and $(\nu'.i, \nu.j) \in R$, and for any $j \in \mathit{dom}(\nu)$ there is an $i \in \mathit{dom}(\nu')$ such that $\nu'(i) = \nu(j)$ and $(\nu'.i, \nu.j) \in R$. Proof proceeds by induction on types and is similar to the proof of lemma 11.2. ■

Proposition 11.4: The mappings *tree* and *inst* provide an isomorphism between the set of bisimulation classes of object-identity instances, $\mathit{Inst}(\mathcal{S})/\approx$, and the set of regular tree instances, $\mathit{RInst}(\mathcal{S})$. ■

Proof: Follows immediately from lemmas 11.2 and 11.3. ■

Part III

The *WOL* Language for Database Transformations and Constraints

12 Introduction

In part I we observed that database transformations arise from a wide variety of tasks in database and information management, and that there is a need for tools and methodologies to aid in implementing such transformations. In section 4 we argued that such tools need to be based on data-models supporting complex data-structures and some kind of reference mechanism, such as object-identities, to allow for the representation of recursive or arbitrarily deeply nested data-structures.

In part II we examined the issue of the information capacity of data-models involving object-identities, and claimed that a precise understanding of information capacity was necessary in order to be able to reason about the correctness or equivalence of database transformations, and hence the semantics of formalisms for expressing database transformations. We showed that the information capacity of a data-model is dependent on the constructs available for querying the data-model, and in particular on the predicates available for comparing object-identities. We introduced *keys* as a mechanism for generating or comparing object identities, and argued that acyclic systems of keys provided an efficient and flexible mechanism for dealing with object identities.

WOL (Well-founded Object Language) is a declarative language for specifying and implementing database transformations and constraints. It is based on the data-model of section 7, assuming acyclic systems of keys as in section 10, and can therefore deal with databases involving object-identity and recursive data-structures as well as complex and arbitrarily nested data-structures.

WOL bears a superficial resemblence to the *F-logic* of Kifer and Lausen [28], in that both are logic-based languages designed for reasoning about database schemas and instances involving various object-oriented concepts. However F-logic differs from *WOL* in blurring the distinctions between objects and schema-classes, and between objects and attributes, and by directly incorporating a notion of inheritence or subsumption. In doing so, F-logic aims to allow reasoning about normally "second-order" concepts, such as inheritence and methods. In the design of *WOL* there is a clear distinction between schemas and instances, and between attributes and classes, since we felt that these were conceptually distinct, and that coalescing them into a single concept would be confusing and unworkable. *WOL* sets itself the less ambitious task of allowing the specification of and reasoning about instance-level properties of a database, such as constraints, at the schema level. More significantly, *WOL* aims to be a practical language allowing the efficient implementation of a significant class of transformation specifications and constraints. As far as the author is aware, there are no practical implementations or effiecient algorithms for constraints, transformations or methods based on F-logic.

In section 5 we argued that there are important interactions between transformations and the constraints imposed on databases: constraints can play a part in determining a transformation, and also transformations can imply constraints on their source and target databases. Although most data-models support some specific kinds of constraints, in general these are rather ad hoc collections, included because of their utility in the particular examples that the designer of the system had in mind rather than on any sound theoretical basis. For example, relational databases will often support keys and sometimes functional and inclusion dependencies [45], while semantic models might incorporate various kinds of cardinality constraints and inheritance [21, 25]. The constraints that occur when dealing with transformations often fall outside such predetermined classes; further it is difficult to anticipate the kinds of constraints that will arise. *WOL* addresses this problem by augmenting the data-model with a general formalism for expressing constraints, which allows one to reason about the interaction between transformations and constraints.

Example 12.1: For example, in our Cities and States database of example 7.1, we would want to impose a constraint that the capital City of a State is in the State of which it is the capital. We can express this as

$$X.\mathit{state} = Y \Longleftarrow Y \in \mathit{State}_A, X = Y.\mathit{capital}$$

This can be read as "if Y is in class *State* and X is the *capital* of Y, then Y is the *state* of X". Suppose also that our States and Cities each had an attribute *population* and we wanted to impose a constraint that the population of a City was less than the population of the State in which it resides. We could express this as

$$X.\mathit{population} < Y.\mathit{population} \Longleftarrow X \in \mathit{City}_A,\ Y = X.\mathit{state};$$

Such a constraint cannot be expressed in the constraint languages associated with most data models.

We can also use constraints to express how the keys of a schema are derived:

$$\begin{aligned} X = \mathit{Mk}^{\mathit{City}_A}(\mathit{name} = N,\ \mathit{state\_name} = S) \ \Longleftarrow \\ X \in \mathit{City}_A,\ N = X.\mathit{name},\ S = X.\mathit{state}.\mathit{name}. \end{aligned}$$

This constraint says that the key of an object of class *City* is a tuple built out of the *name* of the city, and the *name* of its *state*. Such constraints are important in allowing us to identify objects in transformations. ■

WOL is based on Horn clause logic expressions, using a small number of simple predicates and primitive constructors. However it is sufficient to express a large family of constraints including those commonly found in established data-models. In fact the only kinds of constraints which occur in established data-models but can not easily be expressed in *WOL* are finite cardinality constraints: these are constraints that might state, for example, that a certain set-valued attribute has cardinality between 2 and 3, although it wouldn't be difficult in practice to extend *WOL* with operators to express such constraints.

The language *WOL* can also be used to express constraints that span multiple databases, and, in particular, can be used to specify transformations. A *transformation specification* may viewed as a collection of constraints stating how data in a target database arises from data stored in a number of source databases. In general however there may be any number of target database instances satisfying a particular set of constraints for a particular collection of source instances. It is therefore necessary to restrict our attention to *complete* transformation specifications, such that for any collection of source database instances if there is a target instance satisfying the transformation specification then there is a *unique smallest* such target instance.

Possibly the closest existing work to *WOL* are the structural manipulations of Abiteboul and Hull [1] illustrated in example 3. The rewrite rules in [1] have a similar feel to the Horn clauses of *WOL* but are based on pattern matching against complex data-structures, allowing for arbitrarily nested set, record and variant type constructors. *WOL* gains some expressivity over the language of [1] by the inclusion of more general and varied predicates (such as not-equal and not-in), though we have not included tests for cardinality of sets in *WOL*. The main contributions of *WOL* however lie in its ability to deal with object-identity and hence recursive data-structures, and in the uniform treatment of transformation rules and constraints.

The language of [1] allows nested rewrite rules which can generate more general types of nested sets, whereas in *WOL* we require that any set occurring in an instance is identifiable by some means external to the elements of the set itself. Comparing the expressive power of the two formalisms is difficult because of the difference between the underlying models, and because the expressive power of each language depends on the predicates incorporated in the language. However if the rewrite rules of [1] are extended to deal with the data-model presented here, and both languages are adjusted to support equivalent predicates (for example adding inequality and not-in tests to [1] and cardinality tests to *WOL*) then *WOL* can be shown to be at least as expressive as the rules of [1].

12.1 Implementation of *WOL*

Formally specifying a database transformation is a worthwhile task in itself, since it increases our understanding and confidence in the correctness of the transformation. However the *WOL* language also allows us to directly implement an important class of transformation programs. Since we are concerned with structural transformations which can be performed efficiently on large quantities of data, rather than general computations, we need to restrict those transformation specifications that may be used in order to ensure that the transformation procedure will terminate and can be performed in a single pass of the source database. In section 14 we will define a class of *non-recursive* transformation programs, and describe an algorithm for converting such transformation programs written in *WOL* into a *normal form* which can then be translated to an underlying DBPL for implementation.

There are a number of advantages in using the language *WOL* to program database transformations and constraints, beyond the ability to formally reason about them: *WOL* transformation programs are easy to modify and maintain, for example in order to reflect schema evolutions; and the declarative nature of *WOL* means that the conceptually separate parts of a transformation can be specified independently, and the transformation program formed by collecting together the relevant clauses.

12.2 A Roadmap to Part III

In section 13 we will define the syntax and semantics of the *WOL* language. We will present typing rules for *WOL* terms and atoms, define notions of well-typing and range-restriction of *WOL* clauses, and define a well-formed clause to be one which is both well-typed and range-restricted. We will also define syntacticly restricted form of *WOL* clauses called *semi-normal form* clauses, which will be used extensively in later sections. Examples will be used to illustrate various features of *WOL*, and to show how it can be used to express a wide variety of database constraints.

In section 14 we will show how *WOL* can be used for specifying database transformations. First, in sections 14.1 and 14.2 we will show how *WOL* can be used to express constraints relating multiple databases by *partitioning* database schemas, and how such multi-database constraints can be used for specifying a database transformation. In section 14.3 we will show that such collections of *transformation clauses* may not uniquely and unambiguously determine a meaningful transformation of a source database, and will introduce the notions of *deterministic* and *complete* transformation programs which do uniquely determine transformations. In section 14.4 we will define *normal-form* transformation clauses, which uniquely determine a complete object in a target database in terms of source databases only, and which can easily be implemented in a variety of database programming languages. We also describe restrictions on normal form clauses, using the notion of *characterizing formula*, which ensure that they define a complete transformation program. I have tried to make the notion of characterizing formula described in section 14.4 as general as possible, with the result that these ideas are rather complicated and difficult to check. In section 14.5 we present a simplified version of the work on normal-form clauses and characterizing formula based on a more restrictive scenario. It is suggested that a reader interested in practical implementations of *WOL* merely skims the presentation of characterizing formulae presented in section 14.4 and instead concentrates on the simplified versions in section 14.5. In sections 14.6 and 14.7 we show how *non-recursive* complete transformation programs can be unfolded into equivalent normal-form programs, and thus implemented.

In section 15 we show how *WOL* can be extended in order to specify transformations in a language supporting alternative collection types such as *bags* or *lists*. Section 15 is in some sense orthogonal to the other sections of this part, and could be skipped without detracting from the overall understanding of this work.

13 The Syntax and Semantics of *WOL*

In this section we will give a rigorous definition of the language *WOL* introduced informally in previous sections. In section 13.1 we will define the syntax for *WOL*, and in section 13.2 we will give a denotational semantics for *WOL* based on the model of section 7.

13.1 Syntax

We will assume a simply keyed schema, $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$, with classes $\mathcal{C}$, and will define our language, $\mathit{WOL}^{\mathcal{S}\mathcal{K}}$, relative to this schema. We will frequently simply write *WOL* for this language and leave the schema implicit.

As before we will assume a countable set of constant symbols ranged over by $c^{\underline{b}}$ for each base type $\underline{b} \in \mathcal{B}$, and also a countably infinite set of variables, *Var*, ranged over by $X, Y, \ldots$.

We may also assume some additional predicate symbols, ranged over by $p^{\underline{b}_1 \cdots \underline{b}_r}, \ldots$. $p^{\underline{b}_1 \cdots \underline{b}_r}$ is a predicate symbol of arity r, taking arguments of types $\underline{b}_1, \ldots, \underline{b}_r$. In general, when we use additional predicate symbols, they will represent well established predicates, such as $\leq$ on integers, and may make use of infix notations in order to give a more standard appearance.

Terms

Definition 13.1: The set of **terms**

for $(\mathcal{S}, \mathcal{K})$, $\mathit{Terms}^{\mathcal{S}}$, ranged over by $P, Q, \ldots$, is given by the abstract syntax:

$$\begin{array}{rcll} P & ::= & C & \text{— class} \\ & | & c^{\underline{b}} & \text{— constant symbol} \\ & | & X & \text{— variable} \\ & | & \pi_a P & \text{— record projection} \\ & | & \mathit{ins}_a P & \text{— variant insertion} \\ & | & !P & \text{— dereferencing} \\ & | & \mathit{Mk}^C P & \text{— object identity referencing} \end{array}$$

■

A term C represents the set of all object identities of class C. A term $\pi_a P$ represents the a component of the term P, where P should be a term of record type with a as one of its attributes. $ins_a P$ represents a term of variant type built out of the term P and the choice a. $!P$ represents the value associated with the term P, where P is a term representing an object identity. The term $Mk^C P$ represents the object identity of class C with key P.

We can also construct a version of the language, $WOL^{\mathcal{S}}$, for an un-keyed schema, $\mathcal{S}$, by missing out the term constructors Mk^C, $C \in \mathcal{C}$, and skipping the corresponding typing rules and semantic operators in the following definitions.

We will introduce the shorthand notation $P.a$, defined by

$$P.a \equiv \pi_a(!P)$$

since this construct will occur particularly often. For example, if the variable X is bound to some object identity, then the term $X.name$, or $\pi_{name}(!X)$, represents the *name* field of the value associated with X, which must be a suitable record.

Type Contexts and Typing Terms

A **type context**, Γ, is a partial function (with finite domain) from variables to types:

$$\Gamma : Var \overset{\sim}{\rightarrow} Types^{\mathcal{C}}$$

Definition 13.2: Given a type context Γ, the relation $(\Gamma \vdash :) \subseteq Terms^{\mathcal{S}} \times Types^{\mathcal{C}}$ is the smallest relation satisfying the rules:

$$\frac{}{\Gamma \vdash C : \{C\}} \qquad \frac{}{\Gamma \vdash c^{\underline{b}} : \underline{b}}$$

$$\frac{X \in dom(\Gamma)}{\Gamma \vdash X : \Gamma(X)} \qquad \frac{\Gamma \vdash P : (a_1 : \tau_1, \ldots, a_k : \tau_k)}{\Gamma \vdash \pi_{a_i} P : \tau_i}$$

$$\frac{\Gamma \vdash P : \tau_i}{\Gamma \vdash ins^{a_i} P : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle} \qquad \frac{\Gamma \vdash P : C}{\Gamma \vdash !P : \tau^C}$$

$$\frac{\Gamma \vdash P : \kappa^C}{\Gamma \vdash Mk^C P : C}$$

■

So a type context represents a set of assumptions about the types of the values bound to variables, namely that a variable X is bound to a value of type $\Gamma(X)$ if $X \in dom(\Gamma)$. The typing relation $\Gamma \vdash P : \tau$ means that if, for each variable $X \in dom(\Gamma)$, X has type $\Gamma(X)$, then the term P has type τ.

For example, for the schema from example 7.1, if $\Gamma(X) \equiv \mathit{City}$, then $\Gamma \vdash !X : (\mathit{name} : \mathit{str}, \mathit{state} : \mathit{State})$ and $\Gamma \vdash \pi_{\mathit{name}}(!X) : \mathit{str}$.

Note that, if P is a term and Γ a typing context, there may be multiple types τ such that $\Gamma \vdash P : \tau$.

Atoms

Atomic formulae or *atoms* are the basic building blocks of formulae in our language. An atom represents one simple statement about some values.

Definition 13.3: The set of **atoms** for $(\mathcal{S}, \mathcal{K})$, $\mathit{Atoms}^{\mathcal{S}}$, ranged over by $\phi, \psi, \ldots$ is given by the abstract syntax:

$$\begin{array}{rcl} \phi & ::= & P \doteq Q \\ & \mid & P \dot{\neq} Q \\ & \mid & P \dot{\in} Q \\ & \mid & P \dot{\notin} Q \\ & \mid & p^{\underline{b}_1 \cdots \underline{b}_r}(P_1, \ldots, P_r) \\ & \mid & \mathsf{False} \end{array}$$

■

The atoms $P \doteq Q$, $P \dot{\neq} Q$, $P \dot{\in} Q$ and $P \dot{\notin} Q$ represent the obvious comparisons between terms. $p^{\underline{b}_1 \cdots \underline{b}_r}(P_1, \ldots, P_r)$ represents the application of the predicate p to the terms $P_1, \ldots, P_n$. False is an atom which is never satisfied, and is used to represent inconsistent database states.

We mark the symbols $=$, $\neq$, $\in$ and $\notin$ with dots in our syntax in order to distinguish them from the same symbols used as meta-symbols (with their traditional meanings) elsewhere in the paper. However, where no ambiguity is likely to arise, we may omit these dots.

Definition 13.4: An atom ϕ is said to be **well-typed** by a type context Γ iff

1. $\phi \equiv P \doteq Q$ or $\phi \equiv P \dot{\neq} Q$ and $\Gamma \vdash P : \tau$, $\Gamma \vdash Q : \tau$ for some τ; or
2. $\phi \equiv P \dot{\in} Q$ or $\phi \equiv P \dot{\notin} Q$ and $\Gamma \vdash P : \tau$, $\Gamma \vdash Q : \{\tau\}$ for some τ; or
3. $\phi \equiv p^{\underline{b}_1 \cdots \underline{b}_r}(P_1, \ldots, P_r)$ and $\Gamma \vdash P_i : \underline{b}_i$ for $i = 1, \ldots, r$; or
4. $\phi \equiv \mathsf{False}$.

■

Intuitively an atom is well-typed iff that atom *makes sense* with respect to the types of the terms occurring in the atom. For example, for an atom $P = Q$, it wouldn't make sense to reason about the terms P and Q being equal unless they were potentially of the same type.

Term Occurrences and Range Restriction

The concept of *range-restriction* is used to ensure that every term in a collection of atoms is bound to some value occurring in a database instance. This is a necessary requirement if we wish to infer types for the terms, and also to ensure that the truth of a statement of our logic is dependent only on the instance and not the underlying domains of the various types.

In order to define range-restriction we must first introduce the concept of *term occurrences.* There can be several distinct occurrences of a particular term, P, in a set of atoms, and it is necessary to ensure that each individual occurrence of a term is independently range-restricted. Formally a term occurrence can be identified by the atom in which it occurs and a path within the parse tree of that atom. We will develop such a formal notion of term-occurrence and use it in our definition of range restriction in this sub-section, though later we will return to relying on an intuitive notion of occurrences of terms.

Suppose that P is a term. We define the partial function $occ(P) : \mathbb{N} \overset{\sim}{\to} Terms^{\mathcal{S}}$ by

1. If $P \equiv c^{\underline{b}}$, $c^{\underline{b}}$ a constant symbol, or $P \equiv C$, $C \in \mathcal{C}$, or $P \equiv X$, $X \in Var$, then $dom(occ(P)) = \{0\}$ and $occ(P)(0) \equiv P$.
2. If $P \equiv \pi_a Q$ or $P \equiv ins_a Q$ or $P \equiv !Q$ or $P \equiv Mk^C Q$, then
$$dom(occ(P)) = \{0\} \cup \{i+1 | i \in dom(occ(Q))\}$$
and
$$occ(P)(i) \equiv \begin{cases} P & \text{if } i = 0 \\ occ(Q)(i-1) & \text{otherwise} \end{cases}$$

If ϕ is an atom then we define the partial function $occ(\phi) : \mathbb{N} \times \mathbb{N} \overset{\sim}{\to} Terms^{\mathcal{S}}$ by

1. If $\phi \equiv (P_1 \doteq P_2)$ or $\phi \equiv (P_1 \dot{\neq} P_2)$ or $\phi \equiv (P_1 \dot{\in} P_2)$ or $\phi \equiv (P_1 \dot{\notin} P_2)$, then
$$dom(occ(\phi)) = \{(1,i) | i \in dom(occ(P_1))\} \cup \{(2,j) | j \in dom(occ(P_2))\}$$
and
$$occ(\phi)(i,j) \equiv occ(P_i)(j)$$
2. If $\phi \equiv p^{\underline{b}_1 \cdots \underline{b}_r}(P_1, \ldots, P_r)$ then
$$dom(occ(\phi)) = \{(i,j) | i \in 1, \ldots, r,\ j \in dom(occ(P_i))\}$$
and
$$occ(\phi)(i,j) \equiv occ(P_i)(j)$$
3. If $\phi \equiv \mathsf{False}$ then $dom(occ(\phi)) = \emptyset$.

Hence the map $occ(\phi)$ lets us identify and access any particular occurrence of a term in the atom ϕ. The pairs of integers may be thought of as representing a path in the parse tree of ϕ.

Definition 13.5: Suppose Φ is a set of atoms and P a term. An **occurrence of P in** Φ consists of an atom $\phi \in \Phi$ and a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$ such that $(i, j) \in dom(occ(\phi))$ and $occ(\phi)(i, j) = P$.

We write $Occ(\Phi)$ for the set of all term occurrences in Φ. ■

Note that there may be several occurrences of the same term in a particular set of atoms. For example, if we take

$$\Phi \equiv \{X\dot{\in}C,\ !X\doteq ins_aY,\ Z\doteq ins_aY\}$$

then there are two occurrences of the term X, namely $((X\dot{\in}C), (1, 0))$ and $((!X\doteq ins_aY), (1, 1))$, and there are two occurrences of the term ins_aY, namely $((!X\doteq ins_aY), (2, 0))$ and $((Z\doteq ins_aY), (2, 0))$.

Definition 13.6: Suppose Φ is a set of atoms, and $(\phi, (i, j))$ is an *occurrence* of a term P in Φ. Then $(\phi, (i, j))$ is said to be **range-restricted** in Φ iff one of the following holds:

1. $P \equiv C$ where $C \in \mathcal{C}$ is a class;
2. $P \equiv c^{\underline{b}}$ where $c^{\underline{b}}$ is a constant symbol;
3. $P \equiv \pi_a Q$ where $(\phi, (i, j+1))$ is a range restricted occurrence of the term Q in Φ;
4. $(\phi, (i, j-1))$ is a range-restricted occurrence of a term $Q \equiv ins_aP$ in Φ;
5. $P \equiv !Q$ where $(\phi, (i, j+1))$ is a range-restricted occurrence of the term Q in Φ;
6. $(\phi, (i, j)) = (P\doteq Q, (1, 0))$ or $(\phi, (i, j)) = (P\dot{\in}Q, (1, 0))$ and $(\phi, (2, 0))$ is a range restricted occurrence of Q in Φ, or $(\phi, (i, j)) = (Q\doteq P, (2, 0))$ and $(\phi, (1, 0))$ is a range-restricted occurrence of Q in Φ.
7. $P \equiv X$, a variable, and there is a range-restricted occurrence of X in Φ.

■

Note: The distinction here, between syntactic terms and occurrences of those terms in a set of atoms, is important: it is possible for a syntactic term to occur two or more times in a set of atoms, but for only one occurrence of that term to be range-restricted.

For example consider the set of atoms

$$\Phi \equiv \{X\dot{\in}C,\ !X\doteq ins_aY,\ Z\doteq ins_aY\}$$

Here the first occurrence of the term ins_aY is range-restricted, while the second occurrence of ins_aY and the term Z are not.

Clauses

Definition 13.7: A **clause** consists of two finite sets of atoms: the **head** and the **body** of the clause. Suppose $\Phi = \{\phi_1, \ldots, \phi_k\}$ and $\Psi = \{\psi_1, \ldots, \psi_l\}$. We write

$$\psi_1, \ldots, \psi_l \Longleftarrow \phi_1, \ldots, \phi_k$$

or

$$\Psi \Longleftarrow \Phi$$

for the clause with head Ψ and body Φ. Intuitively the meaning of a clause is that if the conjunction of the atoms in the body holds then the conjunction of the atoms in the head also holds. ■

For example, the clause

$$Y.state = X \Longleftarrow X \in State, Y = X.capital$$

means that, for every object identity X in the class *State*, if Y is the capital of X then X is the state of Y.

Well-Formed Clauses

Definition 13.8: A set of atoms Φ is said to be **well-typed** if there is a type context Γ such that each atom in Φ is well-typed by Γ.

A set of atoms Φ is said to be **well-formed** iff it is well typed, and every term occurrence in Φ is range-restricted in Φ.

A clause $\Psi \Longleftarrow \Phi$ is said to be **well-formed** iff Φ is well-formed and $\Phi \cup \Psi$ is well-formed. ■

Intuitively a well-formed clause is one that makes sense, in that all the terms of the clause refer to values in the database, and the types of the terms are compatible with the various predicates being applied to them. In fact we will only be interested in clauses which are well-formed.

Proposition 13.1: If Φ is a well-formed set of atoms then there is a unique type context Γ such that $dom(\Gamma) = Var(\Phi)$ and every atom in Φ is well-typed by Γ. ■

Before proving this proposition we need to define a typing relation on *term occurrences*. Given a set of atoms Φ, we define the relation $(\Phi \vdash :) \subseteq Occ(\Phi) \times Types^{\mathcal{S}}$ between term occurrences in Φ and types to be the smallest relation such that

1. If w is an occurrence of a term $P \equiv C$ in Φ, $C \in \mathcal{C}$, then $\Phi \vdash w : \{C\}$;
2. If w is an occurrence of a term $P \equiv c^{\underline{b}}$ in Φ, then $\Phi \vdash w : \underline{b}$;
3. If $w = (\phi, (i, j))$ is an occurrence of a term $P \equiv \pi_{a_i} Q$, where $\Phi \vdash (\phi, (i, j+1)) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$ then $\Phi \vdash w : \tau_i$;

4. If $w = (\phi, (i, j+1))$, where $w' = (\phi, (i, j))$ is an occurrence of a term $Q \equiv \mathit{ins}_{a_i} P$, and $\Phi \vdash w' : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$, then $\Phi \vdash w : \tau_i$;
5. If $w = (\phi, (i, j))$ is an occurrence of a term $P \equiv !Q$, where $\Phi \vdash (\phi, (i, j+1)) : C$ then $\Phi \vdash w : \tau^C$;
6. If $w = (\phi, (1, 0))$ where ϕ is of the form $P \dot{\in} Q$, and $\Phi \vdash (\phi, (2, 0)) : \{\tau\}$ then $\Phi \vdash w : \tau$;
7. If $w = (\phi, (i, 0))$ where ϕ is of the form $P \doteq Q$, and $i = 1$ and $\Phi \vdash (\phi, (2, 0)) : \tau$, or $i = 2$ and $\Phi \vdash (\phi, (1, 0)) : \tau$, then $\Phi \vdash w : \tau$;
8. If w is an occurrence of a term X, X a variable, and there is a occurrence w' of X such that $\Phi \vdash w' : \tau$, then $\Phi \vdash w : \tau$.

We can now proceed with the proof of proposition 13.1.

Proof: We will prove that, if Φ is a well-formed set of atoms, which is well-typed by Γ and w is an occurrence of a term P in Φ, then there is a *unique* type τ such that $\Phi \vdash w : \tau$, and that if $\Phi \vdash w : \tau$ then $\Gamma \vdash P : \tau$.

The proof will be by induction on the proof that $\Phi \vdash w : \tau$. Note we only need to be concerned about uniqueness of types of an occurrence w of a term P if P has the form $\mathit{ins}_a Q$, since otherwise there is at most one type τ such that $\Gamma \vdash P : \tau$. We will give some sample cases of the induction:

If w is an occurrence of a term $P \equiv C$, $C \in \mathcal{C}$, then $\Phi \vdash w : \{C\}$, and $\Gamma \vdash C : \{C\}$ as required.

If $w = (\phi, (i, j))$ is an occurrence of $P = \pi_{a_i} Q$, and $\Phi \vdash (\phi, (i, j+1)) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$. Then, by our induction hypothesis, $\Gamma \vdash Q : (a_1 : \tau_1, \ldots, a_k : \tau_k)$, and so $\Gamma \vdash \pi_{a_i} Q : \tau_i$.

Suppose $w = (\phi, (1, 0))$ where $\phi \equiv (P \doteq Q)$, and $\Phi \vdash (\phi, (2, 0)) : \tau$ independently. So Q is not of the form $\mathit{ins}_a Q'$. Then, by our induction hypothesis, $\Gamma \vdash Q : \tau$ and there is no other type τ' for which $\Gamma \vdash Q : \tau'$. Hence, since ϕ is well typed, we have $\Gamma \vdash P : \tau$. Suppose P has the form $\mathit{ins}_a R$. Then we observe that the only possible way to derive $\Phi \vdash w : \tau'$ for some τ' is if $\Phi \vdash (\phi, (2, 0)) : \tau'$, and that in this case $\tau' = \tau$ by our induction hypothesis.

The other induction cases are similar.

It follows from this induction that for each variable $X \in \mathit{Var}(\Phi)$, there is a unique τ such that $\Phi \vdash w : \tau$ for every occurrence w of X. Hence for any type context Γ such that Γ well-types Φ, we must have $\Gamma(X) = \tau$. Hence result. ■

Corollary 13.2: If $\Psi \Longleftarrow \Phi$ is a well-formed clause then there is a unique type context Γ such that $\mathit{dom}(\Gamma) = \mathit{Var}(\Phi \cup \Psi)$ and $\Psi \Longleftarrow \Phi$ is well-typed by Γ. ■

Though not difficult, this result is significant in that it means we can assign a unique type to every term occurring in a well-formed clause.

If Φ is a well-formed set of atoms and P is a term occurring in Φ, we write $\Phi \vdash P : \tau$ to mean $\Phi \vdash w : \tau$ where w is some occurrence of P in Φ.

Example 13.1: Let us first add some additional attributes to our running example. Consider the schema $\mathcal{S}$ with classes

$$\mathcal{C} \equiv \{City, State\}$$

and

$$\begin{array}{rcl} \mathcal{S}(City) & \equiv & (name : str,\ state : State,\ popl : int) \\ \mathcal{S}(State) & \equiv & (name : str,\ capital : City,\ popl : int,\ neighbors : \{State\}) \end{array}$$

So both Cities and States have attributes representing their population, and States also have an attribute representing their neighboring States.

Our model itself already ensures the fundamental referential integrity constraints: that the state of each City is in the States extent, that the capital of each State is in the City extent, and that the neighbors of each State are in the States extent. However we would also like to assert additional constraints such as that the capital City of a State is in that State. This could be represented by the clause:

$$Y.state = X \Longleftarrow X \in State, Y = X.capital$$

Equally we would like constraints ensuring that no State is its own neighbor, and each state is a neighbor of its neighbors:

$$\begin{array}{l} Y \notin Y.neighbors \Longleftarrow Y \in State \\ Y \in Z.neighbors \Longleftarrow Y \in State, Z \in Y.neighbors \end{array}$$

Finally we might like to make some restrictions on the values that some other attributes may take, for example that the population of any city is smaller than the population of its state:

$$X.popl \leq X.state.popl \Longleftarrow X \in City$$

Here we're using an additional predicate, $\leq$, on integers, representing the normal ordering on integers. ∎

13.2 Semantics

In this section we will define a semantics for $WOL^{\mathcal{SK}}$ in terms of the model defined in 7.3.

Semantics of Terms

Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance of $(\mathcal{S}, \mathcal{K})$. An **$\mathcal{I}$-environment**, ρ, is a partial function from *Var* to $\mathbf{D}(\sigma^{\mathcal{C}})$. We write $Env(\mathcal{I})$ for the set of all $\mathcal{I}$-environments.

For each constant symbol $c^{\underline{b}}$ we assume an interpretation $\overline{c} \in \mathbf{D}^{\underline{b}}$.

Definition 13.9: We define the semantic operator $[\![\cdot]\!]\mathcal{I} : \mathit{Terms}^{\mathcal{S}} \to \mathit{Env}(\mathcal{I}) \to \mathbf{D}(\sigma^{\mathcal{C}})$ by:

$$\begin{array}{rcl}
[\![C]\!]\mathcal{I}\rho & \equiv & \sigma^C \qquad\quad — C \in \mathcal{C} \\
[\![c^{\underline{b}}]\!]\mathcal{I}\rho & \equiv & \overline{c} \qquad\quad — c^{\underline{b}} \text{ a constant symbol} \\
[\![\pi_a P]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} ([\![P]\!]\mathcal{I}\rho)a & \text{if } [\![P]\!]\mathcal{I}\rho \in (\mathcal{A} \overset{\sim}{\to} \mathbf{D}(\mathcal{I})) \\ & \text{and } a \in \mathit{dom}([\![P]\!]\mathcal{I}\rho) \\ \text{undefined} & \text{otherwise} \end{array} \right. \\
[\![\mathit{ins}_a P]\!]\mathcal{I}\rho & \equiv & (a, [\![P]\!]\mathcal{I}\rho) \\
[\![!P]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} \mathcal{V}^C([\![P]\!]\mathcal{I}\rho) & \text{if } [\![P]\!]\mathcal{I}\rho \in \sigma^C \text{ for some } C \in \mathcal{C} \\ \text{undefined} & \text{otherwise} \end{array} \right. \\
[\![\mathit{Mk}^C(P)]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} o & \text{if } [\![P]\!]\mathcal{I}\rho \in [\![\kappa^C]\!]\mathcal{I} \text{ and } o \in \sigma^C \\ & \text{such that } \mathcal{K}^C(o) = [\![P]\!]\mathcal{I}\rho \\ \text{undefined} & \text{otherwise} \end{array} \right. \\
[\![X]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} \rho(X) & \text{if } X \in \mathit{dom}(\rho) \\ \text{undefined} & \text{otherwise} \end{array} \right.
\end{array}$$

■

For example, for the key specification of example 10.1 and instance of example 7.2,

$$[\![\mathit{Mk}^{\mathit{State}}(\text{“Pennsylvania”})]\!]\mathcal{I}() \ = \ \mathit{PA}$$

and

$$[\![\pi_{\mathit{capital}}(!X)]\!]\mathcal{I}(X \mapsto \mathit{PA}) \ = \ \mathit{Harris}$$

An $\mathcal{I}$-environment, ρ, is said to *satisfy* a type context, Γ, iff $\mathit{dom}(\rho) = \mathit{dom}(\Gamma)$ and $\rho(X) \in [\![\Gamma(X)]\!]\mathcal{I}$ for each $X \in \mathit{dom}(\rho)$.

Proposition 13.3: If $\Gamma \vdash P : \tau$ and ρ satisfies Γ then $[\![P]\!]\mathcal{I}\rho$ is defined and $[\![P]\!]\mathcal{I}\rho \in [\![\tau]\!]\mathcal{I}$. ■

Semantics of Atoms

For each auxiliary predicate $p^{\underline{b}_1 \cdots \underline{b}_r}$ we assume a relation $\overline{p} \subseteq \mathbf{D}^{\underline{b}_1} \times \ldots \times \mathbf{D}^{\underline{b}_r}$.

Definition 13.10: We define the semantic operator $[\![\cdot]\!]\mathcal{I} : \mathit{Atoms}^{\mathcal{S}} \to \mathit{Env}(\mathcal{I}) \to \{\mathbf{T}, \mathbf{F}\}$ by:

$$\begin{array}{rcl}
[\![P \doteq Q]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} \mathbf{T} & \text{if } [\![P]\!]\mathcal{I}\rho \text{ and } [\![Q]\!]\mathcal{I}\rho \text{ are defined} \\ & \text{and } [\![P]\!]\mathcal{I}\rho = [\![Q]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{array} \right. \\
[\![P \dot{\neq} Q]\!]\mathcal{I}\rho & \equiv & \left\{ \begin{array}{ll} \mathbf{T} & \text{if } [\![P]\!]\mathcal{I}\rho \text{ and } [\![Q]\!]\mathcal{I}\rho \text{ are defined} \\ & \text{and } [\![P]\!]\mathcal{I}\rho \neq [\![Q]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{array} \right.
\end{array}$$

$$\begin{aligned}
[\![P \dot{\in} Q]\!]\mathcal{I}\rho &\equiv \begin{cases} \mathbf{T} & \text{if } [\![P]\!]\mathcal{I}\rho \text{ and } [\![Q]\!]\mathcal{I}\rho \text{ are defined} \\ & \text{and } [\![P]\!]\mathcal{I}\rho \in [\![Q]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{cases} \\
[\![P \dot{\notin} Q]\!]\mathcal{I}\rho &\equiv \begin{cases} \mathbf{T} & \text{if } [\![P]\!]\mathcal{I}\rho \text{ and } [\![Q]\!]\mathcal{I}\rho \text{ are defined} \\ & \text{and } [\![P]\!]\mathcal{I}\rho \notin [\![Q]\!]\mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{cases} \\
[\![p(P_1, \ldots, P_r)]\!]\mathcal{I}\rho &\equiv \begin{cases} \mathbf{T} & \text{if } ([\![P_1]\!]\mathcal{I}\rho, \ldots, [\![P_r]\!]\mathcal{I}\rho) \in \overline{p} \\ \mathbf{F} & \text{otherwise} \end{cases} \\
[\![\mathsf{False}]\!]\mathcal{I}\rho &\equiv \mathbf{F}
\end{aligned}$$

If Φ is a set of atoms, we define $[\![\Phi]\!]\mathcal{I}\rho$ by

$$[\![\Phi]\!]\mathcal{I}\rho \equiv \begin{cases} \mathbf{T} & \text{if } [\![\phi]\!]\mathcal{I}\rho = \mathbf{T} \text{ for each } \phi \in \Phi \\ \mathbf{F} & \text{otherwise} \end{cases}$$

■

Semantics of Clauses

In a clause, any variables occurring in the body of the clause are taken to be universally quantified, while any additional variables occurring in the head are existentially quantified. Hence a clause is satisfied if, for any instantiation of the variables in the body of the clause such that all the atoms in the body are true, there is an instantiation of the remaining variables in the head of the clause such that all the atoms in the head are also true.

Definition 13.11: Suppose $\Delta \equiv (\psi_1, \ldots, \psi_l \Longleftarrow \phi_1 \ldots \phi_k)$ is a well-formed clause. An instance $\mathcal{I}$ is said to **satisfy** Δ iff, for any environment ρ with $dom(\rho) = Var(\phi_1, \ldots, \phi_k)$, if

$$[\![\phi_i]\!]\mathcal{I}\rho = \mathbf{T}$$

for $i = 1, \ldots, k$, then there is an extension of ρ, ρ', with $dom(\rho') = Var(\phi_1, \ldots, \phi_k, \psi_1, \ldots, \psi_l)$, such that

$$[\![\psi_j]\!]\mathcal{I}\rho' = \mathbf{T}$$

for $i = 1, \ldots, l$. ■

Two clauses, Δ and Δ', are said to be **equivalent** iff for any instance $\mathcal{I}$, $\mathcal{I}$ satisfies Δ iff $\mathcal{I}$ satisfies Δ'.

Example 13.2: For the instance of example 7.2, suppose the environment ρ is given by

$$\rho \equiv (X \mapsto PA, Y \mapsto Phila)$$

then

$$\begin{array}{rcl}
[\![X \in \mathit{State}]\!]\mathcal{I}\rho & = & \mathbf{T} \\
[\![Y = X.\mathit{capital}]\!]\mathcal{I}\rho & = & \mathbf{T} \\
[\![Y.\mathit{state} = X]\!]\mathcal{I}\rho & = & \mathbf{T}
\end{array}$$

If we check other suitably typed environments, we fine that any environment in which the first two atoms are true also makes the third atom true. So this instance satisfies the clause

$$Y.\mathit{state} = X \Longleftarrow X \in \mathit{State}, Y = X.\mathit{capital}$$

■

If **Pr** is a set of clauses and Δ is a clause, we write $\mathbf{Pr} \models \Delta$ to mean that, for any instance $\mathcal{I}$, if $\mathcal{I}$ satisfies each clause in **Pr** then $\mathcal{I}$ also satisfies Δ.

13.3 Semi-Normal Forms

The language *WOL* is very rich in that it allows us many different ways of expressing the same thing. However when performing structural manipulations of the clauses, as we will do when dealing with implementing transformations later, our life is made easier if there is less variance in the way things can be expressed, so that techniques such as unification can be applied simply.

In this section we will define a *semi-normal form* (snf) for clauses which reduces the variety of forms the atoms of a clause can take. For every clause we will show that there is an equivalent clause in semi-normal form.

There are two main purposes in converting a clause to semi-normal form: firstly, because any two equivalent sets of atoms in snf differ only in their choices of variables, we can apply unification algorithms to atoms and clauses in snf. Secondly, converting a clause to snf ensures that there is a variable introduced at every point where the database is being referenced by the clause. This makes it easy to reason about the information being accessed or implied by a particular clause, which will be necessary in our analysis of recursion. In addition, assuming that clauses are in snf allows us to reduce the number of cases we must consider, and consequently simplifies many of our proofs.

Definition 13.12: An atom is said to be in **semi-normal form** iff it is of one of the forms:

$$\begin{gathered} X \doteq c^{\underline{b}} \\ X \doteq C \\ X \doteq \pi_a Y \\ X \doteq ins_a Y \\ X \doteq\, !Y \\ X \doteq Mk^C(Y) \\ X \doteq Y \\ X \dot{\neq} Y \\ X \dot{\in} Y \\ X \dot{\notin} Y \\ p^{\underline{b}_1 \cdots \underline{b}_r}(X_1, \ldots, X_r) \\ \mathsf{False} \end{gathered}$$

where X and Y are variables, $c^{\underline{b}}$ a constant symbol, $C \in \mathcal{C}$ a class, and $a \in \mathcal{A}$ an attribute label. ■

Note, in particular, that the terms of a snf atom will contain no nested operators, and a snf atom using some predicate other than $\doteq$ will contain only variables as terms.

Lemma 13.4: For any set of atoms, Φ, there is a set of atoms in snf, Φ', with $Var(\Phi) \subseteq Var(\Phi')$, such that for any instance $\mathcal{I}$ and $\mathcal{I}$-environment ρ with $dom(\rho) = Var(\Phi)$, $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$ if and only if there is an extension ρ' of ρ such that $[\![\Phi']\!]\mathcal{I}\rho' = \mathbf{T}$. ■

Intuitively this means that for any set of atoms there is an equivalent set of atoms in semi-normal form, subject to the introduction of additional variables. In general we expect a single atom to be equivalent to a set of snf atoms.

Proof: If is sufficient to show that for any atom ϕ there is a set of semi-normal form atoms Φ such that, for any environment ρ with $dom(\rho) = Var(\phi)$, $[\![\phi]\!]\mathcal{I}\rho = \mathbf{T}$ iff there is an extension ρ' of ρ with $[\![\Phi]\!]\mathcal{I}\rho' = \mathbf{T}$.

Assume that we have an infinite supply of unused variables. For any term P and new variable W, we define the set of atoms $snf(P, W)$ by

1. If $P \equiv C$, $C \in \mathcal{C}$, then $snf(P, W) \equiv \{W \doteq C\}$
2. If $P \equiv c^{\underline{b}}$, $c^{\underline{b}}$ a constant symbol, then $snf(P, W) \equiv \{W \doteq c^{\underline{b}}\}$
3. If $P \equiv X$, X a variable, then $snf(P, W) \equiv \{W \doteq X\}$
4. If $P \equiv \pi_a Q$ then $snf(P, W) \equiv \{W \doteq \pi_a V\} \cup snf(Q, V)$ where V is a new variable.
5. If $P \equiv ins_a Q$ then $snf(P, W) \equiv \{W \doteq ins_a V\} \cup snf(Q, V)$ where V is a new variable.
6. If $P \equiv !Q$ then $snf(P, W) \equiv \{W \doteq !V\} \cup snf(Q, V)$ where V is a new variable.

7. If $P \equiv \mathit{Mk}^C Q$ then $\mathit{snf}(P, W) \equiv \{W \doteq \mathit{Mk}^C V\} \cup \mathit{snf}(Q, V)$ where V is a new variable.

Then, for any environment ρ such that $\mathit{dom}(\rho) = \mathit{Var}(P)$ and $[\![P]\!]\mathcal{I}\rho$ is defined, there is an extension ρ' of ρ such that $[\![\mathit{snf}(P, W)]\!]\mathcal{I}\rho = \mathbf{T}$, and if ρ'' is an extension of ρ such that $[\![\mathit{snf}(P, W)]\!]\mathcal{I}\rho = \mathbf{T}$ then $\rho'(W) = [\![P]\!]\mathcal{I}\rho$.

For any atom, ϕ we define the atoms $\mathit{snf}(\phi)$ by

1. If $\phi \equiv (P \doteq Q)$ then $\mathit{snf}(\phi) \equiv \{V \doteq W\} \cup \mathit{snf}(P, V) \cup \mathit{snf}(Q, W)$ where V and W are new variables.
2. If $\phi \equiv (P \dot{\neq} Q)$ then $\mathit{snf}(\phi) \equiv \{V \dot{\neq} W\} \cup \mathit{snf}(P, V) \cup \mathit{snf}(Q, W)$ where V and W are new variables.
3. If $\phi \equiv (P \dot{\in} Q)$ then $\mathit{snf}(\phi) \equiv \{V \dot{\in} W\} \cup \mathit{snf}(P, V) \cup \mathit{snf}(Q, W)$ where V and W are new variables.
4. If $\phi \equiv (P \dot{\notin} Q)$ then $\mathit{snf}(\phi) \equiv \{V \dot{\notin} W\} \cup \mathit{snf}(P, V) \cup \mathit{snf}(Q, W)$ where V and W are new variables.
5. If $\phi \equiv (p^{\underline{b_1 \cdots b_r}}(P_1, \ldots, P_r))$ then $\mathit{snf}(\phi) \equiv \{p^{\underline{b_1 \cdots b_r}}(W_1, \ldots, W_r)\} \cup \mathit{snf}(P_1, W_1) \cup \ldots \cup \mathit{snf}(P_r, W_r)$ where $W_1, \ldots, W_r$ are new variables.
6. If $\phi = \mathsf{False}$ then $\mathit{snf}(\phi) = \{\mathsf{False}\}$.

Then, for any ρ with $\mathit{dom}(\rho) = \mathit{Var}(\phi)$, we have $[\![\phi]\!]\mathcal{I}\rho = \mathbf{T}$ iff there is an extension ρ' of ρ such that $[\![\mathit{snf}(\phi)]\!]\mathcal{I}\rho' = \mathbf{T}$. ■

Example 13.3: The atom $X \doteq Y.\mathit{state}.\mathit{capital}$ is equivalent to the snf atoms

$$\{X \doteq \pi_{\mathit{capital}}(U),\ U \doteq !V,\ V \doteq \pi_{\mathit{state}}(W),\ mW \doteq !Y\}$$

■

Definition 13.13: A clause, $\Psi \Longleftarrow \Phi$, is in **semi-normal form** iff

1. all its atoms are in semi-normal form;
2. Φ contains no atoms of the form $X \doteq Y$;
3. for any atoms of the form $X \doteq Y$ in Ψ, $X \in \mathit{var}(\Phi)$ and $Y \in \mathit{var}(\Phi)$;
4. if $X, Y, Z \in \mathit{var}(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $Y \doteq \pi_a X$ and $Z \doteq \pi_a X$ then $Y \equiv Z$; and if $\Phi \cup \Psi \vdash X : (a_1 : \tau_1, \ldots, a_n : \tau_n)$, $\Phi \cup \Psi \vdash X : (a_1 : \tau_1, \ldots, a_n : \tau_n)$ and $\Phi \cup \Psi$ contains atoms $Z_1 \doteq \pi_{a_1} X, \ldots, Z_n \doteq \pi_{a_n} X$ and $Z_1 \doteq \pi_{a_1} Y, \ldots, Z_n \doteq \pi_{a_n} Y$ then $X \equiv Y$;
5. if $X, Y, Z \in \mathit{var}(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq \mathit{ins}_a Y$ and $X \doteq \mathit{ins}_b Z$ then $a \equiv b$ and $Y \equiv Z$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq \mathit{ins}_{a_i} Z$ and $Y \doteq \mathit{ins}_{a_i} Z$ and $\Phi \cup \Psi \vdash X : \langle\!|a_1 : \tau_1, \ldots, a_n : \tau_n|\!\rangle$, $\Phi \cup \Psi \vdash X : \langle\!|a_1 : \tau_1, \ldots, a_n : \tau_n|\!\rangle$ then $X \equiv Y$;

6. if $X, Y, Z \in \mathit{var}(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq !Z$ and $Y \doteq !Z$ then $X \equiv Y$;
7. if $X, Y, Z \in \mathit{var}(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq \mathit{Mk}^C Z$ and $Y \doteq Mk^C Z$ then $X \equiv Y$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq \mathit{Mk}^C Y$ and $X \doteq \mathit{Mk}^C Z$ then $Y \equiv Z$;
8. if $X, Y \in \mathit{var}(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq \underline{c^b}$ and $Y \doteq \underline{c^b}$ then $X \equiv Y$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq \underline{c^b}$ and $X \doteq \underline{d^b}$ then $\underline{c^b} \equiv \underline{d^b}$;
9. if $\Phi \cup \Psi$ contains an atom $X \dot{\neq} Y$ then $X \not\equiv Y$;
10. for any $X, Y \in \mathit{var}(\Phi \cup \Psi)$, $\Phi \cup \Psi$ does not contain both of the atoms $X \dot{\in} Y$ and $X \dot{\notin} Y$; and
11. If Φ contains the atom False then $\Phi = \{\mathsf{False}\}$ and $\Psi = \{\mathsf{False}\}$, and if Ψ contains the atom False then $\Psi = \{\mathsf{False}\}$.

■

So intuitively a clause, $\Psi \Longleftarrow \Phi$, is in semi-normal form if

1. all its atoms are in semi-normal form;
2. given 1, it contains a minimal number of variables: in particular there are no distinct variables, $X, Y \in \mathit{var}(\Phi)$, such that $\Phi \models X \doteq Y$, and there are no distinct variables $X, Y \in \mathit{var}(\Phi \cup \Psi)$ such that either X or Y is in $\mathit{var}(\Psi) \setminus \mathit{var}(\Phi)$ and $\Phi \cup \Psi \models X \doteq Y$; and
3. it contains no contradictory atoms other than the atom False which may only occur by itself in the head of the clause.

The following proposition, though straightforward, is perhaps the most significant result in making semi-normal form clauses useful.

Proposition 13.5: For any clause Δ, there is an equivalent clause Δ', unique up to the choice of variables, such that Δ' is in snf. ■

Proof: Suppose we have a clause $\Psi \Longleftarrow \Phi$. We will construct a semi-normal form clause $\Psi' \Longleftarrow \Phi'$ from $\Psi \Longleftarrow \Phi$ in stages, showing that the clause constructed at each stage is equivalent to the clause of the stage before.

1. Let Φ_1 and Ψ_1 be sets of snf atoms which are equivalent to Φ and Ψ respectively, as constructed in lemma 13.4.
2. If ϕ is an atom of the form $X \doteq Y$ in Φ_1, $X \not\equiv Y$, then replace Φ_1 with $\Phi_1[X/Y]$ and Ψ_1 with $\Psi_1[X/Y]$, where $\Psi[X/Y]$ denotes the set of atoms formed by replacing every occurrence of Y by X in Φ. Repeat this process until there are no more atoms of the form $X \doteq Y$, $X \not\equiv Y$, left in Φ_1. Remove any atoms of the form $X \doteq X$ from Φ_1 and Ψ_1, and call the resulting sets of atoms Φ_2 and Ψ_2.

3. If ϕ is an atom of the form $X \doteq Y$ in Ψ_2, where $X \not\equiv Y$ and $X \notin \mathit{Var}(\Phi_2)$ or $Y \notin \mathit{Var}(\Phi_2)$, then replace Φ_2 and Ψ_2 with $\Phi_2[X/Y]$ and $\Psi_2[X/Y]$ respectively. Repeat this process until there are no remaining atoms of this form in Ψ_2. Remove any atoms of the form $X \doteq X$ in Ψ_2, and call the resulting sets of atoms Ψ_3 and Φ_3.

4. If $\Phi_3 \cup \Psi_3$ contains atoms of one of the forms

   (a) $X \doteq \pi_a Z$ and $Y \doteq \pi_a Z$, where $X \not\equiv Y$

   (b) $Z_i \doteq \pi_{a_i} X$ and $Z_i \doteq \pi_{a_i} Y$, for $i = 1, \ldots, k$, where $\Phi_3 \cup \Psi_3 \vdash X : (a_1 : \tau_1, \ldots, a_k : \tau_k)$ where $\Phi_3 \cup \Psi_3 \vdash X : (a_1 : \tau_1, \ldots, a_k : \tau_k)$ for some types $\tau_1, \ldots, \tau_k$

   (c) $Z \doteq \mathit{ins}_a X$ and $Z \doteq \mathit{ins}_a Y$, where $X \not\equiv Y$

   (d) $X \doteq \mathit{ins}_a Z$ and $Y \doteq \mathit{ins}_a Z$, where $X \not\equiv Y$ and $\Phi_3 \cup \Psi_3 \vdash X : \tau$ and $\Phi_3 \cup \Psi_3 \vdash Y : \tau$ for some type τ

   (e) $X \doteq !Z$ and $Y \doteq !Z$, where $X \not\equiv Y$

   (f) $X \doteq \mathit{Mk}^C Z$ and $Y \doteq \mathit{Mk}^C Z$, where $X \not\equiv Y$

   (g) $Z \doteq \mathit{Mk}^C X$ and $Z \doteq \mathit{Mk}^C Y$, where $X \not\equiv Y$

   (h) $X \doteq c^{\underline{b}}$ and $Y \doteq c^{\underline{b}}$, where $X \not\equiv Y$

   then replace Φ_3 and Ψ_3 by $\Phi_3[X/Y]$ and $\Psi_3[X/Y]$ respectively. Repeat this process until no more sets of atoms of one the above forms are present in $\Psi_3 \cup \Phi_3$. Call the resulting sets of atoms Ψ_4 and Φ_4.

5. If Φ_4 contains atoms of one of the forms

   (a) $X \dot{\neq} X$

   (b) $Z \doteq \mathit{ins}_a X$ and $Z \doteq \mathit{ins}_b X$ where $a \not\equiv b$

   (c) $Z \doteq c^{\underline{b}}$ and $Z \doteq d^{\underline{b}}$ where $c^{\underline{b}} \not\equiv d^{\underline{b}}$

   (d) $X \dot{\in} Y$ and $X \dot{\notin} Y$

   (e) False

   then let $\Phi_5 = \{\mathsf{False}\}$ and $\Psi_5 = \{\mathsf{False}\}$. Otherwise, if $\Phi_4 \cup \Psi_4$ contains atoms of one of the above forms, then let $\Phi_5 = \Phi_4$ and let $\Psi_5 = \{\mathsf{False}\}$. Otherwise let $\Phi_5 = \Phi_4$ and let Ψ_5.

It remains to show that the clause $\Psi \Longleftarrow \Phi$ is equivalent to $\Psi_i \Longleftarrow \Phi_i$ for $i = 1, 2, 3, 4, 5$. It is easy to see that $\Psi_{i+1} \Longleftarrow \Phi_{i+1}$ is equivalent to $\Psi_i \Longleftarrow \Phi_i$ for $i = 1, 2, 3, 4$. We will show that $\Psi \Longleftarrow \Phi$ is equivalent to $\Psi_1 \Longleftarrow \Phi_1$.

Suppose $\mathcal{I}$ is an instance satisfying $\Psi \Longleftarrow \Phi$, and ρ' is an environment such that $[\![\Phi_1]\!]\mathcal{I}\rho' = \mathbf{T}$. Then by lemma 13.4, if ρ is the restriction of ρ' to $\mathit{Var}(\Phi)$ then $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Hence there is an extension of ρ to $\mathit{Var}(\Psi \cup \Phi)$, say ρ'', such that $[\![\Psi]\!]\mathcal{I}\rho'' = \mathbf{T}$. Hence, by lemma 13.4, there is an extension of ρ'' to $\mathit{Var}(\Psi_1 \cup \Phi)$, say ρ''', such that $[\![\Psi_1]\!]\mathcal{I}\rho''' = \mathbf{T}$. Hence if we take ρ^* to be the union of ρ' and ρ''', then ρ^* is an extension of ρ', and $[\![\Psi_1]\!]\mathcal{I}\rho^* = \mathbf{T}$.

Similarly we can show that any $\mathcal{I}$ satisfying $\Psi_1 \Longleftarrow \Phi_1$ also satisfies $\Psi \Longleftarrow \Phi$. Hence result. ∎

Example 13.4: Consider again the clause

$$Y.state = X \Longleftarrow X \in State, Y = X.capital$$

Recall that this is shorthand notation for

$$\pi_{state}(!Y) = X \Longleftarrow X \in State, Y = \pi_{capital}(!X)$$

This is equivalent to the snf clause

$$V = !Y, X = \pi_{state}(V) \Longleftarrow W = State, X \in W, U = !X, Y = \pi_{capital}(U)$$

Note that, for every subterm in the original clause, there is a corresponding variable in the snf clause. ∎

14 Database Transformations

In this section we will show how our language, *WOL*, can be used to specify and implement general structural transformations on databases. Transformations are specified at the schema-level, describing the relationship between instances of two or more schemas. We consider transformations to be specified by a series of logical statements in *WOL*, describing the relationships between two databases, just as we consider constraints to be logical statements about a single database. However, since we are concerned with structural transformations which can be performed efficiently on large quantities of data, rather than general computations, we will need to restrict those sets of logical statements that may be used in order to ensure that the transformation procedure will terminate and can be performed in a single pass.

The language as presented so far deals with a single database schema and instance. However in order to express transformations, or other correspondences between two or more databases, it is necessary to extend the language to deal with multiple distinct database values. In general we will distinguish one of these databases, which we will call the *target* database, and we will refer to the other databases as the *source* databases, though this distinction will become more meaningful when we start considering the actual implementation of database transformations.

14.1 Partitioning Schemas and Instances

We say schemas $\mathcal{S}_1$ and $\mathcal{S}_2$ are *disjoint* iff their sets of classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are disjoint. Suppose $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are pairwise disjoint schemas with sets of classes, $\mathcal{C}_1, \ldots, \mathcal{C}_n$. We define their *union*, $\mathcal{S} \equiv \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$, to be the schema with classes $\mathcal{C} \equiv \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$ and

$$\mathcal{S}(C) \equiv \mathcal{S}_i(C)$$

if $C \in \mathcal{C}_i$.

Definition 14.1: A **partition** of a schema $\mathcal{S}$ is a collection of disjoint schemas $\mathcal{S}_1, \ldots, \mathcal{S}_n$ such that $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$.

If $\mathcal{I}_1, \ldots, \mathcal{I}_n$ are instances of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ respectively, where $\mathcal{I}_i \equiv (\sigma_i^{\mathcal{C}_i}, \mathcal{V}_i^{\mathcal{C}_i})$, then $\mathcal{I} \equiv \mathcal{I}_1 \cup \ldots \cup \mathcal{I}_n$ is given by $\mathcal{I} \equiv (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, where $\mathcal{C} = \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$, $\sigma^C \equiv \sigma_i^C$ and $\mathcal{V}^C \equiv \mathcal{V}_i^C$, for $C \in \mathcal{C}_i$.

Clearly $\mathcal{I}_1 \cup \ldots \cup \mathcal{I}_n$ is an instance of $\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$. Further, for any instance $\mathcal{I}$ of $\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ there exist unique instances $\mathcal{I}_1, \ldots, \mathcal{I}_n$ of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ respectively such that $\mathcal{I} = \mathcal{I}_1 \cup \ldots \cup \mathcal{I}_n$. Given a partition $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of a schema $\mathcal{S}$, and an instance $\mathcal{I}$ of $\mathcal{S}$, we write $\mathcal{I}/\mathcal{S}_i$, $i = 1, \ldots, n$, for the unique instances of $\mathcal{S}_i$ such that $\mathcal{I} = \mathcal{I}/\mathcal{S}_1 \cup \ldots \cup \mathcal{I}/\mathcal{S}_n$.

If $(\mathcal{S}_1, \mathcal{K}_1), \ldots, (\mathcal{S}_n, \mathcal{K}_n)$ are disjoint keyed schemas then we define their union, $(\mathcal{S}, \mathcal{K}) \equiv (\mathcal{S}_1, \mathcal{K}_1) \cup \ldots \cup (\mathcal{S}_n, \mathcal{K}_n)$, to be such that $\mathcal{S} \equiv \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and $\kappa^C \equiv \kappa_i^C$ if $C \in \mathcal{C}_i$, and $\mathcal{K}_{\mathcal{I}}^C \equiv \mathcal{K}_i{}_{\mathcal{I}/\mathcal{S}_i}^C$ if $C \in \mathcal{C}_i$. ■

In looking at transformations, we will concentrate on dealing with transformations between two databases. We will assume that we have a simply-keyed schema $(\mathcal{S}, \mathcal{K})$ with a partition $(\mathcal{S}_{Src}, \mathcal{K}_{Src}), (\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$, and will use the language $WOL^{\mathcal{SK}}$ in order to specify transformations from the *source* schema $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ to the *target* schema, $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$ (though the results can be adapted to non-keyed schemas). The source schema $\mathcal{S}_{Src}$ may in turn be partitioned into a number of distinct schemas, so that these methods apply equally well to transforming a single database or a collection of databases.

14.2 Transformation Clauses and Programs

In analyzing the clauses that describe a transformation, it is necessary to classify the terms of a clause as *source terms*, which refer to part of the source database, and *target terms*, which refer to the target database. Certain terms may be considered to be both source terms and target terms: indeed this is necessary in order for a clause to express the transformation of data between two databases.

Suppose $(\mathcal{S}, \mathcal{K})$ is a schema, partitioned into two schemas $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ and $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$ as described earlier, and Φ is a set of $WOL^{\mathcal{SK}}$ atoms. A term P occurring in Φ is said to be a **target term** in Φ iff

1. $P \equiv C$ where $C \in \mathcal{C}_{Tgt}$, or
2. $P \equiv \pi_a Q$ where Q is a target term in Φ, or
3. P occurs in a term $Q \equiv ins_a P$ where Q is a target term in Φ, or
4. $P \equiv !Q$ where Q is a target term in Φ, or
5. P occurs in a term $Q \equiv Mk^C P$ where Q is a target term in Φ, or
6. Φ contains an atom $P \doteq Q$, $Q \doteq P$ or $P \dot{\in} Q$ where Q is a target term in Φ.

The definition of **source terms** is similar.

Definition 14.2: There are three kinds of *WOL* clauses that are relevant in determining transformations:

1. **target constraints** — containing no source terms;
2. **source constraints** — containing no target terms; and
3. **transformation clauses** — clauses for which the translation into semi-normal form, $\Psi \Longleftarrow \Phi$ satisfies
   (a) each term in Ψ is a target term;
   (b) $\Phi \cup \Psi$ contains no *negative* target atoms: that is, no atoms of the form $P \notin Q$ or $P \neq Q$ where P or Q are target terms;
   (c) for any variable $X \in \mathit{var}(\Phi \cup \Psi)$, such that X is a target term of set type in $\Phi \cup \Psi$, (that is $\Phi \cup \Psi \vdash X : \{\tau\}$ for some τ), then there is at most one atom of the form $X \doteq P$ in $\Phi \cup \Psi$; and
   (d) for every atom $\psi \in \Psi \setminus \Phi$ there is a variable X occurring in ψ such that $X \notin \mathit{Var}(\Phi)$.

■

So a transformation clause is one that does not imply any constraints on the source database, and which only implies the existence of certain objects or values in the target database.

Further a transformation clause is limited to using "non-negative" tests on the target database, and cannot test for equality of target values of set type. This is because we need to be able to apply transformation clauses at points where the target database is only partially instantiated, and therefore the tests must remain true even if additional elements are added to the target database. For example, suppose we were to allow a transformation clause such as

$$1 \in X.a \Longleftarrow X \in C,\ Y \in C,\ X.a = Y.a$$

where C is a class with corresponding type $\tau^C \equiv (a : \{\mathit{int}\})$. Then suppose, at some point during the transformation, we were to find an instantiation of X and Y to two objects, say o_1 and o_2, of class C, such that the body of the clause was true at that point in the transformation. Then the clause would cause the constant 1 to be added to the set $X.a$, thus potentially making the body of the clause no longer true.

Note that it is possible in this characterization for a clause to be both a transformation clause and a target constraint. This reflects the fact that a target constraint can perform two functions: determining the data which must be inserted into a target database, and ensuring the integrity of data being inserted into a database.

Source constraints do not play a part in populating a target database, and, since we will assume that the contents of the source database are already know before we evaluate a transformation, they do not play a direct part in determining a transformation. However they play a significant role in simplifying transformation clauses (see section 17).

Example 14.1: For the schemas of the Cities and Countries databases described earlier, suppose we split the description of the instantiation of the $\mathit{Country}_T$ class over several transformation clauses:

$$X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L \impliedby Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L$$

$$X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{currency} = C \impliedby Z \in \mathit{Country}_E,\ Z.\mathit{name} = N,\ Z.\mathit{currency} = C$$

Combining these clauses gives

$$\begin{array}{rl} & X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L,\ X.\mathit{currency} = C \\ \impliedby & Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L \\ & Z \in \mathit{Country}_E,\ Z.\mathit{name} = N,\ Z.\mathit{currency} = C \end{array}$$

To apply this clause we would need to take the product of the source class $\mathit{Country}_E$ with itself try to bind Y and Z to pairs of objects in $\mathit{Country}_E$ which have the same value on their *name* attribute.

Suppose however, we had a constraint on the source database:

$$X = Y \impliedby X \in \mathit{Country}_E\, Y \in \mathit{Country}_E\, X.\mathit{name} = Y.\mathit{name}$$

That is, *name* is a key for $\mathit{Country}_E$. We could then use this source constraint to simplify our previous, derived transformation clause, in order to form the new clause:

$$\begin{array}{rl} & X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L,\ X.\mathit{currency} = C \\ \impliedby & Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L,\ Y.\mathit{currency} = C \end{array}$$

Note that this clause does not actually give us any new information about the target database, but that it is simpler and more efficient to evaluate.

Suppose we also have a target constraint expressing the key specification for $\mathit{Country}_T$:

$$X = \mathit{Mk}^{\mathit{Country}_T}(X.\mathit{name}) \impliedby X \in \mathit{Country}_T$$

We could combine this constraint with our previous transformation clause in order to get

$$\begin{array}{rl} & X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L,\ X.\mathit{currency} = C,\ X.\mathit{name} = N \\ \impliedby & Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L,\ Y.\mathit{currency} = C \end{array}$$

In this case the target constraint has told us how to instantiate another attribute of the object being inserted into the class $\mathit{Country}_T$. So the target constraint is providing additional information for the transformation. ■

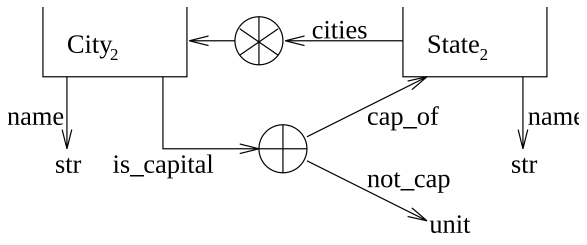


Figure 16: A target schema for Cities and States

A **transformation program**, **Tr**, from schema $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ to schema $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$, is a finite set of source and target constraints and transformation clauses in $WOL^{\mathcal{SK}}$, where $(\mathcal{S}, \mathcal{K}) \equiv (\mathcal{S}_{Src}, \mathcal{K}_{Src}) \cup (\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$.

Example 14.2: We will take the first schema described in examples 7.1 and 10.1 to be our source schema. Our target schema will be given by:

$$\mathcal{C}_{Tgt} \equiv \{State_2, City_2\}$$

and

$$\begin{aligned} \mathcal{S}_{Tgt}(State_2) &\equiv (name : str, cities : \{City_2\}) \\ \mathcal{S}_{Tgt}(City_2) &\equiv (name : str, is\_capital : \langle\!| cap\_of : State_2, not\_cap : unit |\!\rangle) \end{aligned}$$

with key specification

$$\begin{aligned} \mathcal{K}^{State_2}_{Tgt}(o) &\equiv \mathcal{V}^{State_2}(o)(name) \\ \mathcal{K}^{City_2}_{Tgt}(o) &\equiv (name \mapsto \mathcal{V}^{City_2}(o)(name), state \mapsto \{o' \in \sigma^{State_2} | o \in \mathcal{V}^{State_2}(o')(cities)\}) \end{aligned}$$

The schema is illustrated in figure 16. Note that the variant $\langle\!| cap\_of : State_2, not\_cap : unit |\!\rangle$ is an example of a very common construction: it is similar to an optional reference or a pointer with a possible "nil" value in a programming language such as C.

Then a transformation between the two schemas would be given by the transformation clauses

$$Y \in State_2, Y.name = N \Longleftarrow X \in State_A, X.name = N$$

$$\begin{aligned} &W \in City_2,\ W.name = N,\ W.is\_capital = ins_{cap\_of}(V) \Longleftarrow \\ &\qquad X \in City_A,\ X.name = N,\ X.state.capital = X \\ &\qquad V \in State_2,\ V.name = X.state.name \end{aligned}$$

$$\begin{aligned} &W \in City_2,\ W.name = N,\ W.is\_capital = ins_{not\_cap}() \Longleftarrow \\ &\qquad X \in City, X.name = N,\ X.state.capital \neq X \end{aligned}$$

$$\begin{aligned} &W \in Y.cities,\ W.name = C \Longleftarrow \\ &\qquad Z \in City_A,\ Z.name = C,\ Y \in State_2, Y.name = Z.state.name \end{aligned}$$

The first clause says that for each state in the source class State_A there is a corresponding state in the target class State_2 with the same name. The second clause says that for each city in the source class City_A which is the capital of its state, there is a corresponding city in the target class City_2 with the same name and with *is_capital* set to the state of the City, while the third clause says that if a city in the source database is not the capital of its state, then the corresponding city in the target class City_2 has *is_capital* set to *not_cap*. The fourth clause says how the *cities* attribute of the State_2 class is populated by cities.

In addition to these clauses, we would need some constraints on the target database in order to identify elements of State_2 and City_2:

$$X = \mathit{Mk}^{\mathit{State}_2}(X.\mathit{name}) \impliedby X \in \mathit{State}_2$$

$$\begin{aligned} X = \mathit{Mk}^{\mathit{City}_2}(W),\ \pi_{\mathit{name}}W = X.\mathit{name},\ \pi_{\mathit{state}}W = Y \\ \impliedby X \in \mathit{City}_2,\ Y \in \mathit{State}_2,\ X \in Y.\mathit{cities} \end{aligned}$$

$$\begin{aligned} Y \in \mathit{State}_2, X \in Y.\mathit{cities} &\impliedby X \in \mathit{City}_2 \\ X = Y &= X \in \mathit{State}_2, Y \in \mathit{State}_2, Z \in X.\mathit{cities}, Z \in Y.\mathit{cities} \end{aligned}$$

The first two of these constrains are "key constraints" on State_2 and City_2, and tell us how to generate their keys. The third says that every city must be in the *cities* set of some state, and the third says that no city can lie in the *cities* set of two distinct states. ■

14.3 Transformations of Instances

Definition 14.3: Suppose that **Tr** is a transformation program from schema $\mathcal{S}_{\mathit{Src}}$ to $\mathcal{S}_{\mathit{Tgt}}$, and that $\mathcal{I}_{\mathit{Src}}$ is an instance of $\mathcal{S}_{\mathit{Src}}$. Then an instance $\mathcal{I}_{\mathit{Tgt}}$ of $\mathcal{S}_{\mathit{Tgt}}$ is said to be a **Tr-transformation** of $\mathcal{I}_{\mathit{Src}}$ iff, for each clause $(\Psi \impliedby \Phi) \in \mathbf{Tr}$, $\mathcal{I}_{\mathit{Src}} \cup \mathcal{I}_{\mathit{Tgt}}$ satisfies $\Psi \impliedby \Phi$. ■

Unfortunately the **Tr**-transformation of an instance is not, in general unique. A transformation program will imply that certain things must be in the target database, but will not imply that other additional things cannot be included. Consequently there may be infinitely many **Tr**-transformations of a particular instance, representing the inclusion of arbitrary additional data, and so it is necessary to characterize the unique smallest **Tr**-transformation when it exists.

Deterministic and Complete Transformation Programs

Definition 14.4: Suppose **Tr** is a set of clauses. A clause $\Psi \impliedby \Phi$ is said to be **deterministic** with respect to **Tr** iff, for any instance $\mathcal{I}$ satisfying all the clauses in **Tr** and any environment ρ such that $\mathit{dom}(\rho) = \mathit{Var}(\Phi)$ and $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$ there is *at most one* extension of ρ, ρ' say, such that $\mathit{dom}(\rho') = \mathit{Var}(\Psi, \Phi)$ and $[\![\Psi]\!]\mathcal{I}\rho' = \mathbf{T}$. ■

So a clause is deterministic if the values of the instantiation of any existential variables in the clause are uniquely determined by the instantiations of the universal variables in the clause.

Definition 14.5: Suppose $\mathcal{I}$ and $\mathcal{I}'$ are instances of a schema $\mathcal{S}$, and $f^{\mathcal{C}}$ is a family of injective functions, $f^C : \sigma^C \to \sigma'^C$. Then we define the relations $\preceq_f^\tau \subseteq [\![\tau]\!]\mathcal{I} \times [\![\tau]\!]\mathcal{I}'$ to be the smallest relations such that $p \preceq_f^\tau p'$ if

- $\tau \equiv C$ and $p' = f^C p$, or
- $\tau \equiv \underline{b}$ and $p' = p$, or
- $\tau \equiv (a_1 : \tau_1, \ldots, a_k : \tau_k)$ and $p(a_i) \preceq_f^{\tau_i} p'(a_i)$ for $i = 1, \ldots, k$, or
- $\tau \equiv \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ and $p \equiv (a_i, q)$, $p' \equiv (a_i, q')$ and $q \preceq_f^{\tau_i} q'$, or
- $\tau \equiv \{\tau'\}$ and for each $q \in p$ there is a $q' \in p'$ such that $q \preceq_f^{\tau'} q'$.

We write $\mathcal{I} \preceq_f \mathcal{I}'$ iff, for each $C \in \mathcal{C}$, each $o \in \sigma^C$, $\mathcal{V}^C(o) \preceq_f^{\tau^C} \mathcal{V}'^C(f^C o)$.

We write $\mathcal{I} \preceq \mathcal{I}'$ and say that $\mathcal{I}$ is *smaller* than $\mathcal{I}'$ iff there is a family of injective functions, $f^{\mathcal{C}}$, such that $\mathcal{I} \preceq_f \mathcal{I}'$. ■

The relation $\preceq$ could be thought of as a generalized subset relation, allowing for the renaming of object identities.

Lemma 14.1: If $\mathcal{I}$, $\mathcal{I}'$ and $\mathcal{I}''$ are instances of $\mathcal{S}$ then

1. $\mathcal{I} \preceq \mathcal{I}$ — the relation $\preceq$ is reflexive;
2. if $\mathcal{I} \preceq \mathcal{I}'$ and $\mathcal{I}' \preceq \mathcal{I}''$ then $\mathcal{I} \preceq \mathcal{I}''$ — $\preceq$ is transitive; and
3. if $\mathcal{I} \preceq \mathcal{I}'$ and $\mathcal{I}' \preceq \mathcal{I}$ then $\mathcal{I} \cong \mathcal{I}'$.

■

Proof: It is clear that $\preceq$ is both reflexive and transitive. The third condition requires proof however.

Suppose $\mathcal{I}$ and $\mathcal{I}'$ are instances such that $\mathcal{I} \preceq \mathcal{I}'$ and $\mathcal{I}' \preceq \mathcal{I}$, say $\mathcal{I} \preceq_f \mathcal{I}'$ and $\mathcal{I}' \preceq_g \mathcal{I}$. Then, for each class $C \in \mathcal{C}$, $g^C \circ f^C$ is an injective and hence bijective function from σ^C to σ^C, and similarly $f^C \circ g^C$ is a bijection from σ'^C to itself. Hence f^C and g^C are themselves bijections.

For each $C \in \mathcal{C}$, since σ^C is finite, it follows that there is an integer k^C such that $(g^C \circ f^C)^{k^C} = Id_{\sigma^C}$, and similarly a k'^C such that $(f^C \circ g^C)^{k'^C} = Id_{\sigma'^C}$. Take $k = \prod_{C \in \mathcal{C}} (k^C \times k'^C)$. Then for each $C \in \mathcal{C}$, $(g^C \circ f^C)^k = Id_{\sigma^C}$ and $(f^C \circ g^C)^k = Id_{\sigma'^C}$. For each $C \in \mathcal{C}$ define $f'^C \equiv (g^C \circ f^C)^{k-1} \circ g^C$. Then $f'^C \circ f^C = Id_{\sigma^C}$ and $f^C \circ f'^C = Id_{\sigma'^C}$.

We can show by induction on types that, if $u \in [\![\tau]\!]\mathcal{I}$, $v \in [\![\tau]\!]\mathcal{I}'$ are such that $u \preceq_f^\tau v$ and $v \preceq_{f'}^\tau u$, then $v = f^\tau(u)$ and $u = f'^\tau(v)$.

By transitivity of $\preceq$, we have, for each $C \in \mathcal{C}$, $o \in \sigma'^C$, $\mathcal{V}'^C(o) \preceq_{f'} \mathcal{V}^C(f'^C(o))$. Hence, for every $o \in \sigma^C$, $f^{\tau^C}(\mathcal{V}^C(o)) = \mathcal{V}'^C(f^C(o))$ and for every $o' \in \sigma'^C$, $f'^{\tau^C}(\mathcal{V}'^C(o')) = \mathcal{V}^{\tau^C}(f'^C(o))$.

Hence $\mathcal{I} \cong \mathcal{I}'$. ■

Definition 14.6: We say that a transformation program, **Tr**, is **complete** iff every clause in **Tr** is deterministic with respect to **Tr** and, for any instance $\mathcal{I}_{Src}$ of $\mathcal{S}_{Src}$, if there is a **Tr**-transformation of $\mathcal{I}_{Src}$ then there is a unique (up to isomorphism) smallest such **Tr**-transformation. That is, if $\mathcal{I}_{Src}$ has a **Tr** transformation, then there is an $\mathcal{I}_{Tgt}$ such that $\mathcal{I}_{Tgt}$ is a **Tr**-transformation of $\mathcal{I}_{Src}$ and, for any **Tr**-transformation $\mathcal{I}'$ of $\mathcal{I}_{Src}$, $\mathcal{I}_{Tgt} \preceq \mathcal{I}'$. ■

We are therefore interested in complete transformation programs, and in computing these unique smallest transformations.

Example 14.3: We will describe the transformation specified by the transformation program in example 14.2 on the instance of $\mathcal{S}_{Src}$ defined in example 7.2.

We will take our object-identities to be:

$$\begin{aligned} \sigma^{City_2} &\equiv \{Phila', Pitts', Harris', NYC', Albany'\} \\ \sigma^{State_2} &\equiv \{PA', NY'\} \end{aligned}$$

(the choice of object identities is arbitrary since transformations are defined up to isomorphism only). Our mappings are

$$\begin{aligned} \mathcal{V}^{City_2}(Phila') &\equiv (name \mapsto \text{“Philadelphia”}, is\_capital \mapsto (not\_cap, \emptyset)) \\ \mathcal{V}^{City_2}(Pitts') &\equiv (name \mapsto \text{“Pittsburgh”}, is\_capital \mapsto (not\_cap, \emptyset)) \\ \mathcal{V}^{City_2}(Harris') &\equiv (name \mapsto \text{“Harrisburg”}, is\_capital \mapsto (cap\_of, PA')) \\ \mathcal{V}^{City_2}(NYC') &\equiv (name \mapsto \text{“New York City”}, is\_capital \mapsto (not\_cap, \emptyset)) \\ \mathcal{V}^{City_2}(Albany') &\equiv (name \mapsto \text{“Albany”}, is\_capital \mapsto (cap\_of, NY')) \end{aligned}$$

and

$$\begin{aligned} \mathcal{V}^{State_2}(PA') &\equiv (name \mapsto \text{“Pennsylvania”}, cities \mapsto \{Phila', Pitts', Harris'\}) \\ \mathcal{V}^{State_2}(NY') &\equiv (name \mapsto \text{“New York”}, cities \mapsto \{NYC', Albany'\}) \end{aligned}$$

■

14.4 Normal Forms of Transformation Programs

In this section we will define a *normal form* for transformation clauses. A transformation clause in normal form will completely define an insert into the target database in terms of the source database only. That is, a normal form clause will contain no target terms in its body, and will completely and unambiguously determine some element of the target database in its head. Given a transformation program in which all the clauses are in normal form, the transformation may then be easily implemented as a single pass transformation in some suitable DBPL.

Our objective will be to determine certain syntactic constraints on transformation programs (*non-recursion*), such that any *complete* transformation program satisfying these constraints can be converted to an equivalent program in which all the clauses are in normal form.

Term Paths

We introduce the concept of *term paths* in order to reason about which parts of a database a clause will access. Every term in a well-formed set of atoms will have at least one term-path associated with it, representing the part of the database instance where it may be found. Term paths therefore represent a way of navigating a database, starting at some class and then following a series or attribute labels, dereferences and set inclusions.

The concept of term paths will be useful in a number of places, when trying to reason about the information accessed or implied by clauses. We will use them in order to formalize the concept of a formula "unambiguously" determining part of a clause.

Definition 14.7: A **term path** for some schema $\mathcal{S}$ is a pair (C, μ) where $C \in \mathcal{C}$ and μ is a string over the alphabet

$$\{\pi_a | a \in \mathcal{A}\} \cup \{ins_a | a \in \mathcal{A}\} \cup \{!,\ \dot{\in}\}$$

■

We define the typing relation $\vdash:$ on term paths by the rules:

$$\frac{}{\vdash (C, \epsilon) : \{C\}} \qquad \frac{\vdash (C, \mu) : \{\tau\}}{\vdash (C, \mu.\dot{\in}) : \tau} \qquad \frac{\vdash (C, \mu) : D,\ D \in \mathcal{C}}{\vdash (C, \mu.!) : \tau^D}$$

$$\frac{\vdash (C, \mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)}{\vdash (C, \mu.\pi_{a_i}) : \tau_i} \qquad \frac{\vdash (C, \mu) : \langle\!|a_1 : \tau_1, \ldots, a_k : \tau_k|\!\rangle}{\vdash (C, \mu.ins_{a_i}) : \tau_i}$$

Note that the type associated with a term path is dependent only on the term path itself, and not on a type context or the atoms from which the term path arises, or any other influence.

We write $TPaths^{\mathcal{S}}$ for the set of term paths, (C, μ) of a schema $\mathcal{S}$ such that $\vdash (C, \mu) : \tau$ for some type τ.

We define the relationship $\Phi \vdash:$ between the term occurrences of Φ, $Occ(\Phi)$, and term paths, $TPaths^{\mathcal{S}}$, to be the smallest relationship such that, for any occurrence w of a term P in Φ and term path (C, μ), $\Phi \vdash w : (C, \mu)$ if

1. $P \equiv C$ and $\mu = \epsilon$ (the empty string), or
2. $P \equiv !Q$, $w = (\phi, (i, j))$ and $\mu = \mu'.!$ and $\Phi \vdash (\phi, (i, j+1)) : (C, \mu')$, or
3. $P \equiv \pi_a Q$, $w = (\phi, (i, j))$ and $\mu = \mu'.\pi_a$ and $\Phi \vdash (\phi, (i, j+1)) : (C, \mu')$, or
4. $w = (\phi, (i, j+1))$ and $w' = (\phi, (i, j))$ is an occurrence of a term $Q \equiv ins_a(P)$ in Φ, and $\mu = \mu'.ins_a$ and $\Phi \vdash w' : (C, \mu')$, or
5. $w = (\phi, (1, 0))$ where $\phi \equiv (P \dot{\in} Q)$, and $\mu = \mu'.\dot{\in}$, and $\Phi \vdash (\phi, (2, 0)) : (C, \mu')$, or
6. $w = (\phi, (i, 0))$ where $\phi \equiv (P \doteq Q)$ and $i = 1$ and $\Phi \vdash (\phi, (2, 0)) : (C, \mu)$, or $\phi \equiv (Q \doteq P)$ and $i = 2$ and $\Phi \vdash (\phi, (1, 0)) : (C, \mu)$.

We write $\Phi \vdash P : (C, \mu)$ to mean $\Phi \vdash w : (C, \mu)$ for some occurrence w of P in Φ.

Example 14.4: Consider the set of atoms

$$\begin{aligned}\Phi \ \equiv \ & W \in \mathit{Y.cities}, \mathit{W.name} = N, Z \in \mathit{City}_A, \mathit{Z.name} = N, \\ & Y \in \mathit{State}_2, \mathit{Y.name} = \mathit{Z.state.name}\end{aligned}$$

for the schema of the previous examples. Here the term N has two term paths: $\Phi \vdash N : (\mathit{City}_A, \dot{\in}!\pi_{name})$ and $\Phi \vdash N : (\mathit{State}_2, \dot{\in}!\pi_{cities}\dot{\in}!\pi_{\mathrm{name}})$. ■

The following lemma tell us that the typing rules for term paths and for term occurrences in a well-formed set of atoms coincide, and further, that for any term path for which the typing rules assign a type, there is a set of atoms which include a term with that term path.

Lemma 14.2:

1. If Φ is a well-formed set of atoms and w a term occurrence in Φ such that $\Phi \vdash w : (C, \mu)$ in Φ and $\Phi \vdash w : \tau$ then $\vdash (C, \mu) : \tau$.
2. If (C, μ) is a term path and τ a type, such that $\vdash (C, \mu) : \tau$, then there is a well-formed set of atoms, Φ, and a term P occurring in Φ such that $\Phi \vdash P : (C, \mu)$. Further there is a *unique* (up to variable renaming) smallest such set of semi-normal form atoms, Φ, such that $\Phi \vdash X : (C, \mu)$ for some $X \in \mathit{Var}(\Phi)$, and for any set of snf atoms Φ' such that Φ' does not contain any atoms of the form $Z \doteq Y$ and $\Phi' \vdash X : (C, \mu)$ for some $X \in \mathit{Var}(\Phi')$, there is a subset $\Phi'' \subseteq \Phi'$ such that Φ' is a variable renaming of Φ.

■

Proof: The first part follows by a simple induction on the proof that $\Phi \vdash P : (C, \mu)$.

For the second part, we will proceed by induction on the length of μ. For any term path (C, μ) we will construct a set of snf atoms $\mathit{Nav}^X_{(C,\mu)}$ such that

1. $\mathit{Nav}^X_{(C,\mu)} \vdash X : (C, \mu)$,
2. for any set of snf atoms Φ if $|\Phi| \leq |\mathit{Nav}^X_{(C,\mu)}|$ and $\Phi \vdash Y : (C, \mu)$ for some $Y \in \mathit{Var}(\Phi)$ then Φ can be formed from $\mathit{Nav}^X_{(C,\mu)}$ by renaming variables, and
3. for any set of snf atoms Φ not containing equality atoms on variables, if $\Phi \vdash Y : (C, \mu)$ for some $Y \in \mathit{Var}(\Phi)$, then Φ contains a set of atoms Φ' which can be formed from $\mathit{Nav}^X_{(C,\mu)}$ by renaming variables.

We will show the base case and one of the induction cases. The other induction cases are all similar.

For a term path (C, ϵ), consider the set of atoms $\mathit{Nav}^X_{(C,\epsilon)} = \{(X \doteq C)\}$. Then $\mathit{Nav}^X_{(C,\epsilon)} \vdash X : (C, \epsilon)$ and it is clear that there is no smaller set of snf atoms containing a term with type path (C, ϵ).

Further any set of semi-normal form atoms containing a term occurrence with type path (C, ϵ) must contain an atom of the form $Y \doteq C$ for some Y.

Suppose $\mu \equiv \mu' \pi_a$. Consider the set of atoms $Nav^X_{(C,\mu)} = Nav^Y_{(C,\mu')} \cup \{X \doteq \pi_a Y\}$, for some variable Y. Then $Nav^X_{(C,\mu)} \vdash X : (C, \mu)$ since $Nav^Y_{(C,\mu')} \vdash Y : (C, \mu')$. Suppose Φ is a set of snf atoms such that $|\Phi| \leq |Nav^X_{(C,\mu)}|$, and there is an occurrence of a term Y in Φ such that $\Phi \vdash Y : (C, \mu)$ for some $Y \in Var(\Phi)$. Observe from the construction of term paths that Φ must contain an atom of the form $Z \doteq \pi_a W$ in Φ. Further there exists a set of atoms $\Phi' \subseteq \Phi \setminus \{(Z \doteq \pi_a W)\}$ such that $\Phi' \vdash W : (C, \mu')$. But $|\Phi'| \leq |Nav^Y_{(C,\mu')}|$. Hence, by our induction hypothesis, Φ' can be formed from $Nav^Y_{(C,\mu')}$ by renaming variables. Hence $\Phi' \cup \{Z \doteq \pi_a W\} \subseteq \Phi$ can be formed from $Nav^X_{(C,\mu)}$ by renaming variables. Now suppose Φ is a set of snf atoms containing no equality atoms on variables, and $\Phi \vdash Z : (C, \mu)$. Observe from the construction of term paths that Φ must contain an atom $Z \doteq \pi_a W$ where $\Phi \vdash W : (C, \mu')$. Then, by our induction hypothesis, Φ must contain a subset of atoms Φ' such that $\Phi' \vdash W : (C, \mu')$ and Φ' can be formed by renaming variables in $Nav^Y_{(C,\mu')}$. Then the set of atoms $\Phi' \cup \{Z \doteq \pi_a W\} \subseteq \Phi$ can be formed from $Nav^X_{(C,\mu)}$ by renaming variables.

The other induction cases are similar. ∎

From this lemma we can get the intuition that a term path corresponds to the subset of a set of atoms necessary in order to navigate to a particular location in the database.

Characterizing Formulae

We introduce the concept of *characterizing formula* as a means of uniquely characterizing a particular element of a database. Notice that this is more specific than a term-path, which characterizes a *place* in the database. In particular, if a value occurs in some set in a database instance, then its term path will not characterize which element of the set it is.

Characterizing formulae are introduced as an attempt to generalize the notion of *keys* to non-class types, and, in particular, to deal with nested set data-types. In section 14.5 we will see that things can be simplified considerably if we avoid nested set types.

Definition 14.8: Suppose **Tc** is a set of clauses, Θ is a finite set of snf atoms, and $T, S \subseteq Var(\Theta)$ are sets of variables. We say that T is **characterized** by S in Θ, **Tc** iff for any instance $\mathcal{I}$ satisfying each clause in **Tc**, and any $\mathcal{I}$-environments ρ and ρ', if $[\![\Theta]\!]\mathcal{I}\rho = \mathbf{T}$ and $[\![\Theta]\!]\mathcal{I}\rho' = \mathbf{T}$, then $\rho(X) = \rho'(X)$ for each $X \in T$ iff $\rho(Y) = \rho'(Y)$ for each $Y \in S$.

In other words, if ρ is an environment satisfying Θ, then the values of ρ on T are uniquely determined by the values of ρ on S and vice versa. ∎

Definition 14.9: Suppose **Tc** is a set of clauses, and (C, μ) a term path. A **characterizing formula** for (C, μ) in **Tc** is a set of atoms Θ together with a set of variables $\{Y_1, \ldots, Y_n\} \subseteq Var(\Theta)$ and a distinguished variable $X \in Var(\Theta)$ such that

1. $Nav^X_{(C,\mu)} \subseteq \Theta$

2. The set of variables $Var(Nav^X_{(C,\mu)})$ is characterized by $\{Y_1, \ldots, Y_n\}$ in Θ, **Tc**.

We write $\Theta^X(Y_1, \ldots, Y_n)$ for the characterizing formula for some type path (C, μ) with Θ as its set of atoms, $\{Y_1, \ldots, Y_n\}$ as its set of characterizing variables and distinguished variable X. ■

In other words, a characterizing formula, together with an instantiation of its variables, will uniquely characterize a particular element of a database.

It is clear that there are some trivial characterizing formula, such as $\Theta^X(Y_1, \ldots, Y_N)$ where $\Theta = Nav^X_{(C,\mu)}$ and $\{Y_1, \ldots, Y_N\} = Var(\Theta)$. However there are also some more useful examples of characterizing formula.

Example 14.5: A characterizing formula for the term path $(State', \dot{\in})$ in the transformation program of example 14.2 would be

$$\Theta^X(N) \;\equiv\; X \in State_2,\, \pi^{name}(!X) = N$$

since the program implies the target constraint

$$X = Y \;\Longleftarrow\; X \in State_2,\, Y \in State_2,\, \pi^{name}(!X) = N,\, \pi^{name}(!Y) = N$$

A characterizing formula for the term path $(City', \dot{\in})$ would be

$$\Theta'^X(N, Z) \;\equiv\; X \in City_2,\, \pi^{name}(!X) = N,\, Z \in State_2,\, X \in \pi^{cities}(!Z)$$

■

The following lemma tells us that the characterizing formulas for any other target terms paths could be formed from these two.

Lemma 14.3:

1. If $\vdash\, (C, \mu) \,:\, (a_1 \,:\, \tau_1, \ldots, a_k \,:\, \tau_k)$, $\Theta^Y(V)$ is a characterizing formula for (C, μ) and $\Theta' \equiv (\Theta \cup \{X \doteq \pi_{a_i} Y\})$, then $\Theta'^X(V)$ is a characterizing formula for $(C, \mu.\pi_{a_i})$.

2. If $\vdash\, (C, \mu) \,:\, \langle\!| a_1 \,:\, \tau_1, \ldots, a_k \,:\, \tau_k |\!\rangle$, $\Theta^Y(V)$ is a characterizing formula for (C, μ) and $\Theta' \equiv (\Theta \cup \{Y \doteq ins_{a_i} X\})$, then $\Theta'^X(V)$ is a characterizing formula for $(C, \mu.ins_{a_i})$.

3. If $\vdash\, (C, \mu) \,:\, D$, $D \in \mathcal{C}$, $\Theta^Y(V)$ is a characterizing formula for (C, μ) and $\Theta' \equiv (\Theta \cup \{X \doteq !Y\})$, then $\Theta'^X(V)$ is a characterizing formula for $(C, \mu.!)$.

■

Proof: It is sufficient to observe from the semantics of atoms, that for each of the atoms $(X \doteq \pi_a Y)$, $(Y \doteq ins_a X)$ and $(X \doteq !Y)$, for any instance $\mathcal{I}$ and instantiation of the variable Y, there is at most one instantiation of the variable X which makes the atom true. ■

Consequently, if we are describing a database instance, or a transformation, we only need characterization formulae for paths of the form $(C, \mu.\dot{\in})$.

In addition, in the case of a keyed schema, for each class $C \in \mathcal{C}$ the formula

$$\Theta_X(Z) \equiv \{X \in Y, Y = C, X = Mk^C(Z))$$

is a characterization formula for the term path $(C, \dot{\in})$. This gives us a useful special case: if every set type occurring in a schema $\mathcal{S}$ is of the form $\{C\}$, that is a set of values of class type, and we have a key specification on $\mathcal{S}$, then we can automatically find useful characterization formulas for any term path in $\mathcal{S}$.

In fact the only cases where one cannot automatically generate useful characterizing formulae is when our schema involves *nested set types*: a nested set type is a type of the form $\{\tau\}$ where τ involves some type of the form $\{\tau'\}$. In such cases it is necessary to ensure that there are constraints which allow for the construction of characterizing formulae for any term paths of the form $(C, \mu\dot{\in}) \in \mathit{TPaths}^{\mathcal{S}}$. In section 14.5 we will show how these notions can be simplified, and a system of characterizing formulae can be automatically generated, in the case where we do not allow nested set types.

In the following definitions we use characterizing formulae in order to get an analog of the notion of keys for sets of non-object identity values.

Definition 14.10: A **characterizing system** for a set of clauses **Tc**, consists of a finite set of term paths, $\mathcal{T} \subseteq \mathit{TPaths}^{\mathcal{S}}$, such that for each $(C, \mu) \in \mathcal{T}$ the symbol ! occurs at most once in μ, and for each term path $(C, \mu) \in \mathcal{T}$ a set of characterizing formula for (C, μ), $\mathcal{F}^{(C,\mu)}$, such that, for any instance $\mathcal{I}$, term path $(C, \mu) \in \mathcal{T}$ and environment ρ such that $[\![Nav^X_{(C,\mu)}]\!]\mathcal{I}\rho = \mathbf{T}$, there is at most one characterizing formula $\Theta^X(Y_1, \ldots, Y_n) \in \mathcal{F}_{(C,\mu)}$ such that $[\![\Theta]\!]\mathcal{I}\rho' = \mathbf{T}$ for some extension ρ' of ρ. ■

So a characterizing system provides a means of uniquely identifying values of certain type paths in an instance. The uniqueness conditions in the definition ensure that there are never multiple characterizing formula in $\mathcal{F}^{(C,\mu)}$ for some (C, μ) characterizing the same element of an instance. Recall that, if $\Theta^X(Y_1, \ldots, Y_n)$ is a characterizing formula for (C, μ) then $Nav^X_{(C,\mu)} \subseteq \Theta$.

The *graph* of a characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$, written $G(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$, is a digraph such that

1. the nodes of $G(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ are the term paths $\mathit{TPaths}^{\mathcal{S}}$, and
2. $G(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ contains an edge $(C', \mu') \rightarrow (C, \mu)$ iff $(C, \mu) \in \mathcal{T}$ and there is a characterizing formula $\Theta^X(Y_1, \ldots, Y_n) \in \mathcal{F}^{(C,\mu)}$ such that $\Theta \vdash Y_i : (C', \mu')$ for some $i \in 1, \ldots, n$ (where Θ is the set of atoms of the characterizing formula $\Theta^X(Y_1, \ldots, Y_n)$).

A characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ is **acyclic** if the graph $G(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ is acyclic.

A transformation clause $\Psi \Longleftarrow \Phi$ is said to **respect** a characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ iff for every variable $X \in \mathit{Var}(\Psi \cup \Phi)$ either $X \in \mathit{Var}(\Phi)$ or there is a type path $(C, \mu) \in \mathcal{T}$ such that $\Psi \cup \Phi \vdash X : (C, \mu)$ and $\Theta \subseteq \Psi \cup \Phi$ for some characterizing formula $\Theta^X(Y_1, \ldots, Y_n) \in \mathcal{F}^{(C,\mu)}$.

Normal Forms

Definition 14.11: Suppose that **Tr** is a transformation program, with target constraints **Tc**, and $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ is an acyclic characterizing system for **Tc**.

A clause $\Psi \Longleftarrow \Phi$ is in **normal form** for **Tr** and $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ iff

1. $\Psi \Longleftarrow \Phi$ is in semi-normal form;
2. Φ contains no target terms;
3. $\Psi \Longleftarrow \Phi$ respects $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$;
4. If $\Phi \cup \Psi \vdash X : (C, \mu)$ for some X and some target term path (C, μ) and $\vdash (C, \mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$, then for each a_i, $\Phi \cup \Psi$ contains an atom $(Y \doteq \pi_{a_i} X)$ for some $Y \in \mathit{Var}(\Phi \cup \Psi)$;
5. If $\Phi \cup \Psi \vdash X : (C, \mu)$ for some X and some target term path (C, μ) and $\vdash (C, \mu) : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$, then $\Phi \cup \Psi$ contains an atom $\mathit{ins}_{a_i} Y \doteq X$ for some $Y \in \mathit{Var}(\Phi \cup \Psi)$ and some a_i;
6. If $\Phi \cup \Psi \vdash X : (C, \mu)$ for some X and some target path (C, μ), and $\vdash (C, \mu) : \underline{b}$, then either $X \in \mathit{Var}(\Phi)$ or $\Psi \cup \Phi$ contains an atom $X \doteq c^{\underline{b}}$ for some constant symbol $c^{\underline{b}}$.

■

The first requirement, that the clause be in snf, serves to simplify the later requirements. The second requirement implies that the body of the clause can be evaluated by looking at the source databases only, and hence the clause can be applied in a single-pass, non-recursive manner.

The remaining requirements ensure that the head of the clause uniquely and unambiguously determines some part of the target database. In particular, 3 implies that every variable in the head of the clause is characterized by the universal variables of the clause. Note that we don't have a requirement similar to the fourth and fifth requirements for set types: that is, we don't require that, if $\Psi \Longleftarrow \Phi$ contains a term of some path type (C, μ), where $\vdash (C, \mu) : \{\tau\}$, then the clause must specify the contents of the set. This is because we are looking at unique smallest transformations, so the set is uniquely determined anyway: if no elements of a term of set type are specified then the resulting set will be empty.

Example 14.6: The following are normal-form clauses equivalent to the transformation clauses of example 14.2:

$$Y \in \mathit{State}', W =!Y, N = \pi_{\mathit{name}} W \impliedby X \in \mathit{State}, U =!X, N = \pi_{\mathit{name}} U$$

$$\begin{aligned} & Y \in \mathit{State}', W \in \mathit{City}', U =!Y, N = \pi_{\mathit{name}} U, V =!W, C = \pi_{\mathit{name}} V, \\ & \qquad T = \pi_{\mathit{iscap}} V, T = \mathit{ins}_{\mathit{yes}}(), S = \pi_{\mathit{cites}} U, W \in S \\ & \impliedby X \in \mathit{State}, Q =!X, N = \pi_{\mathit{name}} Q, Z \in \mathit{City}, R =!Z, C = \pi_{\mathit{name}} R, \\ & \qquad Z = \pi_{\mathit{state}} Q, X = \pi_{\mathit{capital}} R \end{aligned}$$

$$\begin{aligned} & Y \in \mathit{State}', W \in \mathit{City}', U =!Y, N = \pi_{\mathit{name}} U, V =!W, C = \pi_{\mathit{name}} V, \\ & \qquad T = \pi_{\mathit{iscap}} V, T = \mathit{ins}_{\mathit{no}}(), S = \pi_{\mathit{cites}} U, W \in S \\ & \impliedby X \in \mathit{State}, Q =!X, N = \pi_{\mathit{name}} Q, Z \in \mathit{City}, R =!Z, C = \pi_{\mathit{name}} R, \\ & \qquad Z = \pi_{\mathit{state}} Q, O = \pi_{\mathit{capital}} R, X \neq O \end{aligned}$$

■

Proposition 14.4: Suppose **Tr** is a transformation program consisting of source constraints and transformation clauses, such that for some characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$, every transformation clause in **Tr** is in normal form for $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ and **Tr**. Suppose further that, for any instance $\mathcal{I}$ satisfying **Tr**, target class $C \in \mathcal{C}_{Tgt}$, transformation clause $\Delta = (\Psi \impliedby \Phi)$ in **Tr** such that $\Psi \cup \Phi \vdash X : C$ for some $X \in \mathit{Var}(\Psi)$, and environment ρ such that $[\![\Psi \cup \Phi]\!]\mathcal{I}\rho = \mathbf{T}$, there is a transformation clause $\Delta' = (\Psi' \impliedby \Phi')$, environment ρ' and variable $Y \in \mathit{Var}(\Psi')$ such that $[\![\Psi' \cup \Phi']\!]\mathcal{I}\rho' = \mathbf{T}$, $\rho'(Y) = \rho(X)$ and Ψ contains an atom of the form $Z \doteq !Y$. Then **Tr** is complete. ■

Proof: First we must show that, if a clause $\Psi \impliedby \Phi$ is in normal form for some characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$, then $\Psi \impliedby \Phi$ is deterministic. Suppose that $\mathcal{I}$ is an instance and ρ an environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Suppose that ρ' ane ρ'' are extensions of ρ such that $[\![\Psi]\!]\mathcal{I}\rho' = \mathbf{T}$ and $[\![\Psi]\!]\mathcal{I}\rho'' = \mathbf{T}$. We need to show that, for every $X \in \mathit{Var}(\Psi \cup \Phi)$, $\rho'(X) = \rho''(X)$. If $X \in \mathit{Var}(\Phi)$ the result is clear. If $X \in \mathit{Var}(\Psi) \setminus \mathit{Var}(\Phi)$ then $\Theta^X_{(C,\mu)}(Y_1, \ldots, Y_n) \subseteq \Psi \cup \Phi$ for some $(C, \mu) \in T$ such that $\Psi \cup \Phi \vdash X : (C, \mu)$. The proof then proceeds by induction of the length of paths in $G(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ originating from (C, μ).

Next, suppose we have a transformation program **Tr** and characterizing system $(\mathcal{T}, \mathcal{F}^{\mathcal{T}})$ as described in the proposition. Suppose $\mathcal{I}_{Src}$ is an instance of $\mathcal{S}_{Src}$ such that there is an instance $\mathcal{I}_{Tgt}$ of $\mathcal{S}_{Tgt}$ with $\mathcal{I} = \mathcal{I}_{Src} \cup \mathcal{I}_{Tgt}$ satisfying every clause in **Tr**.

We will first prove that we can build a minimal instance satisfying **Tr** by extracting those parts of the instance $\mathcal{I}_{Tgt}$ that are actually implied by the transformation program.

For each each characterizing formula $\Theta^X(Y_1, \ldots, Y_k) \in \mathcal{F}^{(C,\mu)}$, $(C, \mu) \in \mathcal{T}$, let us assume a function symbol $F^{\Theta}_{(C,\mu)}$ of arity k. Let **Alg** be the freely generated algebra with values and objects occurring in $\mathcal{I}_{Src}$ as its base values, and function symbols $F^{\Theta}_{(C,\mu)}$.

Suppose $\Delta \equiv (\Psi \impliedby \Phi)$ is a transformation clause in **Tr**, and ρ is an $\mathcal{I}$-environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Then, for each $X \in \mathit{Var}(\Psi \cup \Phi)$ we assign a term of the algebra **Alg**, say $\mathit{Alg}^{\rho}(X)$, defined by

$$Alg^{\rho}(X) \equiv \begin{cases} \rho(X) & \text{if } X \in \mathit{Var}(\Phi) \\ F^{\Theta}_{(C,\mu)}(Alg^{\rho}(Y_1), \ldots, Alg^{\rho}(Y_n)) & \text{if } X \notin \mathit{Var}(\Phi) \\ & \text{and } \Theta^X(Y_1, \ldots, Y_n) \in \Psi \cup \Phi \end{cases}$$

(Note that, if $X \in \Phi$ then X is a source term, and so $\rho(X)$ is a value occurring in $\mathcal{I}_{Src}$).

Next we define $\mathit{Assign} : \mathbf{Alg} \overset{\sim}{\rightarrow} \mathbf{D}(\mathcal{I})$ to be the smallest mapping (ordered pointwise by $\preceq_{Id}$) such that:

1. $\mathit{Assign}(u) = u$ for $u \in \mathbf{D}(\mathcal{I})$
2. If $\Delta \equiv (\Psi \Longleftarrow \Phi)$ is a transformation clause in $\mathbf{Tr}$, and ρ is an $\mathcal{I}$-environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$, and ρ' is the (unique) extension of ρ such that $dom(\rho') = \mathit{Var}(\Psi \cup \Phi)$ and $[\![\Psi]\!]\mathcal{I}\rho' = \mathbf{T}$, then for any $X \in \mathit{Var}(\Psi) \setminus \mathit{Var}(\Phi)$
   (a) If $\Psi \cup \Phi \vdash X : C$, $C \in \mathcal{C}_{Tgt}$, then $\mathit{Assign}(Alg^{\rho}(X)) = \rho'(X)$,
   (b) If $\Psi \cup \Phi \vdash X : (a_1 : \tau_1, \ldots, a_n : \tau_n)$ and $\Psi \cup \Phi$ contains the atoms $Y_i \doteq \pi_{a_i} X$ for $i = 1, \ldots, n$, then $\mathit{Assign}(Alg^{\rho}(X)) = (a_1 \mapsto \mathit{Assign}(Alg^{\rho}(Y_1)), \ldots, a_n \mapsto \mathit{Assign}(Alg^{\rho}(Y_n))$,
   (c) If $\Psi \cup \Phi \vdash X : \langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle$ and $\Psi \cup \Phi$ contains an atom $X \doteq ins_{a_i} Y$ then $\mathit{Assign}(Alg^{\rho}(X)) = (a_i, \mathit{Assign}(Alg^{\rho}(Y))$,
   (d) If $\Psi \cup \Phi \vdash X : \underline{b}$ and $\Psi \cup \Phi$ contains the atom $X \doteq c^{\underline{b}}$ then $\mathit{Assign}(Alg^{\rho}(X)) = \bar{c}$,
   (e) If $\Psi \cup \Phi \vdash X : \{\tau\}$ and $\Psi \cup \Phi$ contains the atom $Y \dot{\in} X$, then $\mathit{Assign}(Alg^{\rho}(Y)) \in \mathit{Assign}(Alg^{\rho}(X))$.

Now, for each class $C \in \mathcal{C}_{Tgt}$ form the set of object identities $\sigma'^C \subseteq \sigma^C$ defined by

$$\sigma'^C \equiv \left\{ o \in \sigma^C \;\middle|\; \begin{array}{l} o = \rho(X) \text{ for some } (\Psi \Longleftarrow \Phi) \in \mathbf{Tr}, \\ X \in \mathit{Var}(\Psi) \text{ and } \rho \text{ such that } [\![\Psi \cup \Phi]\!]\mathcal{I}\rho = \mathbf{T} \end{array} \right\}$$

Define $\mathcal{V}'^C : \sigma'^C \rightarrow [\![\tau^C]\!]\sigma'^{\mathcal{C}_{Tgt}}$ to be such that, if $\Delta \equiv (\Psi \Longleftarrow \Phi)$ is a transformation clause in $\mathbf{Tr}$, ρ is such that $[\![\Psi \cup \Phi]\!]\mathcal{I}\rho = \mathbf{T}$, and $\Psi \cup \Phi$ contains the atom $Y \doteq !X$ where $\Psi \cup \Phi \vdash X : C$, then $\mathcal{V}'^C(\rho(X)) = \mathit{Assign}(Alg^{\rho} Y)$.

Then $\mathcal{I}'_{Tgt} = (\sigma'^{\mathcal{C}_{Tgt}}, \mathcal{V}'^{\mathcal{C}_{Tgt}})$ is an instance of $\mathcal{S}_{Tgt}$, and satisfies $\mathbf{Tr}$.

Finally it is necessary to show that $\mathcal{I}'_{Tgt}$ is smaller than any other $\mathbf{Tr}$-transformation of $\mathcal{I}_{Src}$.

Suppose $\mathcal{I}''_{Tgt}$ is an instance of $\mathcal{S}_{Tgt}$ such that $\mathcal{I}''_{Tgt} \cup \mathcal{I}_{Src}$ satisfies every clause in $\mathbf{Tr}$. For any $C \in \mathcal{C}$ and $o \in \sigma'^C$, there is a transformation clause $\Delta \equiv (\Psi \Longleftarrow \Phi)$ an $\mathcal{I}_{Src}$ environment ρ and an $X \in \mathit{Var}(\Psi)$, such that $o = \rho'(X)$ where ρ' is an extension of ρ such that $[\![\Psi \cup \Phi]\!]\mathcal{I}\rho' = \mathbf{T}$. We define $f^C(o) \in \sigma''^C$ to be the (unique) object identity such that $f^C(o) = \rho''(X)$ where ρ'' is an extension of ρ such that $[\![\Psi \cup \Phi]\!](\mathcal{I}''_{Tgt} \cup \mathcal{I}_{Src})\rho'' = \mathbf{T}$, which exists since $\mathcal{I}''_{Tgt} \cup \mathcal{I}_{Src}$ satisfies

Δ. It follows from the definitions of characterizing systems and characterizing formulae that the functions f^C are injective.

If $\Delta \equiv (\Psi \Longleftarrow \Phi)$ is a transformation clause, ρ an environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$ and ρ' the extension of ρ such that $[\![\Psi]\!](\mathcal{I}''_{Tgt} \cup \mathcal{I}_{Src})\rho' = \mathbf{T}$, then we can show by induction on types that, for each $X \in \mathit{Var}(\Psi)$ such that $\Psi \cup \Phi \vdash X : \tau$ $\mathit{Assign}(\mathit{Alg}^{\rho}(X)) \preceq^{\tau}_{f} \rho'(X)$.

Hence we get, for each $C \in \mathcal{C}_{Tgt}$, $o \in \sigma^C$, $\mathcal{V}'^C(o) \preceq^{\tau}_{f} \mathcal{V}''^C(f^C(o))$.

Hence $\mathcal{I}'_{Tgt} \preceq \mathcal{I}''_{Tgt}$ as required. ■

A transformation program in normal form has the advantage that it can be easily evaluated as a single-pass transformation, or translated into a variety of database programming languages, for example using comprehension syntax [14].

Proposition 14.4 tells us that the only way a normal form transformation program can fail to be complete is if it generates some object identities and fails to associate values with them. Though the conditions on transformation programs of proposition 14.4 seem quite complicated, they are in practice often obvious or easy to check. Further, it is straightforward to check these conditions for a transformation of a particular instance, so that one knows that, if a transformation does provide a value associated with every object identity it produces for a particular source database instance, then the resulting target instance is the unique smallest instance satisfying that transformation.

14.5 Simplifying Characterizing Formula

The mechanism of *characterizing formulae* and their use in defining normal form clauses, introduced in section 14.4 seems rather complicated and unintuitive. In general it is not possible to automatically generate characterizing formulae for a term path, or to test whether there exist characterizing formulae which form a characterizing system satisfied by a particular transformation program. Consequently it is worth looking at why characterizing formulae are necessary, and in what situations we can avoid or simplify them.

The purpose of a characterizing formula is to uniquely specify some element to be inserted into a target database. We noted in section 14.4 that if we have characterizing formulae for term paths of the form $(C, \mu. \in)$, then we can automatically generate characterizing formulae for any other term paths. This is the consequence of the fact that, if a term P is of record, variant or class type, then its components, accessed using projection, variant insertion or dereferencing, are uniquely determined by P. This is not the case for terms of set type: if we have a set of atoms Φ containing an atom $X \in Y$, then even if we know exactly what set Y refers to in the database there may still be a choice of possible instantiations for X.

We have already seen that, in the case of a simply keyed schema, $(\mathcal{S}, \mathcal{K})$, it is possible to automatically generate useful characterizing formulae for term paths of the form $(C, \dot{\in})$, namely

$$\Theta^X(Y) \equiv \{X \dot{\in} Z,\ Z \doteq C,\ X \doteq \mathit{Mk}^C(Y)\}$$

A natural question to ask is therefore, what other term paths are there useful, automatically generatable characterizing formula for, and, in particular, what are the problem cases?

It turns out that the most difficult terms to characterize are those involving *nested sets.* A *nested set type* is a type of the form $\{\tau\}$ where τ in turn involves a set type $\{\tau'\}$. To see why such data-types are problematic, let us look at a simple example.

Example 14.7: Suppose we have a source schema with a single class, $\mathcal{C}_{Src} = \{D\}$, and

$$\mathcal{S}_{Src}(D) \equiv (a : int,\, b : \{int\})$$

and a target schema with a single class, $\mathcal{C}_{Tgt} = \{C\}$, and

$$\mathcal{S}_{Tgt}(C) \equiv (a : int,\, b : \{\{int\}\})$$

Suppose that the key specifications of $\mathcal{S}_{Src}$ and $\mathcal{S}_{Tgt}$ are given by $\kappa^C \equiv int$, $\kappa^D \equiv int$, and constraints

$$\begin{array}{rcl} X = Mk^C(X.a) & \Longleftarrow & X \in C \\ Y = Mk^D(Y.a) & \Longleftarrow & Y \in D \end{array}$$

Lets look at transformations from $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$ specified by the clauses:

$$\begin{array}{l} X \in C,\, X.a = Y.a \;\Longleftarrow\; Y \in D \\ X \in \mathcal{C},\, X.a = Y.a,\, W \in X.b,\, Z \in W \;\Longleftarrow\; Y \in D,\, Z \in Y.b \end{array}$$

So for every object in D there is a corresponding object in C with the same value on its a attribute. Further for every integer in the b attribute of an object in D, the integer is also in one of the sets in the b attribute of the corresponding object in C. It is clear that there is no unique smallest transformation of a general instance of $\mathcal{S}_{Src}$.

The problem is that there is nothing to tell us which of the sets in the b attribute of an object in C should contain a particular integer: we need more constraints. Further there is no obvious choice of constraint to use. There are various possibilities: $X.b$ should always be a singleton set, or should be a set of singleton sets, or should consist of two sets of even and odd numbers, and so on. None of these constraints seem particularly natural, and proving that one of them sufficient to characterize a term with type path $(C, \in .!.\pi_b. \in)$ is potentially difficult. ■

Suppose we limit ourselves to dealing with target databases which do not involve nested set types. Then can we automatically find useful characterizing formulae? In this case it turns out that we can.

A schema $\mathcal{S}$ with classes $\mathcal{C}$ is said to be **non-nested** iff, for each class $C \in \mathcal{C}$, the type $\mathcal{S}(C)$ contains no nested-set types.

Suppose we have source and target simply-keyed schemas, $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ and $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$, and $\mathcal{S}_{Tgt}$ is a non-nested schema. For any target type path (C, μ), where $\mu = \mu'.\dot{\in}$, we have:

1. If $\vdash (C,\mu) : (a_1 : \tau_1, \ldots, a_n : \tau_n)$ then

$$\Theta^X(Y_1, \ldots, Y_n) \equiv \mathit{Nav}^X_{(C,\mu)} \cup \{Y_1 \doteq \pi_{a_1} X, \ldots, Y_n \doteq \pi_{a_n} X\}$$

is a characterizing formula for (C,μ);

2. If $\vdash (C,\mu) : \langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle$ then for $i = 1, \ldots, n$,

$$\Theta^X(Y) \equiv \mathit{Nav}^X_{(C,\mu)} \cup \{X \doteq \mathit{ins}_{a_i} Y\}$$

is a characterizing formula for (C,μ).

3. If $\vdash (C,\mu) : \underline{b}$ then, for any constant symbol $c^{\underline{b}}$

$$\Theta^X() \equiv \mathit{Nav}^X_{(C,\mu)} \cup \{X \doteq c^{\underline{b}}\}$$

is a characterizing formula for (C,μ).

What makes these formulae nice is that the conditions already in the definition of normal-form clauses, which are necessary to ensure completeness, are sufficient to ensure that these characterizing formulae are respected.

This leads us to a simplified definition of normal forms for non-nested target schemas:

Definition 14.12: Suppose that **Tr** is a transformation program from source schema $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ to target non-nested schema $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$.

A clause $\Psi \Longleftarrow \Phi$ is in **non-nested normal form** for **Tr** iff

1. $\Psi \Longleftarrow \Phi$ is in semi-normal form;

2. Φ contains no target terms;

3. If $\Phi \cup \Psi \vdash X : (C, \epsilon)$ for some X and some $C \in \mathcal{C}_{Tgt}$, then $\Phi \cup \Psi$ contains an atom $X \doteq \mathit{Mk}^C Y$ for some variable $Y \in \mathit{Var}(\Phi \cup \Psi)$.

4. If $\Phi \cup \Psi \vdash X : (C,\mu)$ for some X and some target term path (C,μ) and $\vdash (C,\mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$, then for each a_i, $\Phi \cup \Psi$ contains an atom $(Y \doteq \pi_{a_i} X)$ for some $Y \in \mathit{Var}(\Phi \cup \Psi)$;

5. If $\Phi \cup \Psi \vdash X : (C,\mu)$ for some X and some target term path (C,μ) and $\vdash (C,\mu) : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$, then $\Phi \cup \Psi$ contains an atom $\mathit{ins}_{a_i} Y \doteq X$ for some $Y \in \mathit{Var}(\Phi \cup \Psi)$ and some a_i;

6. If $\Phi \cup \Psi \vdash X : (C,\mu)$ for some X and some target path (C,μ), and $\vdash (C,\mu) : \underline{b}$, then either $X \in \mathit{Var}(\Phi)$ or $\Psi \cup \Phi$ contains an atom $X \doteq c^{\underline{b}}$ for some constant symbol $c^{\underline{b}}$.

■

It is clear that this is a more usable and easily checked definition than definition 14.11. Further it does not seem that restricting our attention to non-nested target schemas is a major limitation in practice: deeply nested data structures may still be represented by inserting addition object-identities and classes. The author therefore believes that practical implementation work on *WOL* or other similar database transformation techniques should concentrate on the case of target databases which do not involve nested sets.

14.6 Unifiers and Unfoldings

Our algorithm for converting clauses into normal form works by repeatedly unfolding a *target clause* on a series of transformation clauses until the target cause is in normal form. The process starts with a target clause which completely describes part of the target database.

Definition 14.13: A **description clause** for a target database schema $\mathcal{S}_{Tgt}$ is a semi-normal form clause of the form

$$\Phi \Longleftarrow \Phi$$

where

1. Φ contains no source terms and atoms of the form $X \notin Y$, $X \neq Y$ or $X = Y$ (X and Y variables);
2. if $\Phi \vdash X : (C, \mu)$ and $(C, \mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$ then, for each a_i, Φ contains an atom $Y \doteq \pi_{a_i} X$ for some Y; and
3. if $\Phi \vdash X : (C, \mu)$ and $(C, \mu) : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ then, for some a_i and some Y, Φ contains the atom $X \doteq ins_{a_i} Y$.

■

So the head of a description clause satisfies the requirements for a normal form clause, but the body is identical to the head.

However, as we shall see in section 14.7, merely applying all possible sequences of unfoldings is unlikely to be efficient, and may not even terminate.

Unifiers

Unifiers map the variables of a target clause to those of an unfolding (transformation) clause, so that the atoms of the two clauses can be matched and the target clause unfolded. A variable of the unfolding clause may match multiple universal variables from the target clause, although each target variable can match at most one variable from the unifying clause.

Definition 14.14: Suppose $\Xi \equiv (\Psi_t \Longleftarrow \Phi_t)$ and $\Delta \equiv (\Psi_u \Longleftarrow \Phi_u)$ are two clauses in semi-normal form with disjoint variables. A **unifier** from Δ to Ξ is a partial mapping

$$\mathcal{U} : \mathit{var}(\Phi_t) \overset{\sim}{\rightarrow} \mathit{var}(\Phi_u \cup \Psi_u)$$

which respects types. That is, if a variable X has type τ in Ξ and X is in the domain of $\mathcal{U}$, then $\mathcal{U}(X)$ has type τ in Δ. ■

If Φ is a set of atoms, we write $\mathcal{U}(\Phi)$ for the result of replacing each occurrence of a variable X in Φ, where X is in the domain of $\mathcal{U}$, by $\mathcal{U}(X)$. If $\Xi \equiv \Psi_t \Longleftarrow \Phi_t$ is a clause, we write $\mathcal{U}(\Xi)$ for the clause $\mathcal{U}(\Psi_t) \Longleftarrow \mathcal{U}(\Phi_t)$.

Lemma 14.5: If Ξ and Δ are clauses and $\mathcal{U}$ is a unifier from Δ to Ξ, then $\Xi \models \mathcal{U}(\Xi)$. ■

Proof: Suppose $\Xi \equiv (\Psi \Longleftarrow \Phi)$, and that $\mathcal{I}$ is an instance satisfying Ξ. Suppose that ρ is an environment such that $[\![\mathcal{U}(\Phi)]\!]\mathcal{I}\rho = \mathbf{T}$. Define the environment ρ' with $\mathit{dom}(\rho') = \mathit{Var}(\Phi)$ by

$$\rho'(X) \equiv \begin{cases} \rho(\mathcal{U}(X)) & \text{if } X \in \mathit{dom}(\mathcal{U}) \\ \rho(X) & \text{otherwise} \end{cases}$$

Then $[\![\Phi]\!]\mathcal{I}\rho' = \mathbf{T}$. Hence there is an extension ρ'' of ρ' with $\mathit{dom}(\rho'') = \mathit{Var}(\Psi \cup \Phi)$ such that $[\![\Psi]\!]\mathcal{I}\rho'' = \mathbf{T}$. Define the environment ρ''' with $\mathit{dom}(\rho''') = \mathit{Var}(\mathcal{U}(\Psi) \cup \mathcal{U}(\Phi))$ by

$$\rho'''(X) \equiv \begin{cases} \rho(X) & \text{if } X \in \mathit{Var}(\mathcal{U}(\Phi)) \\ \rho''(X) & \text{otherwise} \end{cases}$$

Then $[\![\mathcal{U}(\Psi)]\!]\mathcal{I}\rho''' = \mathbf{T}$. Hence $\mathcal{I}$ satisfies $\mathcal{U}(\Xi)$. ■

Unfolding

Let Ξ and Δ be semi-normal form clauses as before, and $\mathcal{U}$ a unifier from Δ to Ξ. Define $\mathit{Unfold}^*(\Xi, \Delta, \mathcal{U})$ by

$$\mathit{Unfold}^*(\Xi, \Delta, \mathcal{U}) \equiv (\Psi \Longleftarrow \Phi)$$

where

$$\begin{aligned} \Psi &\equiv \mathcal{U}(\Psi_t) \cup (\Psi_u \setminus \mathcal{U}(\Phi_t)) \\ \Phi &\equiv (\mathcal{U}(\Phi_t) \setminus \Psi_u) \cup \Phi_u \end{aligned}$$

Define $\mathit{Unfold}'(\Xi, \Delta, \mathcal{U})$ to be the set of minimal well-formed clauses $\Psi \Longleftarrow \Phi^*$ (ordered by the subset ordering on the heads and bodies of clauses), such that $\Phi \subseteq \Phi^* \subseteq \mathcal{U}(\Phi_t) \cup \Phi_u$, where $\mathit{Unfold}^*(\Xi, \Delta, \mathcal{U}) \equiv \Psi \Longleftarrow \Phi$. So $\mathit{Unfold}'(\Xi, \Delta, \mathcal{U})$ is formed by adding a minimal set of atoms from Φ_t to Φ necessary to make it range-restricted. Since there may be several ways of adding atoms in order to preserve range restriction, Unfold' is set valued.

Define $\mathit{Unfold}(\Xi, \Delta, \mathcal{U})$ to be the set of clauses $\Psi \Longleftarrow \Phi$ such that $\Psi \Longleftarrow \Phi$ is in semi-normal form and there is an equivalent clause $(\Psi' \Longleftarrow \Phi') \in \mathit{Unfold}'(\Xi, \Delta, \mathcal{U})$. So $\mathit{Unfold}(\Xi, \Delta, \mathcal{U})$ is formed by taking the clauses of $\mathit{Unfold}'(\Xi, \Delta, \mathcal{U})$ and combining any variables that are implied to be equal.

Unfoldable Clauses

We would like to capture the idea of when the unfolding of some target clause on a transformation clause and unifier is *valid*, and particularly when it helps to move the target clause towards normal form. For this to be the case, we want to make sure that some atoms will be removed from the body of the target clause as a result of the unfolding. The problem with this is that it may be necessary to put back some atoms, even though they are matched by the head of the unfolding clause, in order to maintain range restriction. We therefore concentrate on making sure that an unfolding will match some atoms which will not have to be put back because they do not play a part in maintaining range restriction.

Suppose Φ is a set of snf atoms. A variable $X \in \mathit{Var}(\Phi)$ is said to be **relatively base** in Φ iff Φ contains no atoms of the forms $X \doteq \mathit{ins}_a Y$, $Y \doteq \pi_a X$, $Y \doteq !X$ or $Y \dot{\in} X$, for any variable Y.

An atom $\phi \in \Phi$ is said to be **relatively base** in Φ iff ϕ contains a variable which is relatively base in Φ.

Definition 14.15: Suppose $\Xi \equiv (\Psi_t \Longleftarrow \Phi_t)$ and $\Delta \equiv (\Psi_u \Longleftarrow \Phi_u)$ are semi-normal form clauses, and $\mathcal{U}$ a unifier from Δ to Ξ. Then an atom $\phi \in \Phi_t$ is said to be **caught** by the unfolding of Ξ on $\Delta, \mathcal{U}$ iff ϕ is relatively base in Φ_t and $\mathcal{U}(\phi) \in \Psi_u$.

We write $\mathit{Caught}(\Xi, \Delta, \mathcal{U})$ to be the set of atoms caught by the unfolding of Ξ on $\Delta, \mathcal{U}$. ■

Note that, if $(\Psi \Longleftarrow \Phi) \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$ and $\phi \in \mathit{Caught}(\Xi, \Delta, \mathcal{U})$ then $\phi \notin \Phi$. That is, any atom caught by a transformation will be successfully removed from the target clause.

Definition 14.16: Ξ is said to be **unfoldable** on Δ, $\mathcal{U}$ iff

1. There is no variable in $X \in \mathit{var}(\Psi_u) \setminus \mathit{var}(\Phi_u)$ such that $X \in \mathit{var}(\Phi)$ for some clause $(\Psi \Longleftarrow \Phi) \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$;
2. $\mathit{Caught}(\Xi, \Delta, \mathcal{U}) \neq \emptyset$

■

The first condition says that no existential variables in the unfolding clause become universal variables in the resulting clause, while the second condition states that the unfoldings are not trivial: that at least one atom was removed from the target clause in each unfolding.

Lemma 14.6: If $\mathcal{U}$ is a unifier from Δ to Ξ and $(\Psi \Longleftarrow \Phi) \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$, then $\Delta, \Xi \models \Psi \Longleftarrow \Phi$. ■

Proof: Suppose $(\Psi \Longleftarrow \Phi) \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$. Then $(\mathcal{U}(\Phi_t) \setminus \Psi_u) \cup \Phi_u \subseteq \Phi$ and $\Psi \subseteq \mathcal{U}(\Psi_t) \cup \Psi_u$.

Suppose $\mathcal{I}$ is an instance such that $\mathcal{I}$ satisfies Δ and Ξ, and ρ is an environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Then $\Phi_u \subseteq \Phi$ so $[\![\Phi_u]\!]\mathcal{I}\rho = \mathbf{T}$. Hence, since $\mathcal{I}$ satisfies Δ, there is an extension ρ' of ρ such that $[\![\Psi_u]\!]\mathcal{I}\rho' = \mathbf{T}$. But $\mathcal{U}(\Phi_t) \subseteq \Phi \cup \Psi_u$, so $[\![\mathcal{U}(\Phi_t)]\!]\mathcal{I}\rho' = \mathbf{T}$. From the previous lemma, $\mathcal{I}$ satisfies $\mathcal{U}(\Xi)$, so there is an extension ρ'' of ρ' such that $[\![\mathcal{U}(\Psi_t)]\!]\mathcal{I}\rho'' = \mathbf{T}$. That is, there is an extension ρ'' of ρ such that $[\![\Psi]\!]\mathcal{I}\rho'' = \mathbf{T}$. Hence $\mathcal{I}$ satisfies $\Psi \Longleftarrow \Phi$. ■

14.7 Recursive Transformation Programs

Intuitively a *recursive* transformation program is one that admits an infinite series of unfoldings of clauses. It seems clear that if a transformation program is *recursive* then we can not hope to find an equivalent program in which all the transformation clauses are in normal form. However it is not decidable whether a transformation program admits such an infinite series of unfoldings. Consequently it is necessary to find some stronger test which ensures that a transformation program does not admit any such infinite sequences of unfoldings, but which allows as many useful non-recursive transformation programs as possible.

For example the following clause, representing a transitive closure property, is clearly recursive:

$$\begin{aligned}&W \in C, W.a = X, W.b = Y \Longleftarrow \\ &\qquad U \in C, V \in C, U.a = X, U.b = Z, V.a = Z, V.b = Y\end{aligned}$$

If we were to include this clause in a transformation program then we could unfold it infinitely many times, never reaching a normal form.

Traditionally, in Datalog, recursion is defined in terms of the *dependency graph* of a program: the dependency graph has nodes for each predicate symbol, and has an arrow from one predicate symbol to another if the program contains a clause with the first symbol occurring in the body and the second symbol as the head. A Datalog program is then said to be recursive if it's dependency graph contains a cycle.

So for example the clause above would be considered as recursive in Datalog, because the dependency graph would contain an edge from the symbol R to itself.

On the other hand, the following clause in not recursive in our language (though it could still comprise part of a recursive program together with some other clauses), even though it would be considered to be recursive using the datalog definition of recursion.

$$X.b = Y \Longleftarrow X \in C, X.a = Y$$

This can be explained by saying that our language works at a finer level than Datalog: in Datalog such a clause would be considered to be defining an element of C in terms of itself, while in our language it is considered to be determining the b-attribute of an element of C in terms of the a-attribute of the element of C.

Consequently we would like a finer notion of non-recursive programs, which disallows any transformation programs such as the first, which do admit infinite series of unfoldings, but which allows for as many transformation programs that do not admit such infinite sequences of unfoldings as possible.

Note that the notion of a *recursive transformation program* is a syntactic, rather than a semantic one. Consequently, given a recursive transformation program, it is quite possible that there is an equivalent non-recursive, and therefore normalizable, transformation program. However the problem of determining whether a transformation program is equivalent to some non-recursive transformation program is almost certainly undecidable.

Infinite Unfolding Sequences

Our mechanism for detecting recursive transformation programs works by making use of semi-normal forms and the presence of atoms corresponding to each point at which values are potentially being created. For each atom a record is kept of the last transformation clause to have *"touched"* that atom during the transformation process. The idea is that, in an infinite series of unfoldings, eventually it will be necessary to do an unfolding on a clause such that every atom involved in the unfolding has already been touched by the clause. When this happens recursion is detected and the transformation program is rejected.

An **unfolding sequence** for a transformation program **Tr** consists of a (possibly infinite) sequence of clauses, $\Xi_0, \Delta_0, \Xi_1, \Delta_1, \ldots, \Xi_i, \Delta_i, \ldots$ and a sequence of unifiers, $\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_i, \ldots$, such that

1. $\Delta_i \in \mathbf{Tr}$ for $i = 0, 1, 2, \ldots$, and
2. $\Xi_{i+1} \in \mathit{Unfold}(\Xi_i, \Delta_i, \mathcal{U}_i)$ for $i = 0, 1, 2, \ldots$.

An unfolding sequence consisting of clauses $\Xi_0, \Delta_0, \Xi_1, \ldots, \Xi_i, \Delta_i, \ldots$ and unifiers $\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_i, \ldots$ is said to be **valid** iff Ξ is unfoldable on $\Delta_i, \mathcal{U}_i$

A **decoration** of an unfolding sequence is a sequence of maps, $\delta_0, \delta_1, \ldots, \delta_i, \ldots$, such that

1. $\delta_i : \Phi_i \to \mathbb{N}$ for $i = 0, 1, 2, \ldots$,
2. $\delta_0(\phi) = 0$ for each $\phi \in \Phi_0$, and
3. $\delta_{i+1}(\phi) = \begin{cases} \delta_i(\phi') & \text{if } \phi' \in \Phi_i,\ \phi = \mathcal{U}_i(\phi') \\ & \text{and } \mathcal{U}_i(\phi') \notin \Phi'_i \\ i+1 & \text{otherwise} \end{cases}$
   for $i = 0, 1, 2, \ldots$

So the decoration of an unfolding sequence represents the number of the *last* unfolding involved in generating an atom.

Proposition 14.7: Suppose $\Xi_0, \Delta_0, \Xi_1, \Delta_1, \ldots, \mathcal{U}_0, \mathcal{U}_1, \ldots$ is valid infinite unfolding sequence for **Tr**. Then there is a k such that, for every $\phi \in \mathit{Caught}(\Xi_k, \Delta_k, \mathcal{U}_k)$ there is an $i \leq \delta_k(\phi)$ such that $\Delta_i = \Delta_k$. ■

This proposition means that, in an infinite unfolding sequence, eventually we will get to an unfolding where, for each atom being removed by the unfolding, the transformation clause of the unfolding has already been used in generating that atom.

In order to prove this we will first need some additional notation and a lemma.

Suppose $\Xi_0, \Delta_0, \Xi_1, \Delta_1, \ldots$ and $\mathcal{U}_0, \mathcal{U}_1, \ldots$ form a valid infinite unfolding sequence, and $\Xi_i \equiv (\Psi_i \Longleftarrow \Phi_i)$, $\Delta_i \equiv (\Psi'_i \Longleftarrow \Phi'_i)$ for $i = 0, 1, \ldots$. For any i and $j > i$, we define the partial function $\mathcal{U}_i^j : \mathit{Var}(\Phi_i) \overset{\sim}{\rightarrow} \mathit{Var}(\Psi'_j \cup \Phi'_j)$ by

$$\begin{array}{lcl}\mathcal{U}_i^{i+1} & \equiv & \mathcal{U}_i \\ \mathcal{U}_i^{j+1} & \equiv & \mathcal{U}_j \circ \mathcal{U}_i^j \quad \text{for } j > i\end{array}$$

Lemma 14.8: Suppose $\Xi_0, \Delta_0, \Xi_1, \Delta_1, \ldots$ and $\mathcal{U}_0, \mathcal{U}_1, \ldots$ form an infinite unfolding sequence, with $\Xi_i \equiv (\Psi_i \Longleftarrow \Phi_i)$ for $i = 0, 1, \ldots$. Then for each i there is a $k > i$ such that, for any $j \geq k$, if $\phi \in \Phi_i$ is such that $\mathcal{U}_i^j(\phi) \in \mathit{Caught}(\Xi_j, \Delta_j, \mathcal{U}_j)$ then $\delta_j(\mathcal{U}_i^j(\phi)) \neq \delta_i(\phi)$. ■

Proof: Suppose there is no such k. Then there are infinitely many js, $j > i$, such that $\mathcal{U}_i^j(\phi) \in \mathit{Caught}(\Xi_j, \Delta_j, \mathcal{U}_j)$ and $\delta_j(\mathcal{U}_i^j(\phi)) = \delta_i(\phi)$ for some $\phi \in \Phi_i$.

But if $\mathcal{U}_i^j(\phi) \in \mathit{Caught}(\Xi_j, \Delta_j, \mathcal{U}_j)$ then $\mathcal{U}_i^{j+1}(\phi) \notin \Phi_{j+1}$. Hence if $\mathcal{U}_i^l(\phi) \in \mathit{Caught}(\Xi_l, \Delta_l, \mathcal{U}_l)$ for some $l > j$, then $\delta_l(\mathcal{U}_i^l(\phi)) > j$.

Since Φ_i is finite, there can be at most finitely many j's such that $\mathcal{U}_i^{j-1}(\phi) \in \mathit{Caught}(\Xi_j, \Delta_j, \mathcal{U}_j)$ and $\delta_j(\mathcal{U}_i^{j-1}(\phi)) = \delta_i(\phi)$ for some $\phi \in \Phi$. Hence result. ■

Proof of proposition 14.7: Suppose the proposition is not true. Then there is a transformation clause $\Delta \in \mathbf{Tr}$ such that Δ occurs infinitely often in the unfolding sequence, and for any i such that $\Delta_i = \Delta$ there is a $\phi \in \mathit{Caught}(\Xi_i, \Delta_i, \mathcal{U}_i)$ such that, for any $j < \delta_i(\phi)$ $\Delta_j \neq \Delta$.

Let i be the first such that $\Delta_i = \Delta$. Then by lemma 14.8, there is a k such that for all $j > k$ and any $\phi \in \mathit{Caught}(\Xi_j, \Delta_j, \mathcal{U}_j)$, $\delta_j(\phi) > i$. Then since there are infinitely many occurrences of Δ in our unfolding sequence, we can pick i such that $i > k$ and $\Delta_i = \Delta$, contradicting our original hypothesis. ■

This proposition leads to a fine definition of recursion, for which, as we will see, tests can be built into out normalization algorithm.

Definition 14.17: Suppose $\mathbf{Tr}$ is a transformation program from $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$. Then $\mathbf{Tr}$ is said to be **recursive** iff there is an unfolding sequence for $\mathbf{Tr}$, $\Xi_0, \Delta_0, \ldots, \Xi_k, \Delta_k, \mathcal{U}_0, \ldots, \mathcal{U}_k$ with decoration $\delta_0, \ldots, \delta_k$, such that Ξ_0 is a description clause for $\mathcal{S}_{Tgt}$, and for any $\phi \in \mathit{Caught}(\Xi_k, \Delta_k, \mathcal{U}_k)$ there is an $i \leq \delta_k(\phi)$ with $\Delta_i = \Delta_k$. ■

Note that, while a transformation program which admits an infinite sequence of unfoldings, starting from a description clause, must be recursive, it does not follow that any recursive transformation program must admit such a sequence of unfoldings.

The following proposition tells us that a complete, non-recursive transformation program can be unfolded into an equivalent normal form program. Note that the result holds only for programs consisting entirely of transformation clauses. If a transformation program included constraints then the result would not necessarily hold, and would have to be checked for the individual program. Also we require that the underlying base domains are infinite.

Proposition 14.9: Given any non-recursive transformation program $\mathbf{Tr}$ from $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$ containing only transformation clauses, if $\mathbf{Tr}$ is complete then there is an equivalent transformation program $\mathbf{Tr}'$ such that $\mathbf{Tr}'$ is in normal form. Further there is an algorithm which will compute such a $\mathbf{Tr}'$ if $\mathbf{Tr}$ is complete, or terminate reporting that $\mathbf{Tr}$ is not complete otherwise. ■

Such a $\mathbf{Tr}'$ can be computed by taking the maximal unfolding sequences of the description clauses for $\mathcal{S}_{Tgt}$ with clauses from $\mathbf{Tr}$.

We will give the proof for the case when $\mathcal{S}_{Tgt}$ is a simply-keyed, non-nested schema as described in section 14.5.

The proof uses an adaption of standard techniques such as Herbrand universes and models. However it is quite long and will proceed in a series of stages which will comprise the remainder of this section, and will require a number of intermediary definitions. In order to simplify the presentation we will take some notational liberties, particularly concerning the distinctions between the values of an instance, syntactic terms and the terms of the algebra which form a Herbrand universe. We will also assume an implicit ordering on the set of all variables, so that the variables of a particular clause or set of atoms will be considered to form an ordered sequence or tuple rather than a set.

Herbrand Universes and Models

We will assume a source schema $\mathcal{S}_{Src}$, a target schema $\mathcal{S}_{Tgt}$ with classes $\mathcal{C}_{Src}$ and $\mathcal{C}_{Tgt}$ respectively, and a non-recursive transformation program $\mathbf{Tr}$.

For each transformation clause $\Delta \in \mathbf{Tr}$, $\Delta \equiv (\Phi \Longleftarrow \Psi)$, and every variable $Y \in \mathit{Var}(\Phi) \setminus \mathit{Var}(\Psi)$, we assume a function symbol f^Y_Δ. If $\mathit{Var}(\Psi) = X_1, \ldots, X_k$, where $\Delta \vdash X_i : \tau_i$ for $i = 1, \ldots, k$, and $\Delta \vdash Y : \tau$, then we say that f^Y_Δ has *domain type* $(\tau_1, \ldots, \tau_k)$ and *range type* τ, and will use the notation $f^Y_\Delta : (\tau_1, \ldots, \tau_k) \to \tau$.

Definition 14.18: A **Herbrand universe** or *H-universe* is a family of sets $\mathcal{H}(\tau)$, $\tau \in \mathit{Types}(\mathcal{C})$ such that, if $u_i \in \mathcal{H}(\tau_i)$ for $i = 1, \ldots, k$, and $f^Y_\Delta : (\tau_1, \ldots, \tau_k) \to \tau$, then $f^Y_\Delta(u_1, \ldots, u_k) \in \mathcal{H}(\tau)$. We will also sometimes overload notation and write $\mathcal{H}$ for the set $\bigcup_{\tau \in \mathit{Types}(\mathcal{C})} \mathcal{H}(\tau)$.

We define *Symb* to be the set

$$\mathit{Symb} \equiv \{!, \dot{\in}\} \cup \{\mathit{Mk}^C | C \in \mathcal{C}\} \cup \{\pi_a | a \in \mathcal{A}\} \cup \{\mathit{ins}_a | a \in \mathcal{A}\}$$

We also assume a set of constant symbols *Const* which include a symbol $c^{\underline{b}}$ corresponding to each element c of the domain of base type $\underline{b}$.

An **H-model**, $(\mathbf{M}, \mathbf{A})$, over an H-universe $\mathcal{H}$ consists of a family of sets $\mathbf{M}(\tau) \subseteq \mathcal{H}(\tau)$, $\tau \in$ *Types*, and a set $\mathbf{A}$ where

$$\mathbf{A} \subseteq (\mathbf{M} \times \mathit{Symb} \times \mathbf{M}) \cup (\mathbf{M} \times \mathit{Const})$$

(where $\mathbf{M}$ denotes $\bigcup_{\tau \in \mathit{Types}(\mathcal{C})} \mathbf{M}(\tau)$), and, in addition, $\mathbf{M}$ and $\mathbf{A}$ satisfy the following conditions:

1. If $f^Y_\Delta(u_1, \ldots, u_n) \in \mathbf{M}$ then $u_i \in \mathbf{M}$ for $i = 1, \ldots, n$,

2. If $(u, \pi_{a_i}, v) \in \mathbf{A}$ then $u \in \mathbf{M}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ and $v \in \mathbf{M}(\tau_i)$ for some type $(a_1 : \tau_1, \ldots, a_k : \tau_k) \in \mathit{Types}(\mathcal{C})$.

3. If $(u, \mathit{ins}_{a_i}, v) \in \mathbf{A}$ then $v \in \mathbf{M}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ and $v \in \mathbf{M}(\tau_i)$ for some type $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle \in \mathit{Types}(\mathcal{C})$.

4. If $(u, !, v) \in \mathbf{A}$ then $u \in \mathbf{M}(\tau^C)$ and $v \in \mathbf{M}(C)$ for some $C \in \mathcal{C}$.

5. If $(u, \mathit{Mk}^C, v) \in \mathbf{A}$ then $u \in \mathbf{M}(C)$ and $v \in \mathbf{M}(\kappa^C)$.

6. If $(u, \dot{\in}, v) \in \mathbf{A}$ then $u \in \mathbf{M}(\tau)$ and $v \in \mathbf{M}(\{\tau\})$ for some type τ.

7. If $(u, c^{\underline{b}}) \in \mathbf{A}$ then $u \in \mathbf{M}(\underline{b})$.

■

So H-universes and H-models provide us with a means of representing the *syntactic* atoms and objects implied by a transformation program. In order to make use of them we will first have to construct an H-model representing precisely the values and objects of a source database instance, and an H-universe representing the objects that can potentially be built out of that instance. To do this we must first extract the values occurring in an instance.

Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance. We define the family of sets $\mathit{Vals}_{\mathcal{I}}(\tau) \subseteq [\![\tau]\!]\mathcal{I}$, $\tau \in \mathit{Types}(\mathcal{C})$, to be the smallest sets such that

1. $\mathit{Vals}_{\mathcal{I}}(C) = \sigma^C$ for each $C \in \mathcal{C}$;

2. if $u \in \mathit{Vals}_{\mathcal{I}}(C)$ for some $C \in \mathcal{C}$, then $\mathcal{V}^C(u) \in \mathit{Vals}_{\mathcal{I}}(\tau^C)$;

3. if $u \in \mathit{Vals}_{\mathcal{I}}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ for some record type $(a_1 : \tau_1, \ldots, a_k : \tau_k)$, then $u(a_i) \in \mathit{Vals}_{\mathcal{I}}(\tau_i)$ for $i = 1, \ldots, k$;

4. if $(a_i, u) \in \mathit{Vals}_{\mathcal{I}}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ for some variant type $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$, then $u \in \mathit{Vals}_{\mathcal{I}}(\tau_i)$; and

5. if $u \in \mathit{Vals}_{\mathcal{I}}(\{\tau\})$ for some type τ, and $v \in u$ then $v \in \mathit{Vals}_{\mathcal{I}}(\tau)$.

Given an instance $\mathcal{I}$ we let $\mathcal{H}(\mathcal{I})$ denote the smallest H-universe such that $\mathit{Vals}_{\mathcal{I}}(\tau) \subseteq \mathcal{H}(\tau)$ for every type τ.

We define $\mathit{Mod}(\mathcal{I})$ to be the smallest H-model over $\mathcal{H}(\mathcal{I})$, $(\mathbf{M}, \mathbf{A})$, such that

1. $\mathit{Vals}_{\mathcal{I}}(\tau) \subseteq \mathbf{M}(\tau)$ for every type τ;

2. if $u \in \sigma^C$ and $v = \mathcal{V}^C(u)$ then $(v, !, u) \in \mathbf{A}$;

3. if $u \in \sigma^C$ and $v = \mathcal{K}^C(u)$ then $(u, \mathit{Mk}^C, v) \in \mathbf{A}$;

4. if $u \in \mathit{Vals}_{\mathcal{I}}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ and $v = u(a_i)$ then $(v, \pi_{a_i}, u) \in \mathbf{A}$;

5. if $u \in \mathit{Vals}_{\mathcal{I}}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ and $u = (a_i, v)$ then $(u, \mathit{ins}_{a_i}, v) \in \mathbf{A}$; and
6. if $u \in \mathit{Vals}_{\mathcal{I}}(\{t\})$ and $v \in u$ then $(v, \dot{\in}, u) \in \mathbf{A}$; and
7. if $c \in \mathit{Vals}_{\mathcal{I}}(\underline{b})$ then $(c, c^{\underline{b}}) \in \mathbf{A}$.

An **interpretation** of an H-model $(\mathbf{M}, \mathbf{A})$ in an instance $\mathcal{I}$ is a family of mappings $\mathbf{I}_\tau : \mathbf{M}(\tau) \to \mathit{Vals}_{\mathcal{I}}(\tau)$ such that

1. if $u \in \mathbf{M}(C)$ and $(v, !, u) \in \mathbf{A}$ then $\mathbf{I}_{\tau^C}(v) = \mathcal{V}^C(\mathbf{I}_C(u))$;
2. if $u \in \mathbf{M}(C)$ and $(u, \mathit{Mk}^C, v) \in \mathbf{A}$ then $\mathbf{I}_{\tau^C}(v) = \mathcal{K}^C(\mathbf{I}_C(u))$;
3. if $u \in \mathbf{M}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ and $(v, \pi_{a_i}, u) \in \mathbf{A}$ then $\mathbf{I}_{\tau_i}(v) = (\mathbf{I}_{(a_1:\tau_1,\ldots,a_k:\tau_k)}(u))(a_i)$;
4. if $u \in \mathbf{M}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ and $(u, \mathit{ins}_{a_i}, v) \in \mathbf{A}$ then $\mathbf{I}_{\langle\!| a_1:\tau_1,\ldots,a_k:\tau_k |\!\rangle}(u) = (a_i, \mathbf{I}_{\tau_i}(v))$;
5. if $u \in \mathbf{M}(\{\tau\})$ and $(v, \dot{\in}, u) \in \mathbf{A}$ then $\mathbf{I}_\tau(v) \in \mathbf{I}_{\{\tau\}}(u)$; and
6. if $(u, c^{\underline{b}}) \in \mathbf{A}$ then $\mathbf{I}_{\underline{b}}(u) = c$.

Interpretations provide a means of connecting H-models to the instances they represent.

Note that, for any instance $\mathcal{I}$, there is a unique interpretation $\mathbf{I}$ of the H-model $\mathit{Mod}(\mathcal{I})$ in $\mathcal{I}$ such that $\mathbf{I}_\tau(u) = u$ for each $u \in \mathit{Vals}_{\mathcal{I}}(\tau)$.

Skolemization of Clauses

We introduce the process of Skolemization in order to remove existential variables from our clauses and replace them with applications of Skolem functions. The way in which these functions are applied within a term will show how the term was created and the object it represents arose.

First we must extend definition 13.1 to include terms build using Skolem functions:

$$\begin{array}{rcl} P & ::= & \ldots \\ & | & f^Y_\Delta(P_1, \ldots, P_k) \end{array}$$

and add the following typing rule to definition 13.2:

$$\frac{\Gamma \vdash P_1 : \tau_1 \ \ldots \ \Gamma \vdash P_k : \tau_k}{\Gamma \vdash f^Y_\Delta(P_1, \ldots, P_k) : \tau} \quad \text{where } f^Y_\Delta : (\tau_1, \ldots, \tau_k) \to \tau$$

Suppose $\Delta \in \mathbf{Tr}$ is a transformation clause, $\Delta \equiv (\Phi \Longleftarrow \Psi)$, and $\mathit{Var}(\Psi) = X_1, \ldots, X_k$. For any term P occurring in Δ we define $\widehat{P}$ to be the term formed by replacing any occurrence of a variable $Y \in \mathit{Var}(\Phi) \setminus \mathit{Var}(\Psi)$ in P with the term $f^Y_\Delta(X_1, \ldots, X_k)$.

We define $\widehat{\Delta}$ to be the clause formed by replacing every term P occurring in Δ, such that P does not occur within some larger term, with $\widehat{P}$. Consequently $\widehat{\Delta}$ contains no existential variables.

We say a clause Δ is *Skolemized* if it contains no existential variables, though it may contain universal variables and applications of Skolem functions. We say a term P occurring in Δ is a *variable term* iff either $P \equiv X$ for some variable X, or $P \equiv f^Y_\Delta(Q_1, \ldots, Q_k)$ where $Q_1, \ldots, Q_k$ are variable terms.

Generating H-models from Transformation Programs and Instances

Next we need to define how we can apply such Skolemized clauses to an H-model.

Suppose $(\mathbf{M}, \mathbf{A})$ is an H-model. We define $\mathit{Eq}(\mathbf{A})$ to be the smallest equivalence relation on $\mathbf{M}$ such that

1. if $(u, c^{\underline{b}}) \in \mathbf{A}$ and $(v, c^{\underline{b}}) \in \mathbf{A}$ for some $c^{\underline{b}} \in \mathit{Const}$, then $(u, v) \in \mathit{Eq}(\mathbf{A})$;
2. if $(u, v) \in \mathit{Eq}(A)$ and $(u', \pi_a, u) \in \mathbf{A}$ and $(v', \pi_a, v) \in \mathbf{A}$ then $(u', v') \in \mathit{Eq}(\mathbf{A})$;
3. if $u, v \in \mathbf{M}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ and there exist $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ such that $(u_i, \pi_{a_i}, u) \in \mathbf{A}$ and $(v_i, \pi_{a_i}, v) \in \mathbf{A}$ and $(u_i, v_i) \in \mathit{Eq}(\mathbf{A})$ for $i = 1, \ldots, k$, then $(u, v) \in \mathit{Eq}(\mathbf{A})$.
4. if $(u', \mathit{ins}_a, u) \in \mathbf{A}$ and $(v', \mathit{ins}_a, v) \in \mathbf{A}$ then $(u', v') \in \mathit{Eq}(\mathbf{A})$ iff $(u, v) \in \mathit{Eq}(\mathbf{A})$;
5. if $(u', \mathit{Mk}^C, u) \in \mathbf{A}$ and $(v', \mathit{Mk}^C, v) \in \mathbf{A}$ then $(u', v') \in \mathit{Eq}(\mathbf{A})$ iff $(u, v) \in \mathit{Eq}(\mathbf{A})$; and
6. if $(u', !, u) \in \mathbf{A}$ and $(v', !, v) \in \mathbf{A}$ and $(u, v) \in \mathit{Eq}(\mathbf{A})$ then $(u', v') \in \mathit{Eq}(\mathbf{A})$.

Lemma 14.10: If $(\mathbf{M}, \mathbf{A})$ is an H-model, $\mathbf{I}$ and interpretation of $(\mathbf{M}, \mathbf{A})$ in some instance $\mathcal{I}$, and $u, v \in \mathbf{M}(\tau)$ are such that $(u, v) \in \mathit{Eq}(\mathbf{A})$, then $\mathbf{I}_\tau(u) = \mathbf{I}_\tau(v)$. ∎

Proof: Follows by induction on the definition of $\mathit{Eq}(\mathbf{A})$. ∎

Suppose $\Delta \equiv (\Phi \Longleftarrow \Psi)$ is a transformation clause with $\mathit{Var}(\Psi) = X_1, \ldots, X_k$ and $\Delta \vdash X_i : \tau_i$ for $i = 1, \ldots, k$. An **H-environment** for Δ in an H-model $(\mathbf{M}, \mathbf{A})$ is a tuple $(u_1, \ldots, u_k)$ such that $u_i \in \mathbf{M}(\tau_i)$ for $i = 1 \ldots, k$.

If P is a variable term occurring in a Skolemized clause Δ such that $\Delta \vdash P : \tau$, and ξ is an H-environment for Δ, say $\xi = (u_1, \ldots, u_k)$, then we define $\xi(P) \in \mathcal{H}(\tau)$ by

1. if $P \equiv X_i$ then $\xi(P) \equiv u_i$; and
2. if $P \equiv f^Y_\Delta(Q_1, \ldots, Q_k)$ then $\xi(P) \equiv f^Y_\Delta(\xi(Q_1), \ldots, \xi(Q_k))$.

Suppose $(\mathbf{M}, \mathbf{A})$ is an H-model and ξ an H-environment for a clause Δ. $(\mathbf{M}, \mathbf{A})$ and ξ are said to satisfy an atom ψ occurring in Δ iff

1. $\psi \equiv (P \dot{=} !Q)$ and there exist $u, v \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(v, \xi(Q)) \in Eq(\mathbf{A})$ and $(u, !, v) \in \mathbf{A}$; or
2. $\psi \equiv (P \dot{=} Mk^C Q)$ and there exist $u, v \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(v, \xi(Q)) \in Eq(\mathbf{A})$ and $(u, Mk^C, v) \in \mathbf{A}$; or
3. $\psi \equiv (P \dot{=} \pi_a Q)$ and there exist $u, v \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(v, \xi(Q)) \in Eq(\mathbf{A})$ and $(u, \pi_a, v) \in \mathbf{A}$; or
4. $\psi \equiv (P \dot{=} ins_a Q)$ and there exist $u, v \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(v, \xi(Q)) \in Eq(\mathbf{A})$ and $(u, ins_a, v) \in \mathbf{A}$; or
5. $\psi \equiv (P \dot{\in} Q)$ and there exist $u, v \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(v, \xi(Q)) \in Eq(\mathbf{A})$ and $(u, \dot{\in}, v) \in \mathbf{A}$; or
6. $\psi \equiv (P \dot{=} c^{\underline{b}})$ and there exists $u \in \mathbf{M}$ such that $(u, \xi(P)) \in Eq(\mathbf{A})$ and $(u, c^{\underline{b}}) \in \mathbf{A}$; or
7. $\psi \equiv (P \dot{\neq} Q)$ and $(\xi(P), \xi(Q)) \notin Eq(\mathbf{A})$; or
8. $\psi \equiv (P \dot{\not\in} Q)$ and there do not exist $u, v \in \mathbf{M}$ such that $(\xi(P), u) \in Eq(\mathbf{A})$, $(\xi(Q), v) \in Eq(\mathbf{A})$ and $(u, \in, v) \in \mathbf{A}$.

If ξ is an H-environment for a clause Δ and ψ is a positive snf atom occurring in Δ, then we define $\xi(\psi)$ by

1. if $\psi \equiv (P \dot{=} !Q)$ then $\xi(\psi) \equiv (\xi(P), !, \xi(Q))$; and
2. if $\psi \equiv (P \dot{=} Mk^C Q)$ then $\xi(\psi) \equiv (\xi(P), Mk^C, \xi(Q))$; and
3. if $\psi \equiv (P \dot{=} \pi_a Q)$ then $\xi(\psi) \equiv (\xi(P), \pi_a, \xi(Q))$; and
4. if $\psi \equiv (P \dot{=} ins_a Q)$ then $\xi(\psi) \equiv (\xi(P), ins_a, \xi(Q))$; and
5. if $\psi \equiv (P \dot{\in} Q)$ then $\xi(\psi) \equiv (\xi(P), \dot{\in}, \xi(Q))$; and
6. if $\psi \equiv (P \dot{=} c^{\underline{b}})$ then $\xi(\psi) \equiv (\xi(P), c^{\underline{b}})$.

Given a transformation program $\mathbf{Tr}$ and an H-model $(\mathbf{M}, \mathbf{A})$ for $\mathbf{Tr}$, we define $\mathbf{Tr}(\mathbf{M}, \mathbf{A})$ to be the smallest H-model $(\mathbf{M}', \mathbf{A}')$ such that

1. For each type τ, $\mathbf{M}(\tau) \subseteq \mathbf{M}'(\tau)$; and
2. $\mathbf{A} \subseteq \mathbf{A}'$; and
3. for any transformation clause $\Delta \in \mathbf{Tr}$, $\Delta \equiv (\Phi \Longleftarrow \Psi)$, and H-environment ξ for Δ, if ξ and $(\mathbf{M}, \mathbf{A})$ satisfy ψ for each $\psi \in \Psi$, then $\xi(\phi) \in \mathbf{A}'$ for each $\phi \in \Phi$.

Given an instance $\mathcal{I}_{Src}$ of $\mathcal{S}_{Src}$ we can now form a series of H-models, $(\mathbf{M}^i, \mathbf{A}^i)$, $i \in \mathbb{N}$, defined by

$$\begin{array}{rcl}(\mathbf{M}^0, \mathbf{A}^0) & \equiv & \mathit{Mod}(\mathcal{I}_{Src}) \\ (\mathbf{M}^{i+1}, \mathbf{A}^{i+1}) & \equiv & \mathbf{Tr}(\mathbf{M}^i, \mathbf{A}^i)\end{array}$$

We can also define the H-model $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ such that $\mathbf{M}^\infty(\tau) = \bigcup_{i=0}^\infty \mathbf{M}^i(\tau)$ for each type τ, and $\mathbf{A}^\infty = \bigcup_{i=0}^\infty \mathbf{A}^i$. It is easy to check that $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is indeed an H-model.

Lemma 14.11: Suppose $\mathcal{I}_{Src}$ is an instance of $\mathcal{S}_{Src}$, $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is the H-model built from $\mathcal{I}_{Src}$ as described above, and $\mathcal{I}_{Tgt}$ is a $\mathbf{Tr}$-transformation of $\mathcal{I}_{Src}$: that is $\mathcal{I}_{Tgt}$ is an instance of $\mathcal{S}_{Tgt}$ such that the instance $\mathcal{I} = \mathcal{I}_{Src} \cup \mathcal{I}_{Tgt}$ satisfies each clause in $\mathbf{Tr}$. Then there is a unique interpretation $\mathbf{I}$ of $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ in $\mathcal{I}$ such that $\mathbf{I}_\tau(u) = u$ for all $u \in \mathit{Vals}_{\mathcal{I}_{Src}}(\tau)$. ■

Proof: We show that, for every $i \in \mathbb{N}$, there is a unique interpretation $\mathbf{I}^i$ of $(\mathbf{M}^i, \mathbf{A}^i)$ in $\mathcal{I}$ satisfying these properties.

We already have this result for $(\mathbf{M}^0, \mathbf{A}^0)$. Suppose we have such an $\mathbf{I}^i$ for $(\mathbf{M}^i, \mathbf{A}^i)$. For each $u \in \mathbf{M}^{i+1}(\tau)$ if $u \in \mathbf{M}^i(\tau)$ then we must have $\mathbf{I}^{i+1}_\tau(u) = \mathbf{I}^i_\tau(u)$ by uniqueness of $\mathbf{I}^i$ since $\mathbf{I}^{i+1}$ restricted to $\mathbf{M}^i$ is an interpretation from $(\mathbf{M}^i, \mathbf{A}^i)$ to $\mathcal{I}$.

Suppose $u \in \mathbf{M}^{i+1}(t) \setminus \mathbf{M}^i(\tau)$. Then there is a transformation clause $\Delta \equiv (\Phi \Longleftarrow \Psi)$ in $\mathbf{Tr}$ and an H-environment ξ for Δ in $(\mathbf{M}^i, \mathbf{A}^i)$ such that ξ satisfies each atom $\psi \in \Psi$ and there is a variable Y occurring in $\mathit{Var}(\Phi) \setminus \mathit{Var}(\Psi)$ such that $\xi(\widehat{(Y)}) = u$. Further $\xi(\widehat{\phi}) \in \mathbf{A}^{i+1}$ for each $\phi \in \Phi$.

Consider the $\mathcal{I}$-environment ρ with $\mathit{dom}(\rho) = \mathit{Var}(\Psi)$ defined by $\rho(X) \equiv \mathcal{I}^i_\tau(\xi(X))$ for $X \in \mathit{Var}(\Psi)$ where $\Delta \vdash X : \tau$. Since $\mathcal{I}^i$ is an interpretation of $(\mathbf{M}^i, \mathbf{A}^i)$ in $\mathcal{I}$ we have $[\![\psi]\!]\mathcal{I}\rho = \mathbf{T}$ for each $\psi \in \Psi$. Hence there is a *unique* extension of ρ to $\mathit{Var}(\Phi) \cup \mathit{Var}(\Psi)$, ρ' say, such that $[\![\phi]\!]\mathcal{I}\rho' = \mathbf{T}$ for all $\psi \in \Psi$. Then we can define $\mathbf{I}^{i+1}(u) \equiv \rho'(Y)$.

It is clear that $\mathbf{I}^{i+1}$ is an interpretation of $(\mathbf{M}^{i+1}, \mathbf{A}^{i+1})$ in $\mathcal{I}$. Finally we can define $\mathbf{I}$ by $\mathbf{I}_\tau(u) \equiv \mathbf{I}^j_\tau(u)$ where $u \in \mathbf{M}^j$. ■

Lemma 14.12: Suppose that $\mathcal{I}_{Src}$ is an instance of $\mathcal{S}_{Src}$ and $\mathcal{I}_{Tgt}$ an instance of $\mathcal{S}_{Tgt}$, and $\mathbf{I}$ is an interpretation of $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ in $\mathcal{I}$ ($\mathcal{I} = \mathcal{I}_{Src} \cup \mathcal{I}_{Tgt}$), such that for any transformation clause $\Delta \in \mathbf{Tr}$, $\Delta \equiv (\Psi \Longleftarrow \Phi)$, and $\mathcal{I}$-environment ρ, if $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$ then there exists an H-environment for Δ in $(\mathbf{M}^\infty, \mathbf{A}^\infty)$, ξ, such that $\mathcal{I}(\xi(X)) = \rho(X)$ for each $X \in \mathit{Var}(\Phi)$, ξ satisfies each atom $\phi \in \Phi$. Then $\mathcal{I}_{Tgt}$ is a $\mathbf{Tr}$-transformation of $\mathcal{I}_{Src}$. ■

Proof: Suppose $\mathcal{I}_{Tgt}$ is an instance and $\mathbf{I}$ an interpretation, as described in the lemma. Suppose $\Delta \equiv (\Psi \Longleftarrow \Phi)$ is a transformation clause and ρ an environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Suppose $\mathit{Var}(\Phi) = X_1, \ldots, X_k$, and $\xi = (u_1, \ldots, u_k)$ is an H-environment such that $\mathbf{I}(u_i) = \rho(X_i)$ for $i = 1, \ldots, k$, and ξ satisfies ϕ for each $\phi \in \Phi$. Then $\xi(\psi) \in \mathbf{A}^\infty$ for each $\psi \in \Psi$. Hence if we take ρ' to be the extension of ρ given by $\rho'(Y) = \mathbf{I}(f^Y_\Delta(u_1, \ldots, u_k))$ for $Y \in \mathit{Var}(\Psi) \setminus \mathit{Var}(\Phi)$, then $[\![\psi]\!]\mathcal{I}\rho' = \mathbf{T}$ for each $\psi \in \Psi$. Hence $\mathcal{I}$ satisfies Δ. ■

Complete H-Models

An H-model, $(\mathbf{M}, \mathbf{A})$ is said to be **complete** iff

1. if $u \in \mathbf{M}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ then for each $i \in 1, \ldots, k$ there exists a $v \in \mathbf{M}(\tau_i)$ such that $(v, \pi_{a_i}, u) \in \mathbf{A}$; and
2. if $u \in \mathbf{M}(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ then for some $i \in 1, \ldots, k$ there is a $v \in \mathbf{M}(\tau_i)$ such that $(u, \mathit{ins}_{a_i}, v) \in \mathbf{A}$; and
3. if $u \in \mathbf{M}(C)$ then there is a $v \in \mathbf{M}(\kappa^C)$ such that $(u, \mathit{Mk}^C, v) \in \mathbf{A}$, and there is a $v' in \mathbf{M}(\tau^C)$ such that $(v, !, u) \in \mathbf{A}$; and
4. if $u \in \mathbf{M}(\underline{b})$ for some base type $\underline{b}$, then there exists a $c^{\underline{b}} \in \mathit{Consts}$ such that $(u, c^{\underline{b}}) \in \mathbf{A}$.

Lemma 14.13: If $(\mathbf{M}, \mathbf{A})$ is a complete H-model and $\mathbf{I}$ is an interpretation of $(\mathbf{M}, \mathbf{A})$ in an instance $\mathcal{I}$, then for any $u, v \in \mathbf{M}(\tau)$, if $\mathbf{I}^\tau(u) = \mathbf{I}^\tau(v)$ then $(u, v) \in \mathit{Eq}(\mathbf{A})$. ■

Proof: Follows by induction on types and paths in the key-type dependency graph. ■

Lemma 14.14: If $\mathbf{Tr}$ is a complete transformation program then the H-model $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ formed as described above is complete. ■

Proof: It is enough to show that, if $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is not complete, and $\mathcal{I}_{\mathit{Tgt}}$ is a target instance such that $\mathcal{I} = \mathcal{I}_{\mathit{Src}} \cup \mathcal{I}_{\mathit{Tgt}}$ is a minimal instance satisfying $\mathbf{Tr}$, then there is another instance $\mathcal{I}'_{\mathit{Tgt}}$ such that $\mathcal{I}' = \mathcal{I}_{\mathit{Src}} \cup \mathcal{I}'_{\mathit{Tgt}}$ also satisfies $\mathbf{Tr}$ and $\mathcal{I}'$ is not isomorphic to $\mathcal{I}$.

Since $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is not complete, we can pick a $u\mathbf{M}^\infty$ such that one of the following conditions holds:

1. $u \in \mathbf{M}^\infty(a_1 : \tau_1, \ldots, a_k : \tau_k)$ and for some $i \in 1, \ldots, k$ there is no $v \in \mathbf{M}^\infty(\tau_k)$ such that $(v, \pi_{a_i}, u) \in \mathbf{A}^\infty$; or
2. $u \in \mathbf{M}^\infty(\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle)$ and there is no $i \in 1, \ldots, k$ and $v \in \mathbf{M}^\infty(\tau_i)$ such that $(u, \mathit{ins}_{a_i}, v) \in \mathbf{A}^\infty$; or
3. $u \in \mathbf{M}^\infty(\underline{b})$ and there is no $c^{\underline{b}} \in \mathit{Const}$ such that $(u, c^{\underline{b}}) \in \mathbf{A}^\infty$.

Suppose $\mathbf{I}$ is the unique interpretation of $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ in $\mathcal{I}$ described in lemma 14.11. Then we can form $\mathcal{I}'$ by replacing some base value occurring in $\mathbf{I}(u)$ (or $\mathbf{I}(u)$ itself if $u \in \mathbf{M}^\infty(\underline{b})$) with a new base value not occurring in $\mathcal{I}$ or in any of the clauses in $\mathbf{Tr}$. ■

Lemma 14.15: Suppose $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is a complete H-model for $\mathbf{Tr}$, $\mathcal{I}_{\mathit{Tgt}}$ is an instance of $\mathcal{S}_{\mathit{Tgt}}$, and $\mathbf{I}$ an interpretation of $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ in $\mathcal{I}$, such that, for each type τ, and for each $u \in \mathit{Vals}_{\mathcal{I}}(\tau)$ there is a $v \in \mathbf{M}(\tau)$ with $\mathbf{I}^\tau(v) = u$, and for each $u \in \mathit{Vals}_{\mathcal{I}}(\{\tau\})$ and $v \in u$ there is a

$u' \in \mathbf{M}(\{\tau\})$ and $\nu' \in \mathbf{M}(\tau)$ such that $\mathbf{I}^{\{\tau\}}(u') = u$, $\mathbf{I}^{\tau}(v') = v$ and $(v', \dot{\in}, u') \in \mathbf{A}$. Then $\mathcal{I}_{Tgt}$ is a **Tr**-transformation of $\mathcal{I}_{Src}$. ■

Proof: It is enough to prove that the such an instance satisfies the conditions of lemma 14.12.

Suppose that $\Delta = (\Psi \Longleftarrow \Phi)$ is a transformation clause and ρ an $\mathcal{I}$-environment such that $[\![\Phi]\!]\mathcal{I}\rho = \mathbf{T}$. Suppose $Var(\Phi) = X_1, \ldots, X_k$. We can choose $u_1, \ldots, u_k \in \mathbf{M}^\infty$ such that $\mathbf{I}(u_i) = \rho(X_i)$, for $i = 1, \ldots, k$, and let $\xi = (u_1, \ldots, u_k)$. We will show that, for each $\phi \in \Phi$, ξ satisfies ϕ in $(\mathbf{M}^\infty, \mathbf{A}^\infty)$.

Suppose $\phi \equiv (X_i \dot{\neq} X_j)$. Then, since Δ is a transformation clause, X_i and X_j are source terms. Hence $\rho(X_i) \in Vals(\mathcal{I}_{Src})$ and $\rho(X_j) \in Vals(\mathcal{I}_{Src})$. Since $\mathbf{I}(u_i) = \rho(X_i)$ and $\mathbf{I}(u_j) = \rho(X_j)$ it follows from lemma 14.10 that ξ satisfies ϕ. Similarly if $\phi \equiv (X_i \dot{\notin} X_j)$ then ξ satisfies ϕ.

Suppose $\phi \equiv (X_i \dot{in} X_j)$. Then the result follows from our assumptions about $\mathbf{I}$.

Suppose $\phi \equiv (X_i \doteq d^{\underline{b}})$. Then $u_i \in \mathbf{M}^\infty(\underline{b})$. Hence, since $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is complete, there is a constant $c^{\underline{b}}$ such that $(u_i, c^{\underline{b}}) \in \mathbf{A}^\infty$. Hence, since $\mathbf{I}$ is an interpretation, $\mathcal{I}^{\underline{b}}(u_i) = c$. Hence $c^{\underline{b}} \equiv d^{\underline{b}}$, and $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ satisfies ϕ.

Suppose $\phi \equiv (X_i \doteq \pi_{a_l} X_j)$, and $\Delta \vdash X_j : (a_1 : \tau_1, \ldots, a_n : \tau_n)$. Then $u_j \in \mathbf{M}^\infty(a_1 : \tau_1, \ldots, a_n : \tau_n)$, and, since $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is complete, there exists a $v \in \mathbf{M}^\infty(\tau_l)$ such that $(v, \pi_{a_l}, u_j) \in \mathbf{A}^\infty$. Hence $\mathbf{I}^{\tau_l}(v) = \mathbf{I}^{\tau_l}(u_i)$. Hence, by lemma 14.13, $(v, u) \in Eq(\mathbf{A}^\infty)$. Hence ξ satisfies ϕ.

The remaining cases are similar. ■

Note, in particular, that lemma 14.15 tells us that, if $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is complete then an instance satisfying the conditions of this lemma is a unique (up to isomorphism) minimal model satisfying **Tr**, and hence **Tr** is complete. Consequently lemmas 14.14 and lemma:compl-inst-to-compl-hmod combine to prove that a **Tr** is complete iff $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is complete.

It remains to show that there is a normal-form transformation program formed by unfolding clauses from **Tr** which generates $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ in a single step.

Unfolding Sequences of Skolemized Clauses

We must first extend our definition of unifiers (definition 14.14) and unfoldings to deal with Skolemized clauses.

Suppose $\Xi \equiv (\Psi_t \Longleftarrow \Phi_t)$ and $\Delta \equiv (\Psi_u \Longleftarrow \Phi_u)$ are Skolemized clauses. A *unifier* from Δ to Ξ is a partial mapping from the variable terms of Φ_t to the variable terms of $\Phi_u \cup \Psi_u$.

We can define the set of clauses $Unfold(\Xi, \Delta, \mathcal{U})$ and unfolding sequences in an analogous way to in section 14.6.

Lemma 14.16: For any $q \in \mathbf{A}^\infty$ either $q \in \mathbf{A}^0$ or there is an unfolding sequence $\Xi_0, \Delta_1, \Xi_1, \ldots,$ Δ_N, Ξ_N, and $\mathcal{U}_1 \ldots, \mathcal{U}_N$, where $\Xi_N \equiv (\Psi \Longleftarrow \Phi)$, such that $\Xi_0 \in \mathbf{Tr}$, $\Delta_i \in \mathbf{Tr}$ for $i = 1, \ldots, N$, and an $\mathcal{I}_{Src}$-environment ρ, such that $[\![\Phi]\!]\mathcal{I}_{Src}\rho = \mathbf{T}$, and $s = \xi(\psi)$ for some $\psi \in \Psi$, where

$Var(\Phi) = X_1, \ldots, X_k$ and ξ is the H-environment $(\rho(X_1), \ldots, \rho(X_k))$. ■

Proof: We will show by induction on i that, for each $q \in \mathbf{A}^i$ there is an unfolding sequence as described in the lemma. For $i = 0$ the result is trivial.

Suppose $i \geq 0$ and we have the result for i, and that $q \in \mathbf{A}^{i+1} \backslash \mathbf{A}^i$. Then there is a transformation clause $\Xi \in \mathbf{Tr}$, $\Xi \equiv (\Psi \Longleftarrow \Phi)$, and an H-environment ξ for $(\mathbf{M}^i, \mathbf{A}^i)$ such that ξ satisfies each $\phi \in \Phi$. and $s = \xi(\psi)$ for some $\psi \in \Psi$.

We need to show that, for any atom ϕ and H-environment ξ such that ξ *satisfies* ϕ in $(\mathbf{M}^i, \mathbf{A}^i)$, there is a finite unfolding sequence which generates $\xi(\phi)$. We will show this for the case where $\phi \equiv (X_i \doteq c^{\underline{b}})$. The other cases are identical in principle (though slightly more complicated to present because of involving two variables).

Suppose $\xi = (u_1, \ldots, u_k)$, $\phi \equiv (X_i \doteq c^{\underline{b}})$ and ξ satisfies ϕ in $(\mathbf{M}^i, \mathbf{A}^i)$. Then there is a $v \in \mathbf{M}^i(\underline{b})$ such that $(v, u_i) \in Eq(\mathbf{A}^i)$ and $(v, c^{\underline{b}}) \in \mathbf{A}^i$. Hence there is a *finite* set $\mathbf{A}' \subseteq \mathbf{A}^i$ such that $(v, u_i) \in Eq(\mathbf{A}')$. Assume $\mathbf{A}'$ is a minimal such set, and let $\mathbf{A}'' = \mathbf{A}' \setminus \mathbf{A}^0$. Suppose $\mathbf{A}'' = \{q_1, \ldots, q_l\}$. Then for $j = 1, \ldots, k$ there exist unfolding sequences $\Xi^j_0, \Delta^j_1, \Xi^j_1, \ldots, \Delta^j_{n^j}, \Xi^j_{n^j}$ and $\mathcal{U}^j_1, \ldots, \mathcal{U}^j_{n^j}$, where $\Xi^j_0 \in \mathbf{Tr}$ and $\Delta^j_i \in \mathbf{Tr}$ for $i = 1, \ldots, n^j$, and H-environments ξ^j for $\Xi^j_{n^j}$, such that $\Xi^j_{n^j} \equiv (\Psi^j \Longleftarrow \Phi^j)$ ξ^j satisfies ϕ' in $(\mathbf{M}^0, \mathbf{A}^0)$ for each $\phi' \in \Phi^j$, and $\xi^{\psi'} = q_j$ for some $\psi' \in \Psi^j$.

Also there is an unfolding sequence, $\Xi'_0, \Delta'_1, \Xi'_1, \ldots, \Delta'_n, \Xi'_n$ and $\mathcal{U}'_1, \ldots, \mathcal{U}'_{n^j}$, and an H-environment ξ' such that $\Xi'_n \equiv (\Psi' \Longleftarrow \Phi')$ and, for each $\phi' \in \Phi'$, ξ' satisfies ϕ' in $(\mathbf{M}^0, \mathbf{A}^0)$ and $\xi'(\psi') = (v, c^{\underline{b}})$ for some $\psi' \in \Psi'$.

Hence we can form an unfolding sequence with clauses $\Xi'_0, \Delta'_1, \ldots, \Delta'_n, \Xi'_n, \Xi^1_0, \Theta^1_0, \Delta^1_1, \ldots, \Delta^1_{n^1}, \Theta^1_{n^1}, \ldots, \Xi^l_0, \Theta^l_0, \Delta^l_1, \ldots, \Delta^l_{n^l}, \Theta^l_{n^l}$ and unifiers $\mathcal{U}'_1, \ldots, \mathcal{U}'_n, \mathcal{U}'^1_0, \mathcal{U}'^1_1, \ldots, U'^l_0, \mathcal{U}'^l_1, \ldots, \mathcal{U}'^l_{n^l}$, and an H-environment ξ'' such that, if $\Theta^l_{n^l} \equiv (\Psi'' \Longleftarrow \Phi'')$ then ξ'' satisfies ϕ'' in $(\mathbf{M}^0, \mathbf{A}^0)$ for each $\phi'' \in \Phi''$, and $\xi''(\psi'') = (u_i, c^{\underline{b}})$ for some $\psi'' \in \Psi''$.

Returning to our original clause $\Xi \equiv (\Psi \Longleftarrow \Phi)$, suppose $\Phi = \{\phi_1, \ldots, \phi_m\}$. Then, using an argument similar to above, for each ϕ_j we can find an unfolding sequence with clauses $\Xi^j_0, \Delta^j_1, \Xi^j_1, \ldots, \Delta^j_{n^j}, \Xi^j_{n^j}$ and unifiers $\mathcal{U}^j_1, \ldots, \mathcal{U}^j_{n^j}$ such that $\Xi^j_0 \in \mathbf{Tr}$ and $\Delta^j_i \in \mathbf{Tr}$ for $i = 1, \ldots, n^j$, and there is an H-environment ξ' such that, if $\Xi^j_{n^j} \equiv (\Psi' \Longleftarrow \Phi')$, then ξ' satisfies ϕ' in $(\mathbf{M}^0, \mathbf{A}^0)$ for each $\phi' \in \Phi'$, and $\xi'(\psi') = \xi(\phi_j)$ for some $\psi' \in \Psi'$.

Hence we can form an unfolding sequence with clauses $\Xi, \Xi^1_0, \Theta^1_0, \Delta^1_1, \Theta^1_1, \ldots, \Delta^1_{n^1}, \Theta^1_{n^1}, \Xi^2_0, \Theta^2_0, \ldots, \Xi^m_0, \Theta^m_0, \Delta^m_1, \Theta^m_1, \ldots, \Delta^m_{n^m}, \Theta^m_{n^m}$ and unifiers $\mathcal{U}'^1_0, \ldots, \mathcal{U}'^1_{n^1}, \ldots, \mathcal{U}'^m_0, \ldots, \mathcal{U}'^m_{n^m}$, and an H-environment ξ'' such that, if $\Theta^m_{n^m} \equiv (\Psi'' \Longleftarrow \Phi'')$ then ξ'' satisfies ϕ'' in $(\mathbf{M}^0, \mathbf{A}^0)$ for $\phi'' \in \Phi''$, and $\xi''(\psi'') = q$ for some $\psi'' \in \Psi''$. ■

Finally we must show that, for each $q \in \mathbf{A}^\infty \setminus \mathbf{A}^0$ there is an unfolding from a description clause which generates q.

If $q \in \mathbf{A}^\infty \setminus \mathbf{A}^0$ then there is a clause $\Delta \in \mathbf{Tr}$, $\Delta \equiv (\Psi \Longleftarrow \Phi)$, and an H-environment ξ such that ξ and $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ satisfy each $\phi \in \Phi$ and $\xi(\psi) = q$ for some $\psi \in \Psi$. Let Θ be a minimal set of atoms such that, if P is a variable term in ψ and $\Delta \vdash P : (C, \mu)$ then $Nav^P_{(C,\mu)} \subseteq \Theta$ and $\Theta \Longleftarrow \Theta$

is a description clause. Then since $(\mathbf{M}^\infty, \mathbf{A}^\infty)$ is complete, we can find an H-environment ξ' such that $\xi'(\theta) \in \mathbf{A}^\infty$ for each $\theta \in \Theta$ and $\xi'(\theta') = \xi(\psi)$ for some $\theta' \in \Theta$. Hence, for each $\theta \in \Theta$, we can find an unfolding sequence and H-environment that generates $\xi'(\theta)$ from $(\mathbf{M}^0, \mathbf{A}^0)$. Using the techniques of the proof of lemma 14.16, we can combine these unfolding sequences to get a sequence starting with $\Theta \Longleftarrow \Theta$ and ending in a normal-form clause (that is with only source terms in the body), which generates q.

This concludes our proof of proposition 14.9. ∎

15 Transformations of Alternative Collection Types

The type system we defined in 7.1 supports *set types* as well as variants, records and class types. However many data-models support other kinds of collection types, in particular *bags*, in which elements have multiplicity, and *lists*, in which elements have both multiplicity and order. Sets, bags and lists, and possibly other kinds of collection type, may be viewed as being based around the same or similar constructors, but with differing semantics. This idea is exemplified in [10] where programming languages based on structural induction are introduced for these three collection types.

Consequently the question arises as to how we can adapt the language *WOL*, and the associated programming techniques, for defining transformations between databases involving bags and lists. We will concentrate on solving this problem for lists, since a solution for lists can then be generalized for bags (and, in fact, sets) via the appropriate coercions.

The problem is that *WOL* is a declarative language, in which there is no implicit concept of order of evaluation or assignment. The traditional presentation of lists involves the constructors *cons* and *nil*, and lists are built using ordered sequences of applications of these constructors. As it stands, *WOL* uses a single predicate $\dot{\in}$ in order to indicate inclusion in a collection type, but does not have any means of indicating the order or multiplicity of elements in a collection. The *cons, nil* presentation of lists is not suitable for inclusion in *WOL* because it requires some kind of recursion in order to construct lists or arbitrary length.

For example suppose we have a target class *Person*, with associated type $\tau^{Person} \equiv (name : str, children : [Person])$. Here $[\tau]$ is used the type of lists of type τ. Suppose our source contains a table *Parents* of type $[(pname : str, cname : str)]$. We could write a clause

$$\begin{array}{l} X = Mk^{Person}(N),\ X.name = N,\ Mk^{Person}(C) \in X.children \\ \quad \Longleftarrow \quad P \in Parents,\ P.pname = N,\ P.cname = C \end{array}$$

Then we would like to have the order of the list of children of a particular person in the *Person* class coincide with the order of the corresponding children in the *Parents* table. So if our *Parents* list was

$$\begin{aligned}
\mathit{Parents} \equiv\ & [(\mathit{pname} = \text{“Susan”}, \mathit{cname} = \text{“Jeremy”}),\\
& (\mathit{pname} = \text{“Susan”}, \mathit{cname} = \text{“Chris”}),\\
& (\mathit{pname} = \text{“Val”}, \mathit{cname} = \text{“Alexander”}),\\
& (\mathit{pname} = \text{“Val”}, \mathit{cname} = \text{“Nicholas”}),\\
& (\mathit{pname} = \text{“Carl”}, \mathit{cname} = \text{“George”}),\\
& (\mathit{pname} = \text{“Victor”}, \mathit{cname} = \text{“Athalia”}),\\
& (\mathit{pname} = \text{“Victor”}, \mathit{cname} = \text{“Mirit”})]
\end{aligned}$$

then we would like our class *Person* to have objects

$$\begin{aligned}
\sigma^{\mathit{Person}} \equiv\ & \{\mathit{Susan}, \mathit{Jeremy}, \mathit{Chris}, \mathit{Val}, \mathit{Alexander}, \mathit{Nicholas},\\
& \mathit{Carl}, \mathit{George}, \mathit{Victor}, \mathit{Athalia}, \mathit{Mirit}\}
\end{aligned}$$

with associated values

$$\begin{aligned}
\mathcal{V}^{\mathit{Person}}(\mathit{Susan}) &= (\mathit{name} = \text{“Susan”}, \mathit{children} = [\mathit{Jeremy}, \mathit{Chris}])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Val}) &= (\mathit{name} = \text{“Val”}, \mathit{children} = [\mathit{Alexander}, \mathit{Nicholas}])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Carl}) &= (\mathit{name} = \text{“Carl”}, \mathit{children} = [\mathit{George}])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Victor}) &= (\mathit{name} = \text{“Victor”}, \mathit{children} = [\mathit{Athalia}, \mathit{Mirit}])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Jeremy}) &= (\mathit{name} = \text{“Jeremy”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Chris}) &= (\mathit{name} = \text{“Chris”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Alexander}) &= (\mathit{name} = \text{“Alexander”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Nicholas}) &= (\mathit{name} = \text{“Nicholas”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{George}) &= (\mathit{name} = \text{“George”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Athalia}) &= (\mathit{name} = \text{“Athalia”}, \mathit{children} = [])\\
\mathcal{V}^{\mathit{Person}}(\mathit{Mirit}) &= (\mathit{name} = \text{“Mirit”}, \mathit{children} = [])
\end{aligned}$$

.

15.1 An alternative representation for lists

Since we are avoiding recursion and the *cons, nil* presentation of lists in our language, we will present an alternative construction for lists which relies on the idea of assigning a *precedence* to each element of a list, representing its position.

We will assume a linearly ordered set $(\mathbf{L}, <)$. The particular linear order we choose does not matter here, though later we'll be settling on the set of strings of natural numbers ordered lexicographically.

Definition 15.1: Suppose D is some set. A **list** over domain D is a partial function $l : \mathbf{L} \overset{\sim}{\to} D$ such that l has a finite domain. ■

If $i \in \mathbf{L}$ and $l(i) = p$ then we say p is in l with **precedence** i. The idea is that, if p and q occur in l with precedences i and j respectively, and $i < j$, then p occurs in the list l before q.

We define the relation $\approx$ on lists to be such that $l \approx l'$ iff there is a bijective function $f : \mathit{dom}(l) \rightarrow \mathit{dom}(l')$ such that if $i, j \in \mathit{dom}(l)$, $i < j$, then $f(i) < f(j)$, and for every $i \in \mathit{dom}(l)$, $l(i) = l'(f(i))$. Note that the relation $\approx$ is an equivalence on lists.

We will consider two lists, l and l', to be equal if $l \approx l'$.

For example, if we take $(\mathbf{L}, <)$ to be the natural numbers with their normal ordering, we could represent the list ["a", "b", "c"] as $(1 \mapsto \text{"a"}, 2 \mapsto \text{"b"}, 3 \mapsto \text{"c"})$, or, equally well, as $(36 \mapsto \text{"a"}, 54 \mapsto \text{"b"}, 63 \mapsto \text{"c"})$, and so on.

15.2 Assigning precedence to list elements

Given our new representation of lists, the problem is now to find a way assign precedences to elements of a list in a target database, based on the precedences of elements of lists in the source database.

We will adopt the same typing rules for terms and atoms involving list types as those given for set types in definitions 13.2 and 13.4.

We will expand the definition of the semantic operator on types (definition 7.2) with

$$[\![[\tau]]\!]\sigma^{\mathcal{C}} \;\equiv\; (\mathbf{L} \overset{\sim}{\rightarrow} [\![\tau]\!]\sigma^{\mathcal{C}})$$

However we will change the definition of the semantic operator on atoms from that in definition 13.10, so that $[\![\cdot]\!]\mathcal{I} : \mathit{Atoms}^{\mathcal{S}} \rightarrow \mathit{Env}(\mathcal{I}) \rightarrow (\{\mathbf{T}, \mathbf{F}\} \cup \mathcal{P}_{fin}(\mathbf{L}))$ and

$$[\![P \dot{\in} Q]\!]\mathcal{I}\rho \;\equiv\; \begin{cases} \mathbf{F} & \text{if } [\![P]\!]\mathcal{I}\rho \neq ([\![Q]\!]\mathcal{I}\rho)(i) \\ & \text{for all } i \in \mathit{dom}([\![Q]\!]\mathcal{I}\rho) \\ \{n \,|\, ([\![Q]\!]\mathcal{I}\rho)(n) = [\![P]\!]\mathcal{I}\rho\} & \text{otherwise} \end{cases}$$

So $[\![P \dot{\in} Q]\!]\mathcal{I}\rho$ is $\mathbf{F}$ if $[\![P]\!]\mathcal{I}\rho$ does not occur in the list $[\![Q]\!]\mathcal{I}\rho$, and is the set of precedences with which $[\![P]\!]\mathcal{I}\rho$ occurs in $[\![Q]\!]\mathcal{I}\rho$ otherwise.

For simple clauses, such as our clause

$$\begin{aligned} X = \mathit{Mk}^{\mathit{Person}}(N),\ X.\mathit{name} = N,\ \mathit{Mk}^{\mathit{Person}}(C) \in X.\mathit{children} \\ \Longleftarrow\ \ P \in \mathit{Parents},\ P.\mathit{pname} = N,\ P.\mathit{cname} = C \end{aligned}$$

this would seem sufficient: we could take the clause to mean, if P is in list *Person* with precedence i, then $\mathit{Mk}^{\mathit{Person}}(C)$ is in the list $X.\mathit{children}$ with precedence i also.

However this does not solve the problem for a clause with multiple $\dot{\in}$ atoms in the body. For example for a clause of the form

$$X \dot{\in} L_1 \;\Longleftarrow\; Y \dot{\in} L_2,\ Z \dot{\in} L_3,\ \Phi$$

where L_1, L_2 and L_3 are lists, it is necessary to combine the precedences of the two $\dot{\in}$ atoms in the body of the clause to find the precedence of the head of the clause.

At this point it is necessary to make some precise decisions about the underlying linear order for lists. We will use the set of strings of natural numbers, $\mathbb{N}^*$, ordered lexicographically. We will also assume that each list occuring in the source database has its precedences taken from $\mathbb{N}$ (or strings of length one). We can make this assumption without loss of generality since for any list l there is an equivalent list $l' \approx l$ with $dom(l') = \{1, \ldots, n\}$ for some n.

Definition 15.2: A **ranked** set of atoms is a set of atoms, Φ, together with an assignment of a *distinct* rank, $r \in \mathbb{N}$, to each atom of the form $P \dot{\in} Q$ in Φ, such that the ranks of atoms in Φ form an initial sequence of the natural numbers. ■

We write $P \in^r Q$ to denote the atom $P \in Q$ with rank r. We also adopt the convention, when writing a sequence of atoms, that the order in which we write the atoms corresponds to their ranks, when the sequence is interpreted as a ranked set of atoms. For example the sequence of atoms

$$X \in Y.a,\ Z \in Y.b,\ Z = ins_d W,\ U \in W$$

would be interpreted as the ranked set of atoms

$$\{X \in^1 Y.a, Z \in^2 Y.b, Z = ins_d W, U \in^3 W\}$$

We will change our definition of clauses (definition 13.7), to say that a clause consists of two *ranked* sets of atoms: the *head* and the *body*.

Definition 15.3: We will define a semantic operator $[\![\cdot]\!]\mathcal{I}_B$ on ranked sets of atoms, such that for any $\mathcal{I}$-environment ρ and ranked set of atoms Φ with atoms of ranks $1, \ldots, k$, $[\![\Phi]\!]\mathcal{I}_B\rho \subseteq \mathbb{N}^*$, by

$$[\![\Phi]\!]\mathcal{I}_B\rho \ \equiv \ \begin{cases} \emptyset & \text{if } [\![\phi]\!]\mathcal{I}\rho = \mathbf{F} \\ & \text{for some } \phi \in \Phi \\ \left\{ n_1 \ldots n_k \;\middle|\; \begin{array}{l} \text{where } n_i \in [\![\phi_i]\!]\mathcal{I}\rho \text{ and } \phi_i \text{ has} \\ \text{rank } i \text{ in } \Phi, \text{ for } i = 1, \ldots, k \end{array} \right\} & \text{otherwise} \end{cases}$$

■

So $[\![\Phi]\!]\mathcal{I}_B\rho = \emptyset$ if any of the atoms in Φ are unsatisfiable, and $[\![\Phi]\!]\mathcal{I}_B\rho$ consists of the set of strings of precedences of ranked atoms in Φ, ordered by rank, otherwise.

For example, if $[\![X \in Y.a]\!]\mathcal{I}\rho = \{4, 7\}$, $[\![Z \in Y.b]\!]\mathcal{I}\rho = \{2, 33\}$, $[\![U \in W]\!]\mathcal{I}\rho = \{15\}$ and $[\![Z = ins_d W]\!]\mathcal{I}\rho = \mathbf{T}$, then

$$[\![\{X \in^1 Y.a,\ Z \in^2 Y.b,\ Z = ins_d W,\ U \in^3 W\}]\!]\mathcal{I}_B\rho \ = \ \{4.2.15,\ 4.33.15,\ 7.2.15,\ 7.33.15\}$$

Note that, for any set of atoms Φ and $\mathcal{I}$-environment ρ, ρ satisfies Φ iff $[\![\Phi]\!]\mathcal{I}_B\rho \neq \emptyset$. In particular, if Φ contains no atoms of the form $P \dot{\in} Q$, and ρ satisfies Φ, then $[\![\Phi]\!]\mathcal{I}_B\rho = \{\epsilon\}$, the set containing only the empty string.

The operator $[\![\cdot]\!]\mathcal{I}_B$ then gives us a way of getting a precedence string from the body of a clause with multiple $\dot{\in}$ atoms. We can use these precedences to order the elements of a list in the head of the clause. So if L_1 and L_2 denoted the lists [“a”, “b”, “c”] and [“d”, “e”, “f”] respectively, then the smallest list L_3 satisfying the clause

$$X \in L_3 \Longleftarrow y \in L_1,\ Z \in L_2, X.\#1 = Y,\ X.\#2 = Z$$

would be

$$[(\text{“a”},\text{“d”}),(\text{“a”},\text{“e”}),(\text{“a”},\text{“f”}),(\text{“b”},\text{“d”}),(\text{“b”},\text{“e”}),(\text{“b”},\text{“f”}),(\text{“c”},\text{“d”}),(\text{“c”},\text{“e”}),(\text{“c”},\text{“f”})]$$

The next problem to face is how to assign precedences to target lists if they occur multiple times in the head of a clause. For example suppose we had a clause

$$X \in L,\ Y \in L \ \Longleftarrow\ \Phi$$

and we found an environment that satisfied Φ. We would want to insert *two* elements into the list L and assign them two distinct precedences. This time we will make use of the ranks on the atoms in the head of the clause to determine the order of insertion.

Definition 15.4: For any $\sigma \in \mathbb{N}^*$, we define the semantic operator $[\![\cdot]\!]\mathcal{I}_\sigma$ on ranked sets of atoms, such that for any ranked set of atoms Φ and $\mathcal{I}$-environment ρ,

$$[\![\Phi]\!]\mathcal{I}_\sigma\rho \ \equiv\ \begin{cases} \mathbf{T} & \text{if, for each } \phi \in \Phi, \text{ if } \phi \text{ is of the form } (P\dot{\in}^r Q) \\ & \text{then } \sigma.r \in [\![\phi]\!]\mathcal{I}\rho \\ & \text{and } [\![\phi]\!]\mathcal{I}\rho = \mathbf{T} \text{ otherwise} \\ \mathbf{F} & \text{otherwise} \end{cases}$$

■

So, for example,

$$[\![\{X \in^1 Y.a,\ Z \in^2 Y.b,\ Z = \mathit{ins}_d W,\ U \in^3 W\}]\!]\mathcal{I}_{(4.2.15)}\rho = \mathbf{T}$$

iff $4.2.15.1 \in [\![X \in Y.a]\!]\mathcal{I}\rho$, $4.2.15.2 \in [\![Z \in Y.b]\!]\mathcal{I}\rho$, $4.2.15.3 \in [\![U \in W]\!]\mathcal{I}\rho$, and $[\![Z = \mathit{ins}_d W]\!]\mathcal{I}\rho = \mathbf{T}$.

Then we could say a clause $\Psi \Longleftarrow \Phi$ is satisfied by an instance $\mathcal{I}$ iff, for any $\mathcal{I}$-environment ρ such that $\mathit{dom}(\rho) = \mathit{Var}(\Phi)$, and any $\sigma \in [\![\Phi]\!]\mathcal{I}_B\rho$, there is an extension of ρ, say ρ', such that $[\![\Psi]\!]\mathcal{I}_\sigma\rho' = \mathbf{T}$.

For example if L_1 represented the list [(“a”,“b”),(“c”,“d”),(“e”,“f”)] then the smallest list L_2 satisfying the clause

$$X.\#1 \in L_2,\ X.\#2 \in L_2 \ \Longleftarrow\ X \in L_1$$

would be [“a”,“b”,“c”,“d”,“e”,“f”].

Note that these definitions cause the elements inserted into a list by different atoms in the head of a clause to be “interleaved”. By changing the definitions slightly we could make it so that all the elements inserted into a list by the first atom in the head of the clause come before all the atoms inserted by the second atom in the list.

There remains a problem, however, if we are dealing with multiple clauses each of which may insert into some target list. For example, suppose we had a transformation program with clauses

$$\begin{aligned} X \in P &\Longleftarrow \Phi_1 \\ Y \in Q &\Longleftarrow \Phi_2 \end{aligned}$$

and we could find environments, ρ_1 and ρ_2 such that $[\![\Phi_1]\!]\mathcal{I}_B\rho_1 \neq \emptyset$ and $[\![\Phi_2]\!]\mathcal{I}_B\rho_2 \neq \emptyset$, and $[\![P]\!]\mathcal{I}\rho_1 = [\![Q]\!]\mathcal{I}\rho_2$. Then we would need to ensure the two clauses insert elements into the list with different precedences.

Suppose **Tr** is a (normal form) transformation program. For each clause $\Delta \in \mathbf{Tr}$ we assign a distinct *rank*, $r \in \mathbb{N}$ to Δ. We will write $\Psi \Longleftarrow^r \Phi$ to represent that the clause $\Psi \Longleftarrow \Phi$ has rank r. When writing transformation programs, we will also assume that the transformation clauses are written in order of ascending rank, so that we will not need to annotate the clauses with their ranks.

We will take the approach that each clause inserts any values into a list before those inserted by any other clauses of higher rank. In other words, if $\Delta_1 \equiv (\Psi_1 \Longleftarrow^{r_1} \Phi_1)$ and $\Delta_2 \equiv (\Psi_2 \Longleftarrow^{r_2} \Phi_2)$ are two clauses in **Tr**, and $r_1 < r_2$, then any elements inserted into a list by clause Δ_1 would come before any inserted by the clause Δ_2. We will ensure this property by prepending the rank r of a clause Δ to any of the precedences of elements inserted into some list by that clause.

Definition 15.5: Suppose $\mathcal{I}$ is an instance and $\Delta \equiv \Psi \Longleftarrow \Phi$ a clause. Then $\mathcal{I}$ is said to **satisfy Δ with rank** r iff, for any $\mathcal{I}$-environment ρ such that $dom(\rho) = Var(\Phi)$, and any $\sigma \in [\![\Phi]\!]\mathcal{I}_B\rho$, there is an extension of ρ, ρ' say, such that $[\![\Psi]\!]\mathcal{I}_{r.\sigma}\rho' = \mathbf{T}$.

An instance $\mathcal{I}$ is said to **satisfy** a transformation program **Tr** iff for every clause $\Delta \in \mathbf{Tr}$, if Δ has rank r, then $\mathcal{I}$ satisfies Δ with rank r. ■

So, if we had a pair of clauses

$$\begin{aligned} X \in L_3 &\Longleftarrow X \in L_1 \\ X \in L_3 &\Longleftarrow X \in L_2 \end{aligned}$$

where L_3 is an expression representing the same list in each clause, then the smallest list L_3 would be the result of appending L_1 and L_2.

Finally, having computed a transformation involving lists, we must replace any lists occuring in the target database with equivalent lists in which the precedences are taken from $\mathbb{N}$, or sequences with length one. This is so that we can compose transformations.

This now gives us all we need to do transformations between databases using lists instead of sets. If we are dealing with source databases which involve both sets *and* lists, then we need to invent some arbitrary precedence for each element of the set: in other words we need to treat sets as lists with some arbitrary ordering on their elements. If we are transforming to a target database involving both sets and list, then we can carry out the transformation as if the target database had only lists, and then throw away any precedence information for sets, inserting elements in an unordered and duplicate-eliminating manner.

A consequence of this is that, if we are transforming from sets to lists, then the transformation may not be deterministic because of the need to choose an arbitrary ordering on the elements of a set. In practice, however, it is likely that we will be able to use some canonical ordering on the elements of a set, so that this will not be a problem.

Part IV

Implementation and Trials of the *WOL* Language

16 Introduction

In this section we will propose an approach to implementing database transformations specified in the language *WOL*, based on the algorithms and techniques suggested in section 14. The implementation works by taking a transformation program written in *WOL* and converting it to a normal form program, which can then be translated into an underlying database programming language. The architecture for such an implementation is illustrated in figure 17. In fact this architecture has already been used for a prototype implementation of a restricted version of *WOL*, which has been applied to some transformations between Human Genome Project Databases (see section 19).

The input to the system is a *WOL* transformation program. The transformation rules will normally be written by the user of the system. However a large numer of important constraints can potentially be derived directly from the meta-data associated with the source and target databases. The kind of contraints that can be derived this way depend on the particular DBMSs being used, but frequently include type information, keys and some other dependencies. Such contraints represent a significant part of a transformation program, but are time consuming and tedious to program by hand. Deriving them directly from meta-data will reduce the amount of grunge work that needs to be done by the programmer, and allow him or her to concentrate on the structural part of a transformation. *WOL* programs written by hand might make use of various shorthands and notational conveniences which would be resolved in the parsing phase.

The translation of a *WOL* transformation program has several stages. First the program is translated into semi-normal form, using the algorithm from the proof of proposition 13.5. Then the snf program is transformed into a normal-form program if possible. This is the most expensive part of the process: a naive implementation of the algorithm suggested in section 14.7 would be intolerably inefficient, and a number of optimizations are required (section 17). Finally

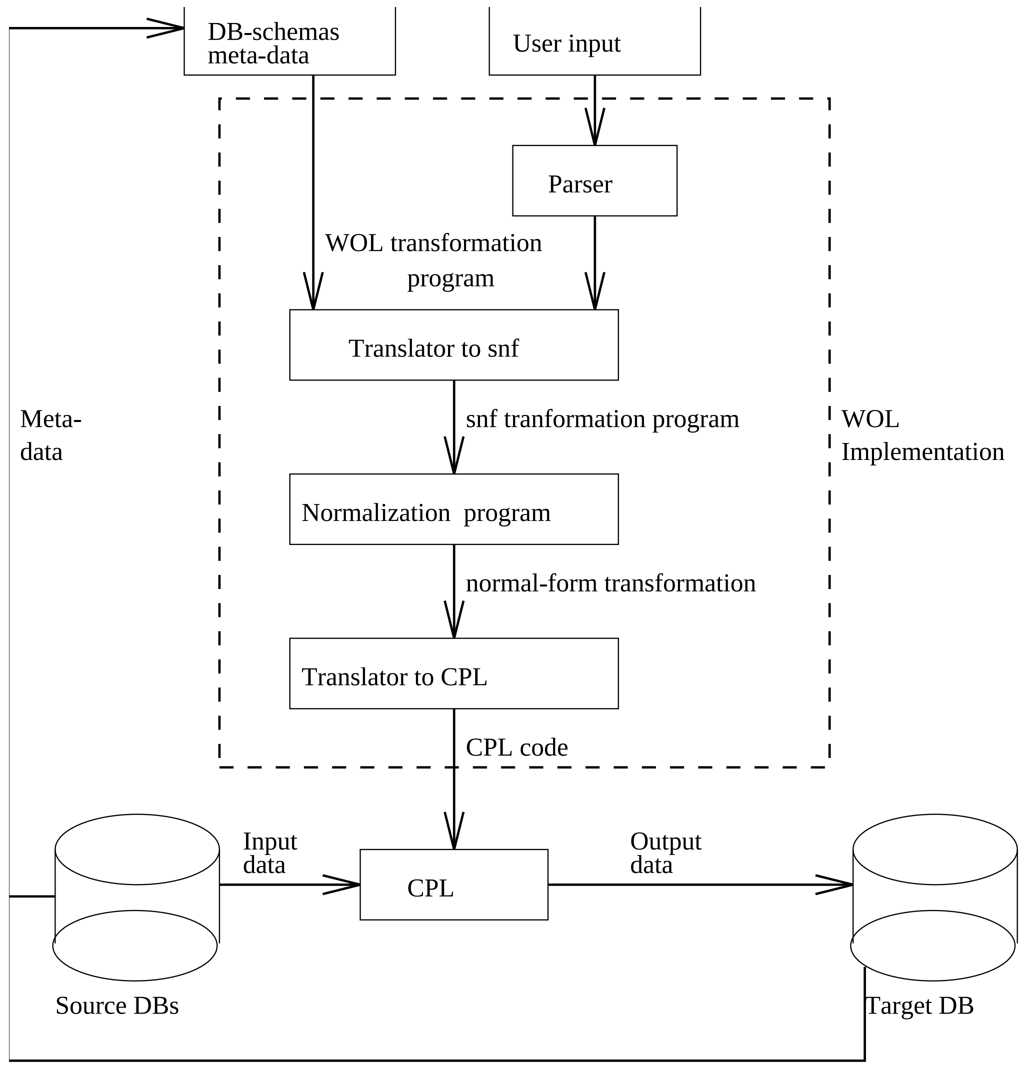


Figure 17: Architecture for proposed implementation of *WOL* transformations

the normal form program is converted into some underlying database programming language, allowing the transformation to be carried out.

Current work has concentrated on translating *WOL* transformation programs into *CPL* [49], a language based on the nested relational model. The reason for using CPL as an implementation language is that it supports many of the data-types that we are interested in, is easily extensible, and can be used to access a variety of different data sources and database systems, including those that we wished to use in our trials. Further, it is an easy task to add additional data drivers to CPL, so that transformations between new database systems can be carried out. However translating normal form *WOL* programs into some other sufficiently expressive DBPL should be a straightforward task, and so the implementation should be easily adaptable to other systems. Beyond adding new data-drivers to CPL, the only major task necessary in order to adapt the *WOL* implementation to new database systems is adding mechanisms for reading meta-data from these systems and translating it into *WOL* constraints.

In the remainder of this section we will discuss the prototype *WOL* implementation, and describe its limitations. In section 17 we will describe a series of optimizations necessary in order to build a pratical implementation of the *WOL* normalization algorithm. In section 18 we describe how variants in a source database schema can lead to exponential blowups in the size of a normal-form transformation program, and in the size of the corresponding program in an underlying database programming language. We show how this problem can be avoided by introducing intermediate data structures and using two-stage transformations. In section 19 we describe some trials in which we applied the prototype implementation to various database transformation problems encountered at the Philadelphia Genome Center for Chromosome 22 at the University of Pennsylvania. We will motivate the transformation problems, and briefly describe the biological domain which they addressed, and will then describe the various programs that were used to implement these transformations. We will describe how the certain different methods of programming the transformation effected the performance of the system and the size of the normal-form programs generated.

16.1 A Prototype Implementation

A prototype implementation, based on a restricted datamodel, has been developed in order to test some of these ideas, and to demonstrate the practical use of such a system. The prototype system was developed in collaboration with the Philadelphia Genome Center for Chromosome 22, at the University of Pennsylvania, and was intended to address database transformation problems arising in a Human Genome Project center.

The prototype implementation made use of a restricted form of the language, *WOL++*, which was based on the nested relational calculus with extensions for simulating object identities. Instead of incorporating the notion of classes into the type system of the language, identities in *WOL++* are attributes which are generated using Skolem functions: an approach proposed in [24]. In addition the prototype is limited to the case where there are no nested sets: a target database is considered to consist of a tuple of sets; each set containing objects built using record and variant constructors, and possibly identities generated using Skolem functions.

There were several reasons for these restrictions, the most significant being that it seemed best to build a prototype for a simplified case in order to get an idea of the problems involved, before attempting to implement the system for more general transformations. The planned Human Genome Project trials of the system involved transformations from various data sources into target flat-relational (Sybase) databases, and so it seemed best to concentrate on these transformations first. In addition the target language, *CPL*, does not incorporate direct support of object-identities and classes, and extending *CPL* to support Skolem functions was a relatively easy task. It is still not clear how best to extend CPL to support general object-identities and classes. In the following sections we will be concentrating on describing a proposed implementation of the full language *WOL*, assuming a simply-keyed and non-nested target schema as described in section 14.5. However most of the optimizations proposed have been succesfully incorporated into the prototype implementation, or address known problems and inefficiencies in this implementation. The trials described in section 19 were carried out using the *WOL++* implementation, but an effort was made to structure the transformation programs and data-structures in a way which would behave in the same manner as a full *WOL* implementation. Further details of the *WOL++* implementation may be found in [16].

17 Optimizing the Normalization Algorithm

Proposition 14.9 suggests an algorithm for converting a semi-normal form transformation program into an equivalent normal form program. Such an algorithm may be summarized as:

1. Generate a set of description clauses;
2. For each description clause generate the maximal unfolding sequences starting from that clause, while testing for recursion;
3. If recursion is detected then raise a "recursive transformation program" error;
4. If any of the clauses resulting from the maximal unfolding sequences are not complete (they contain no target terms in their body but are not in normal form) then raise an "incomplete transformation program" error;
5. Otherwise return those final clauses that are in normal form.

The process of generating the maximal unfolding sequences involves doing a breadth-first unfolding of each description clause on all the transformation rules: a process that is inherently exponential. In addition, at each stage there may be many possible unifiers for each transformation rule, and the number of description clauses is potentially exponential in the number of set and variant type constructs in the schema. It is clear then that such a naive algorithm would be infeasible.

Fortunately we can use various information in order to reduce the search space and restrict our attention to a small subset of relevent unfolding sequences. Many unfolding sequences are equivalent in that they differ only in the order in which rules are applied, but result in the same final clauses, while others are subsumed by more general unfolding sequences, or can never reach a normal form. Our objective in optimizing the normalization process is therefore to construct an algorithm which explores as few equivalent unfoldings as possible, and which discards unproductive unfolding sequences as early as possible.

Avoiding Multiple Description Clauses

If the type associated with a target class by a schema involves multiple variants or set type constructors, then the number of distinct description clauses could be exponential in the size of the type. For example, suppose our target schema consisted of a single class, $\mathcal{C}_{Tgt} \equiv \{C\}$, with associated type

$$\tau^C \equiv (\#a : \langle\!|\#d : \underline{b}, \#e : \underline{b}|\!\rangle, \#b : \langle\!|\#f : \underline{b}, \#g : \underline{b}|\!\rangle, \#c : \{\underline{b}\})$$

Then we would have description clauses:

$$\begin{array}{l}
X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z' \\
\quad \Longleftarrow \quad X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z'; \\
X \in C,\ X.\#a = Y,\ Y = ins_{\#e}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z' \\
\quad \Longleftarrow \quad X \in C,\ X.\#a = Y,\ Y = ins_{\#e}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z'; \\
X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#g}Z' \\
\quad \Longleftarrow \quad X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#g}Z'; \\
X \in C,\ X.\#a = Y,\ Y = ins_{\#e}Y',\ X.\#b = Z,\ Z = ins_{\#g}Z' \\
\quad \Longleftarrow \quad X \in C,\ X.\#a = Y,\ Y = ins_{\#e}Y',\ X.\#b = Z,\ Z = ins_{\#g}Z'; \\
X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z',\ W \in X.\#c \\
\quad \Longleftarrow \quad X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y',\ X.\#b = Z,\ Z = ins_{\#f}Z',\ W \in X.\#c
\end{array}$$

and so on. Clearly there is an overhead in generating so many different clauses, and in generating unfolding sequences for each of them. Instead we can merge the the description clauses into a single clause:

$$\begin{array}{l}
X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y_1,\ Y = ins_{\#d}Y_2, \\
X.\#b = Z,\ Z = ins_{\#f}Z_1,\ Z = ins_{\#g}Z_2,\ W \in X.\#c \\
\qquad \Longleftarrow \ X \in C,\ X.\#a = Y,\ Y = ins_{\#d}Y_1,\ Y = ins_{\#d}Y_2, \\
\qquad\qquad X.\#b = Z,\ Z = ins_{\#f}Z_1,\ Z = ins_{\#g}Z_2,\ W \in X.\#c
\end{array}$$

If this is taken as a regular *WOL* clause then both the body and the head would be unsatisfiable. Instead the various *ins* atoms for each variable are taken as being disjunctive, and, when an atom of the form $X \doteq ins_a Y$ is matched in a description clause, any other atoms of the form $X \doteq ins_b Y'$, where $b \neq a$, are also removed from the clause. For example if we had a transformation clause

$$X \in C,\ X.\#a = Y,\ Y = ins_{\#d} Y' \impliedby \Phi_1$$

and we were to unfold our description clause on it, we would get a clause

$$\begin{aligned}&X \in C,\ X.\#a = Y,\ Y = ins_{\#d} Y',\ X.\#b = Z,\ Z = ins_{\#f} Z_1,\ Z = ins_{\#g} Z_2,\ W \in X.\#c\\ &\quad \impliedby X \in C,\ X.\#b = Z,\ Z = ins_{\#f} Z_1,\ Z = ins_{\#g} Z_2,\ W \in X.\#c,\ \Phi_1\end{aligned}$$

A set inclusion atom in a description clause can potentially be matched many times or not at all. Consequently, if an atom $Y \dot{\in} X$ in a description clause is matched, a new copy of the atom with Y renamed is added to the clause, and similarly copies of any other atoms restricted by Y. So, if the next step in our unfolding sequence was to match on a clause with a transformation clause

$$X \in C,\ W \in X.\#c \impliedby \Phi_2$$

then we would get the clause

$$\begin{aligned}&X \in C,\ X.\#a = Y,\ Y = ins_{\#d} Y',\ X.\#b = Z,\ Z = ins_{\#f} Z_1,\ Z = ins_{\#g} Z_2,\\ &W \in X.\#c,\ W' \in X.\#c\\ &\quad \impliedby X \in C,\ X.\#b = Z,\ Z = ins_{\#f} Z_1,\ Z = ins_{\#g} Z_2,\ W' \in X.\#c,\ \Phi_1,\ \Phi_2\end{aligned}$$

Once maximal unfolding sequences have been found for a description clause, any remaining set inclusion atoms which originated from the description clause, and any other atoms that were matched but remained in the clause because they were needed to maintain range-restriction, are removed. So if the result of our unfolding sequence was a clause:

$$\begin{aligned}&X \in C,\ X.\#a = Y,\ Y = ins_{\#d} Y',\ X.\#b = Z,\ Z = ins_{\#f} Z', W \in X.\#c,\ W' \in X.\#c\\ &\quad \impliedby X \in C,\ W' \in X.\#c,\ \Phi_1,\ \Phi_2,\ \Phi_3\end{aligned}$$

Then this would be rewritten to

$$\begin{aligned}&X \in C,\ X.\#a = Y,\ Y = ins_{\#d} Y',\ X.\#b = Z,\ Z = ins_{\#f} Z', W \in X.\#c\\ &\quad \impliedby \Phi_1,\ \Phi_2,\ \Phi_3\end{aligned}$$

Note that it is only variant insertion atoms and set inclusion atoms that originated in a description clause that can be treated in this special way: any *ins* or $\dot{\in}$ atoms that were introduced by an unfolding must be treated normally. It is therefore necessary to tag those atoms that belonged to the original description clause of an unfolding sequence, and to keep track of which atoms originated from which clauses.

Using Maximal Unifiers

In general there will be many possible unifiers between two clauses. For example if we have a target clause

$$\Psi \;\Longleftarrow\; X = C,\, Y \in X,\, Z =!Y,\, U = \pi_a Z,\, V = \pi_b Z$$

and an unfolding clause

$$X' = C,\, Y' \in X',\, Z' =!Y',\, U' = \pi_a Z' \;\Longleftarrow\; \Phi$$

then there is an obvious unifier, namely

$$\mathcal{U} \;\equiv\; (X \mapsto X',\, Y \mapsto Y',\, Z \mapsto Z',\, U \mapsto U')$$

However there are also many other possible unifiers, such as $(U \mapsto U')$ and $(Z \mapsto Z', U \mapsto U')$ and so on. At each stage we limit our attention to the *maximal* unifiers: that is we unify as many variables as possible at each stage.

Ordering Transformation Rules

In general there may be many equivalent unfolding sequences for a particular target clause, differing only in the order in which they apply clauses. For example, suppose we have a target class C with $\tau^C = (\#a : \underline{b},\, \#b : \underline{b})$, and we are unfolding the clause

$$\Xi \equiv (\Psi \Longleftarrow X \in C,\, X.\#a = Y,\, X.\#b = Z)$$

and transformation clauses

$$\begin{aligned} \Delta_1 \;&\equiv\; (X \in C,\, X.\#a = Y \Longleftarrow \Phi_1) \\ \Delta_2 \;&\equiv\; (X \in C,\, X.\#b = Z \Longleftarrow \Phi_2) \end{aligned}$$

where Φ_1 and Φ_2 contain no target atoms. Then we can unfold Ξ first on Δ_1 and then on Δ_2, or first on Δ_2 and then on Δ_1. In either case the result is

$$\Psi \Longleftarrow \Phi_1, \Phi_2$$

so it is unnecessary to consider both unfolding sequences.

Note, however, that if we have additional clauses then we also need to consider the unfolding sequences involving unfolding Ξ on only Δ_1 or only Δ_2, since unfolding on Δ_1 or Δ_2 may preclude unfolding on some other clause.

We can avoid such multiple unfolding paths by assuming some arbitrary ordering on the transformation clauses of a program. For example, if we decided that Δ_1 came before Δ_2 in the ordering, then we would not attempt to unfold on Δ_1 after having unfolded on Δ_2.

A problem with this approach is that unfolding on one clause might enable an unfolding on another clause which was not proviously possible. For example, suppose we have two transformation clauses:

$$\begin{aligned}\Delta_1' &\equiv (X \in C,\, X.\#a = Y \Longleftarrow \Phi_1)\\ \Delta_2' &\equiv (X \in C,\, X.\#b = Z \Longleftarrow X \in C,\, X.\#a = Y,\, \Phi_2)\end{aligned}$$

where Δ_1' comes before Δ_2' in our ordering, and we were trying to unfold a clause

$$\Xi' \equiv (\Psi \Longleftarrow X \in C,\, X.\#b = Z)$$

Then Ξ' is not unfoldable on Δ_1'. However we can unfold Ξ' on Δ_2' followed by Δ_1', to get the clause

$$\Psi \Longleftarrow \Phi_1, \Phi_2$$

This unfolding sequence should be allowed because the unfolding on Δ_1' involves matching some atoms that were not available at the start of the unfolding sequence.

In order to deal with this problem, we need to *mark* each atom of the unfolding sequence with the set of clauses that have been applied or declined since the atom was introduced. For an unfolding on some tranformation clause Δ to be valid there must be some atom caught by the unfolding which is not yet marked with Δ.

The following psuedo-ML program expresses this algorithm more clearly. The function $\mathit{marks}(\phi)$ returns the set of transformation clauses with which the atom ϕ is marked. The function $\mathit{mark}(\Xi, \Delta)$ returns a clause which is the same as Ξ except that each atom in its body is marked with Δ. $\mathit{unifiers}(\Xi, \Delta)$ returns the set of maximal unifiers from Δ to Ξ, and *Unfold* and *Caught* are the functions described in section 14.6.

```
fun $unfold\_on\_unifier$ $(\Xi, \Delta, \mathcal{U})$ =
        let $CA = Caught(\Xi, \Delta, \mathcal{U})$
        in
             if $(\exists \phi \in CA \cdot \Delta \notin marks(\phi))$        $/*$ new atoms are caught $*/$
             then $Unfold(mark(\Xi, \Delta),\ \Delta,\ \mathcal{U})$
             else $\emptyset$
        end;

fun $apply\_trans\_clause$ $(\Xi, \Delta)$ =
        let $UNS = unifiers(\Xi, \Delta)$
        in
             $\{mark(\Xi, \Delta)\} \cup \bigcup_{\mathcal{U} \in UN} unfold\_on\_unifier(\Xi, \Delta, \mathcal{U})$
        end;

fun $repeat\_apply\_trans\_clause$ $(TCS, \Delta)$ =
        let $NewTCS = \bigcup_{\Xi \in TCS} apply\_trans\_clause(\Xi, \Delta)$
        in
             if $size(TCS) = size(NewTCS)$        $/*$ no new unfoldings $*/$
             then $TCS$
             else $repeat\_apply\_trans\_clause$ $(NewTCS, \Delta)$
        end;

fun $apply\_trans\_prog$ $(TCS, (\Delta{::}Prog))$ =
        $repeat\_apply\_trans\_clause$ $(apply\_trans\_prog(TCS, Prog), \Delta)$
|     $apply\_trans\_prog$ $(TCS,$ nil$)$ = $TCS$;

fun $repeat\_apply\_trans\_prog$ $(TCS, Prog)$ =
        let $NewTCS = apply\_trans\_prog$ $(TCS, Prog)$
        in
             if $size(TCS) = size(NewTCS)$        $/*$ no newly unfoldable clauses $*/$
             then $TCS$
             else $repeat\_apply\_trans\_clause$ $(NewTCS, \Delta)$
        end;
```

So the function *repeat_apply_trans_prog* takes a set of clauses to be unfolded and a *list* of transformation clauses as its arguments. The order imposed on the transformation clauses is taken to be the reverse of the order of the list. The algorithm then repeatedly cycles through the list of transformation clauses, trying to apply each clause as many times as possible, until that clause yields no more unfoldings, at which point it goes onto the next. When none of the transformation clauses yield any new unfoldings then the algorithm terminates.

Note that this algorithm is slightly simplified and does not include tests for recursion.

Avoiding Redundant Unfoldings

It is also possible for an unfolding on one clause to be made redundant by an unfolding on some later clause in an unfolding sequence. For example if we had transformation clauses

$$\begin{array}{lcl}\Delta_1 & \equiv & (Z \in C,\ X = Z.\#a,\ Y = Z.\#b \Longleftarrow W \in D_1,\ X = W.\#a,\ Y = W.\#b)\\ \Delta_2 & \equiv & (Z \in C,\ X = Z.\#a,\ Y = Z.\#c \Longleftarrow W \in D_2,\ X = W.\#a,\ Y = W.\#c)\\ \Delta_3 & \equiv & (Z \in C,\ X = Z.\#a,\ Y = Z.\#b,\ X = Z.\#c \Longleftarrow W \in D_3,\ X = W.\#a,\ Y = W.\#b)\end{array}$$

and we are unfolding a clause

$$\Xi \equiv (\Psi \Longleftarrow Z \in C,\ X = Z.\#a,\ Y = Z.\#b, V = Z.\#c)$$

Then unfolding Ξ first on Δ_1 and then on Δ_3 yields the clause

$$\Psi[Y/V] \Longleftarrow U \in D_1,\ X = U.\#a,\ Y = U.\#b,\ W \in D_3,\ X = W.\#a,\ Y = W.\#b$$

which is subsumed by the clause reached by just unfolding Ξ on Δ_3:

$$\Psi[Y/V] \Longleftarrow W \in D_3,\ X = W.\#a,\ Y = W.\#b$$

Equally an application of Δ_2 would be made redundant by a following application of Δ_3, though an application of Δ_1 could usefully follow an application of Δ_2 and vice versa. We would like to avoid such redundent unfoldings. In order to do so, it is necessary to adapt our algorithm so that an unfolding is not carried out if it makes a previous unfolding redundant. We can make use of the marking of atoms in our target clause to achieve this, by changing the *unfold_on_unifier* function as follows:

```
fun $unfold\_on\_unifier\ (\Xi, \Delta, \mathcal{U}) =$
    let $CA = Caught(\Xi, \Delta, \mathcal{U})$
    in
      if $(\exists \phi \in CA \cdot \Delta \notin marks(\phi))$        /* new atoms are caught */
        $\wedge\neg(\exists \Delta' \cdot \forall \phi \in CA \cdot \Delta' \in marks(\phi))$      /* and no redundant unfoldings */
      then $Unfold(mark(\Xi, \Delta),\ \Delta,\ \mathcal{U})$
      else $\emptyset$
    end;
```

Using Term Paths for Early Rejection of Unproductive Unfolding Sequences

The process of generating the maximal unifiers on a transformation clause, and then checking if a clause is unfoldable on that transformation clause is expensive. We can make use of term paths (section 14.4) in order to rule out many non-applicable transformation clauses early without having to attempt to unify the clauses.

For each transformation clause in a program we can store a list of the term paths of variables occuring in the head of the clause. For our target clause we can keep track of the term paths of *relatively base* variables in the body of the target clause (section 14.6). If the intersection of the two sets of term paths for a target clause and a transformation clause is empty then the target clause will not be unfoldable on the transformation clause.

For example suppose we were unfolding a clause

$$\Xi \equiv (\Psi \Longleftarrow X \in C,\, Y = X.\#a,\, Z = X.\#b)$$

and we had a transformation clause

$$\Delta \equiv (X \in C,\, W = X.\#c \Longleftarrow \Phi)$$

Then the term paths of relatively base variables in Ξ would be $(C, \dot{\in}!\pi_{\#a})$ and $(C, \dot{\in}!\pi_{\#b})$. The set of term paths in the head of Δ would be $\{(C, \epsilon), (C, \dot{\in}), (C, \dot{\in}!), (C, \dot{\in}!\pi_{\#c})\}$. Consequently Ξ is not unfoldable on Δ.

More significantly, we can use such records of term paths in order to detect unfolding sequences which can not reach normal form, and to stop attempting to unfold them at an earlier stage. In the previous example, suppose that there are no transformation clauses which the term path $(C, \dot{\in}!\pi_{\#a})$ in their sets of term paths. Then we will not be able to unfold Ξ to an equivalent normal form clause *even if* there are clauses with the term path $(C, \dot{\in}!\pi_{\#b})$ in their term path list.

We can combine this technique with the markings on atoms of our target clause in order to catch still more of the unproductive unfolding sequences: if Ξ contains a relatively base variable X, such that X has term path (C, μ) and for any ϕ in the body Ξ such that X occurs in ϕ, there is no transfomation clause Δ such that $\Delta \notin marks(\phi)$ and Δ contains a variable with type path (C, μ), then we cannot unfold Ξ to an equivalent normal form clause.

Using Range Restriction to Improve the Efficiency of Unification

Finding the most general unifier between two arbitary sets of atoms in computationally expensive (exponential in the number of atoms). Fortunately we have a great deal of additional information about the atoms of our clauses: that each term is range restricted and that the atoms are in semi-normal form and there are no equality atoms between variables. This means that, rather than representing clauses as pairs of sets of atoms, we can represent them as pairs of *forests* (sets of trees). Each tree in the forests would have a class as its root, variables as its other nodes, and edges marked with the symbols $!, \dot{\in}, \pi_a, \pi_b, \ldots, ins_a, ins_b, \ldots$. Atoms of the forms $X \dot{\neq} Y$ or $X \dot{\notin} Y$ would be represented as additional sets of atoms.

For example the clause

$$\begin{aligned} X = C,\, Y \in X,\, Z =!Y,\, U = \pi_a Z,\, V = \pi_b Z \\ \Longleftarrow \quad W = D,\, Q \in W,\, R =!Q,\, U \in R,\, V \in R, U \neq V \end{aligned}$$

Would be represented by the tree structures

$$\begin{tikzcd}
 & C \arrow[d] & & & & D \arrow[d] & \\
 & X \arrow[d, "\in"] & & \Longleftarrow & & W \arrow[d, "\in"] & \\
 & Y \arrow[d, "!"] & & & & Q \arrow[d, "!"] & \\
 & Z \arrow[dl, "\pi_a"'] \arrow[dr, "\pi_b"] & & & & R \arrow[dl, "\in"'] \arrow[dr, "\in"] & \\
U & & V & & U & & V
\end{tikzcd}$$

$$U \neq V$$

Unifying such trees is in practice considerably more efficient than unifying arbitary sets of atoms: it is still potentially exponential in the number of $\dot{\in}$ atoms, but this is usually small. If we were to limit ourselves to cases where the was at most one atom of the form $X \dot{\in} Y$ for any variable Y then the unification algorithm would be linear in the number of atoms.

Dynamically Altering Transformation Programs

The rules of a transformation program will in general define the objects of one target class in terms of the objects of various other target classes. Consequently it is often necessary to repeat a series of unfoldings in several different unfolding sequences, possibly for different target classes.

For example, suppose our target schema had four classes, $\{C_1, C_2, C_3, C_4\}$, our source schema contained three classes, $\{D_1, D_2, D_3\}$, and our transformation program **Tr** consisted of the clauses:

$$\begin{aligned}
\Delta_1 &\equiv (X \in C_1,\, U = X.\#a,\, V = X.\#b \\
&\qquad\qquad \Longleftarrow Y \in C_3,\, U = Y.\#a,\, V = Y.\#b,\, Z \in D_1,\, U = Z.\#a) \\
\Delta_2 &\equiv (X \in C_2,\, U = X.\#a,\, V = X.\#b \\
&\qquad\qquad \Longleftarrow Y \in C_3,\, U = Y.\#a,\, V = Y.\#b,\, Z \in D_2,\, V = Z.\#b) \\
\Delta_3 &\equiv (Y \in C_3,\, U = Y.\#a, V = Y.\#b \Longleftarrow Z \in C_4,\, U = Z.\#a,\, V = Z.\#b) \\
\Delta_4 &\equiv (Z \in C_4,\, U = Z.\#a, V = Z.\#c \Longleftarrow W \in D_3,\, U = W.\#a,\, V = W.\#b)
\end{aligned}$$

Then in order to get a normal form clause for the class C_1 it is necessary to unfold a description clause on first Δ_1, then Δ_3 and then Δ_4, whereas to find a normal form clause for C_2 it is necessary to unfold on Δ_2 then Δ_3 and then Δ_4, and for C_3 it is necessary to unfold on Δ_3 then Δ_4. Clearly there is some duplicate effort involved here, and we could improve efficiency by "memo-izing" the result of unfolding Δ_3 then Δ_4. In particular, if we first generate a new rule, say Δ_3', by unfolding Δ_3 on Δ_4, and then use this rule in place of Δ_3 in our transformation program, then we would replace the repeated unfolding sequences with single unfoldings on Δ_3'.

We define a **partial normal form** clause to be a clause $\Psi \Longleftarrow \Phi$ such that

1. $\Psi \Longleftarrow \Phi$ is in semi-normal form;
2. Φ contains no target terms; and
3. for any atom of the form $X \dot{\in} Y$ in Ψ, either $X \in \mathit{Var}(\Phi)$ or X is characterized in $\Psi \Longleftarrow \Phi$.

Assuming that we are dealing with a non-nested target schema, the phrase "X is characterized in $\Psi \Longleftarrow \Phi$" here means that

1. if $\vdash X : C$ then $\Psi \cup \Phi$ contains an atom $X \doteq \mathit{Mk}^C Z$,
2. if $\vdash X : \underline{b}$ then $\Psi \cup \Phi$ contains an atom $X \doteq c^{\underline{b}}$,
3. if $\vdash X : (a_1 : \tau_1, \ldots, a_n : \tau_n)$ then $\Psi \cup \Phi$ contains atoms $Z_i \doteq \pi_{a_i} X$ for $i = 1, \ldots, n$, and
4. if $\vdash X : \langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle$ then $\Psi \cup \Phi$ contains an atom $X \doteq \mathit{ins}_{a_i} Z$ for some $i \in 1, \ldots, n$.

So partial normal form clauses are similar to normal form clauses, except that we don't require that they give a full description of a value. Using a partial normal form clause in an unfolding sequence will not lead to any new unfoldings. Consequently we can avoid many repeated unfolding sequences by unfolding rules to partial normal form, and then using the partial normal form rules instead of the original rules.

In the previous example, the rule Δ_3' would be the partial normal form clause

$$\Delta_3' \equiv (Y \in C_3,\, U = Y.\#a, V = Y.\#b \Longleftarrow W \in D_3,\, U = W.\#a,\, V = W.\#b)$$

Note that, in general, unfolding a transformation clause to partial normal form may introduce new atoms into the head. Consequently it is possible that converting rules to partial normal form will increase the number of rules in our program with a particular term path in their head. For example suppose we had rules

$$\begin{array}{lcl}\Delta_5 & \equiv & (X \in C,\, Y = X.\#a \Longleftarrow X \in C,\, Z = X.\#b, \Phi_5) \\ \Delta_6 & \equiv & (X \in C,\, Z = X.\#b,\, W = X.\#c \Longleftarrow \Phi_6\end{array}$$

where Φ_5 and Φ_6 contain no target terms. Then unfolding Δ_5 on Δ_6 returns the partial normal form rule

$$\Delta_5' \equiv (X \in C,\, Y = X.\#a,\, W = X.\#c \Longleftarrow \Phi_5, \Phi_6)$$

If we replaced Δ_5 with Δ_5', there would then be two clauses with heads containing a variable with term path $(C, \in!\pi_{\#c})$ instead of one. Consequently, when unfolding other clauses which had relatively base variables with this term path, there would be more clauses to unfold them on.

This increase in the number of clauses for a particular term path could potentially outweigh any advantage gained by converting rules into partial normal form. However we can avoid this problem by not changing the term path sets associated with a clause when we replace it with partial normal form clauses. In the above example we would then count the term path set of Δ'_5 as being $\{(C, \epsilon), (C, \in), (C, \in!), (C, \in!\pi_{\#a})\}$ — the same as the term path set for Δ_5. It is safe to do this since any new atoms in the head of Δ'_5 would have been generated by some other clause, and will therefore already be implied by other clauses in the transformation program.

Stratifying Transformation Programs

If we are going to convert rules to partial normal form, it is first necessary to decide on an order in which to unfold them. To do so, we first *stratify* the transformation program.

Given a transformation program, **Tr**, we construct its *dependency graph*, $G(\mathbf{Tr})$, as follows:

1. The nodes of $G(\mathbf{Tr})$ are the target classes $\mathcal{C}_{Tgt}$.
2. $G(\mathbf{Tr})$ contains an edge (C', C) iff there is a clause $\Psi \Longleftarrow \Phi$ in **Tr** such that there is an $X \in \mathit{Var}(\Psi) \setminus \mathit{Var}(\Phi)$ with a type path (C, μ), and there is a $Y \in \mathit{Var}(\Phi)$ with a type path (C', μ'), and $C \neq C'$.

A **stratification** of $\mathcal{C}_{Tgt}$ for the transformation program **Tr** is a series of disjoint sets of target classes $\mathcal{C}_1, \ldots, \mathcal{C}_k$ such that

1. $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_k = \mathcal{C}_{Tgt}$;
2. if $C \in \mathcal{C}_i$ and $C' \in \mathcal{C}_{Tgt}$ are such that $G(\mathbf{Tr})$ contains the edge (C', C) but not the edge (C,C') then $C \in \mathcal{C}_j$ for some $j < i$; and
3. if $C \in \mathcal{C}_i$ and $C' \in \mathcal{C}_{Tgt}$ are such that $G(\mathbf{Tr})$ contains the edges (C, C') and (C', C) then $C' \in \mathcal{C}_i$.

For example, for the transformation clauses Δ_1, Δ_2, Δ_3 and Δ_4 described previously, a stratification of the classes $\{C_1, C_2, C_3, C_4\}$ would be $\{C_4\}$, $\{C_3\}$, $\{C_1, C_2\}$.

If we have a stratification $\mathcal{C}_1, \ldots, \mathcal{C}_k$ of the target clauses $\mathcal{C}_{Tgt}$ for a transformation program **Tr**, and $C \in \mathcal{C}_i$, $C' \in \mathcal{C}_j$ where $j < i$, it follows that no transformation clauses for C will be used in the unfolding of C', but transformation clauses for C' may be needed in unfolding C.

Consequently, a good strategy for deciding the order in which to convert transformation clauses to partial normal form is to first stratify the target casses, and, for each strata, unfold all the transformation clauses for classes in that strata before going on to the next strata.

In our example, we would first unfold the clause Δ_4, then Δ_3, and then Δ_1 and Δ_2.

18 Two Stage Transformations

In section 17 we saw that the presence of variant types in a target schema can lead to an exponential blowup in the number of description clauses that need to be unfolded. In this section we will see that variant types in a source type will leed to an exponential blowup in the number of normal form clauses in a transformation. In fact this is a more serious problem than that caused by multiple variants in a target type: in practice, though the number of description clauses for a target schema may be exponential in the number of variant types, most of these description clauses will not unfold to valid normal-form transformation clauses, and, as has already been shown, by reinterpreting the semantics of certain combinations of atoms in a description clause, we can avoid having to generate an exponential number of description clauses. However, if a transformation program is to reflect all the possible combinations of data in a source database, it can be inevitable that the number of normal form clauses will be exponential in the number of variants in a source type, and that all these normal form clauses must be generated. We will propose a method of dealing with this problem which relies on generating various intermediate data-structures and implementing a transformation in two stages. First we will briefly describe how this problem arises when dealing with a variety of data models.

18.1 Variants and Option Types

Though variants themselves are not directly supported by many data-models, almost all data-models do support them in some restricted form. For example variants may be used to encode generalizations in an object-oriented or semantic data-model. A more pervasive source of variants are *optional attributes*. An optional attribute is an attribute that can either take some value or can be undefined or *Null*. Some form of optional attributes is included in almost every data-model: in certain data-models, such as ACEDB [43] virtually every attribute is optional.

In a type-driven model, such as the *WOL* model defined in section 7, a natural way of dealing with optional attributes would be to have an *option* type constructor: for any type τ, a value of type $option(\tau)$ would be either a value of type τ or *none*. The reason *option* types were not included in the type system of definition 7.1 is because they can easily be encoded as variant types: $option(\tau)$ can be considered as a shorthand for $\langle\!| some : \tau, none : unit |\!\rangle$.

Note: this interpretation of option types is not in fact well suited to optional attributes occuring in the target schema of a transformation. This is because we would like a value of option type to default to being *none* if it is not defined to be anything else by a transformation program, and so we would like a *none* value to be less than any defined or *present* value in our ordering on instances. A better interpretation of option types for these purposes would be as sets with cardinality at most one. However, since in this section we are concerned with option types occuring in source schemas, our interpretation of options as variants will suffice.

Since we would like to show that the problem of exponential blowup in the number of normal form transformation clauses is a pervasive one, effecting almost all data-models, we will concentrate on showing how these problems can arrise from source schemas involving only option types and no other variants.

We will interpret $\mathit{option}(\tau)$ as a shorthand for $\langle\!|\mathit{some} : \tau, \mathit{none} : \mathit{unit}|\!\rangle$, none as a shorthand for $\mathit{ins}_{\mathit{null}}()$, and $\mathit{some}(p)$ as a shorthand for $\mathit{ins}_{\mathit{some}}(p)$.

18.2 Variants in Source Types: an Example

Suppose we have a source schemas with a single class C, with associated type $\tau^C \equiv (\mathit{id} : \underline{b}, a_1 : \mathit{option}(\underline{b}), \ldots, a_k : \mathit{option}(\underline{b}))$: that is, objects of class C have an attribute id and k optional attributes $a_1, \ldots, a_k$. Suppose that we were transforming transforming an instance of class C to a new class D with associated type $\tau^D \equiv (\mathit{id} : \underline{b}, b_1 : \underline{b}, \ldots, b_k : \underline{b})$, and that the transformation mapped objects in C to objects in D with the same id attributes, and with the attribute b_i equal to the value of a_i if it is defined, and equal to some default value $c^{\underline{b}}$ if the value of a_i is null. Suppose also that the id attributes are keys for both C and D, so that the key types of C and D are $\kappa^C \equiv \kappa^D \equiv \underline{b}$.

The transformation can be expressed in WOL by the following $2k + 1$ clauses:

$$\begin{array}{l}
X \in D,\ X = \mathit{Mk}^D(Z),\ X.\mathit{id} = Z \ \Longleftarrow\ Y \in C,\ Y.\mathit{id} = Z; \\
X.b_1 = W \ \Longleftarrow\ X \in D,\ X.\mathit{id} = Z,\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_1 = \mathit{some}(W); \\
X.b_1 = c^{\underline{b}} \ \Longleftarrow\ X \in D,\ X.\mathit{id} = Z,\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_1 = \mathit{none}; \\
\quad\vdots \qquad\qquad \vdots \\
X.b_k = W \ \Longleftarrow\ X \in D,\ X.\mathit{id} = Z,\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_k = \mathit{some}(W); \\
X.b_k = c^{\underline{b}} \ \Longleftarrow\ X \in D,\ X.\mathit{id} = Z,\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_k = \mathit{none};
\end{array}$$

However, when converted to normal-form, this transformation program yields 2^k transformation clauses:

$$\begin{array}{l}
X \in D,\ X = \mathit{Mk}^D(Z),\ X.\mathit{id} = Z,\ X.a_1 = c^{\underline{b}}, \ldots, X.a_k = c^{\underline{b}} \\
\quad \Longleftarrow\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_1 = \mathit{none}, \ldots, Y.a_k = \mathit{none}; \\
X \in D,\ X = \mathit{Mk}^D(Z),\ X.\mathit{id} = Z,\ X.a_1 = c^{\underline{b}}, \ldots, X.a_{k-1} = c^{\underline{b}},\ X.a_k = W_k \\
\quad \Longleftarrow\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_1 = \mathit{none}, \ldots, Y.a_{k-1} = \mathit{none},\ Y.a_k = \mathit{some}(W_k); \\
\ \vdots \qquad\quad \vdots \\
X \in D,\ X = \mathit{Mk}^D(Z),\ X.\mathit{id} = Z,\ X.a_1 = W_1, \ldots, X.a_k = W_k \\
\quad \Longleftarrow\ Y \in C,\ Y.\mathit{id} = Z,\ Y.a_1 = \mathit{some}(W_1), \ldots, Y.a_k = \mathit{some}(W_k);
\end{array}$$

In fact this problem is common to any transformation in which the source database involves multiple variants: if the type τ^C of a source class C involves k non-nested variants, each with more than one choice, then it will require at least 2^k clauses to cover all the possible values of an object of class C. Further this is not a problem particular to WOL normal-form transformation programs: the size of a transformation program directly implementing such a transformation written in a language such as CPL or IQL, or any other language which does not allow partial descriptions of values, will be exponential in the number of variants in the source type.

Clearly the aproach we have described so far will leed to unmanageably large transformation programs if our source schema involves many variants or optional types, and some way to avoid these problems is required. In the remainder of this section, we will show that in most cases, if, instead of implementing transformations in a single pass, we introduce some intermediate data structures and perform a transformation in two stages, we can avoid this problem.

18.3 Two-Stage Transformation Programs

Consider again the source schema with a single class C described in section 18.2. Suppose we have a target schema with k classes, $D_1, \ldots, D_k$, each with associated type $\tau^{D_i} \equiv (id : \underline{b},\ b_i : \underline{b})$, and each with the *id* attribute as a key. Consider the transformation between these two schemas given by the clauses:

$$\begin{array}{rcl}
X \in D_1, X = Mk^{D_1}(Z),\ X.id = Z,\ X.b_1 = W & \Longleftarrow & Y \in C,\ Y.id = Z,\ Y.a_1 = some(W); \\
X \in D_1, X = Mk^{D_1}(Z),\ X.id = Z,\ X.b_1 = c^{\underline{b}} & \Longleftarrow & Y \in C,\ Y.id = Z,\ Y.a_1 = none; \\
\vdots & & \vdots \\
X \in D_k, X = Mk^{D_k}(Z),\ X.id = Z,\ X.b_k = W & \Longleftarrow & Y \in C,\ Y.id = Z,\ Y.a_k = some(W); \\
X \in D_k, X = Mk^{D_k}(Z),\ X.id = Z,\ X.b_k = c^{\underline{b}} & \Longleftarrow & Y \in C,\ Y.id = Z,\ Y.a_k = none;
\end{array}$$

Each of these clauses maps part of the source class C to one of the classes D_i. Also note that, once converted to semi-normal form, these clauses will already be in normal-form: so the equivalent normal form transformation program will have exactly $2k$ clauses.

Next consider the transformation from the schema with classes $D_1, \ldots, D_k$ to the schema of section 18.2 with single class D given by the transformation clauses:

$$\begin{array}{rcl}
X \in D,\ X = Mk^D(Z),\ X.id = Z,\ X.b_1 = W_1 & \Longleftarrow & Y_1 \in D_1,\ Y_1.id = Z,\ Y_1.b_1 = W_1; \\
\vdots & & \vdots \\
X \in D,\ X = Mk^D(Z),\ X.id = Z,\ X.b_k = W_k & \Longleftarrow & Y_k \in D_1,\ Y_k.id = Z,\ Y_k.b_k = W_k;
\end{array}$$

These clauses unfold to a normal-form transformation program with a single clause:

$$\begin{array}{l}
X \in D,\ X = Mk^D(Z),\ X.id = Z,\ X.b_1 = W_1, \ldots, X.b_k = W_k \\
\quad \Longleftarrow \quad Y_1 \in D_1,\ Y_1.id = Z,\ Y_1.b_1 = W_1, \ldots Y_k \in D_k,\ Y_k.id = Z,\ Y_k.b_k = W_k;
\end{array}$$

Further, composing this transformation with the previous transformation from C to $D_1, \ldots, D_k$ yields a transformation equivalent to the direct transformation from C to D of section 18.2: by decomposing the transformation into two stages we have encoded it in a total of $2k + 1$ normal form transformation clauses.

18.4 Generality of Two-Stage Decompositions of Transformation Programs

The question naturally arises as to whether it is always possible to decompose a transformation into two stages, using a number of clauses and intermediate classes linear in the number of variant types in the source schema. Unfortunately the answer to this is negative. For example consider once again the source schema with single class C described in section 18.2, and a target schemas with a single class E with associated type $\tau^E \equiv (id : \underline{b},\ b : \underline{b})$.

Suppose that we have 2^k distinct constants of type $\underline{b}$, $c_1, \ldots, c_{2^k}$, and consider the transformation program given by

$$
\begin{array}{l}
X \in E,\ X = Mk^E(Z),\ X.id = Z,\ X.b = c_1 \\
\quad \Longleftarrow\ Y \in C,\ Y.id = Z,\ Y.a_1 = none,\ Y.a_2 = none, \ldots, Y.a_k = none; \\
X \in E,\ X = Mk^E(Z),\ X.id = Z,\ X.b = c_2 \\
\quad \Longleftarrow\ Y \in C,\ Y.id = Z,\ Y.a_1 = some(W_1),\ Y.a_2 = none, \ldots, Y.a_k = none; \\
X \in E,\ X = Mk^E(Z),\ X.id = Z,\ X.b = c_3 \\
\quad \Longleftarrow\ Y \in C,\ Y.id = Z,\ Y.a_1 = none,\ Y.a_2 = someW_2, \ldots, Y.a_k = none; \\
X \in E,\ X = Mk^E(Z),\ X.id = Z,\ X.b = c_4 \\
\quad \Longleftarrow\ Y \in C,\ Y.id = Z,\ Y.a_1 = some(W_1),\ Y.a_2 = some(W_2), \ldots, Y.a_k = none; \\
\quad \vdots \qquad\qquad \vdots \\
X \in E,\ X = Mk^E(Z),\ X.id = Z,\ X.b = c_{s^k} \\
\quad \Longleftarrow\ Y \in C,\ Y.id = Z,\ Y.a_1 = some(W_1),\ Y.a_2 = some(W_2), \ldots, Y.a_k = some(W_k);
\end{array}
$$

It is clear that any composition of transformation programs equivalent to this transformation program will involve at least 2^k clauses, simply because a separate clause is required for each constant c_i.

It appears that the required property, in order to be able to decompose a transformation program into two stages with a linear number of clauses, is that it should somehow be possible to separate the values arising from each of the variants in the source database. Experience shows that most "natural" transformations involving variants can be decomposed in this manner.

It would be desirable to find some reasonably broad restrictions on transformations or transformation programs which guarantee that they can be implemented using a two-stage transformation with linearly many clauses, and further to find ways of automatically generating intermediate data-structures from transformation programs satisfying such restrictions. It seems likely that such restrictions and algorithms are possible. However they are beyond the scope of this thesis. A particularly useful result in this direction would be a proof of the following conjecture:

Conjecture 18.1: Given any complete (but not necessarily normal-form) *WOL* transformation program, it is possible to encode the same transformation using the composition of two normal-form *WOL* programs with size polynomial in the size of the original transformation program. ∎

Unfortunately the investigation of this conjecture is also beyond the scope of this thesis.

19 Trials

In order to test the methods of implementing transformations described in the previous sections, a series of trials were carried out using the prototype implementation described in section 16.1. The trials were based on practical transformation problems arising in the Philadelphia Genome Center for Chromosome 22 at the University of Pennsylvania, and involved tranforming data between a variety of molecular biology databases.

Much of the work on optimizing the normalization procedure described in section 17 was inspired by these trials, and many the optimizations were incorporated in the prototype. Unfortunately there was not sufficient time available to re-implement the normalization program using the optimized data-structures described in section 17, which would have lead to considerable improvements in performance. However the performance improvements yielded by such data-structures would have been relatively linear, so that the trials carried out with the prototype implementation still provide a good measure of the relative performance of the system with different transformation programs.

As already mentioned, the prototype implementation used a restricted form of the *WOL* language, *WOL*++, in which object identities were simulated using Skolem functions and identity-attributes [24]. In order to get an accurate idea of how the system would perform on the full *WOL* language, it was therefore necessary to write transformation programs in such a way the the identity attributes and Skolem function applications closely mimicked the proposed use of object identities in *WOL*. However an added benefit of the prototype implementation was that we could experiment with transformations without using identity, and thus get a measure of the effect that identities have on the performance of the normalization algorithm and the generated transformation programs.

The transformation trials concentranted on transformations from GDB [34] to the Philadelphia Genome Center for Chromosome 22's local laboratory notebook database, Chr22DB, and from Chr22DB to a local database, ACe22DB at the Sanger Centre based in Cambridge, England. GDB is an archival database maintained at Johns Hopkins University based on the Sybase relational database system. We will concentrate on describing the Chr22DB to ACe22DB transformations, since these are structurally more interesting, and since the investigation of different implementations and their relative performance was most thorough for these transformations.

19.1 Transforming from ACe to Chr22DB

ACe22DB is the laboratory notebook database for the chromosome 22 mapping effort at the Sanger Centre, Cambridge, England, and Chr22DB is the laboratory notebook database for the Philadelphia Genome Center for Chromosome 22. Both of these databases store experimental data and sequencing information on Chromosome 22 generated at the sites, as well as data imported from other sites and archival databases. These two centers are working on the mapping of the chromosome 22 of the Human Genome. The databases are used in planning experiments as well as in the mapping of the genome. So as to reduce duplicated effort the two sites are collaborating and attempting to share their existing data. Unfortunately the two databases are very different, being based on different and incompatible data-models, and on differing interpretations of the data and how it should be structured.

The ACe22DB database uses the ACeDB data-model and DBMS. ACeDB represents data in tree-like structures with object identities providing a means of cross-referencing between these trees. In ACeDB data structures are often deeply nested, and every attribute is either set-valued or optional. However it does not have support for general variants. Consequently ACeDB is particularly well suited to representing *sparsely populated data* or data where many attributes may be omitted. ACeDB has gained considerable sucess in the molecular biology community, in part because it is well suited to genomic and molecular biology data, and in part because of the availability of a variety of popular tools for viewing and analyzing such data.

The Chr22DB database is implemented using Sybase, a commercial relational database system augmented with system-generated identities and triggers for enforcing constraints.

A Very Short Biology Lesson

Before proceeding to describe the transformation from ACe22DB to Chr22DB, it is perhaps worthwhile to digress briefly in order to explain enough biology to understand the examples. For a more in-depth explanation of the molecular biology and genetics involved the reader is recommend to consult [20].

A chromosome consists of a long double-stranded molecule of deoxyribonucleic acid (DNA), each strand of which is in turn made up of a string of *nuclides* of *bases*. There are four different possible bases, named A, C, G and T, which are arranged in complementary pairs along the DNA molecule: A's opposite T's and C's opposite G's. Consequently a chromosome may be represented as a long string over the alphabet $\{A, C, G, T\}$. The ultimate goal of the Human Genome Project is to *sequence* the twenty four chromosomes that comprise the human genome: that is to determine the order in which A's, C's, G's and T's occur in each chromosome (about 3 billion base pairs). Unfortunately technology does not yet exist to sequence intervals of DNA consisting of more than a few hundred base pairs in one go. Consequently, the Human Genome Project has set itself the short term goal of *mapping* the human genome.

The process of mapping a chromosome involves identifying various short "landmark" sequences of DNA or *probes* within the chromosome, and determining their relative locations. In order to map a chromosome, it is first cut into various random, overlapping pieces of a manageable size (between 50,000 and 1 million bases) using various enzymes. These fragments of DNA are then tested to see if they contain certain probes as subsequences. If two fragments contain the same probe it can be deduced that they overlap, and this information can be used in order to find the relative order of the fragments, and the possition of the probes on the chromosome. This process is illustrated in figure 18.[6]

The data we will be dealing with in our transformations are for a kind of probe known as

[6]This figure was made by David Searls

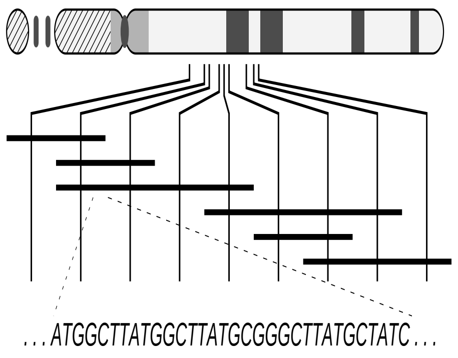


Figure 18: Physical Mapping of a Chromosome

sequence tag sites (STS's). An STS is an interval of DNA which is identified by a pair of *oligos* or *primers*, which are very short sequenced intervals of DNA representing the two ends of the STS. One can test whether an STS is a subsequence of a DNA interval by attempting to induce a chemical reaction called *amplification by polymerase chain reaction* (PCR amplification) on the interval in the presence of the pair of primers of the STS. A PCR-amplification reaction involves several distinct stages, each carried out at different temperatures. The reaction will only be sucessful if the DNA interval contains the STS.

```
(#STS: {(#name: str,
         #locus_symbol: str,
         #STS_length_lo:~ int,
         #STS_length_hi:~ int,
         #oligo1: str,
         #oligo2: str,
         #id: Id,
         #annealing_temp_time:~ str
       )},
 #STS_YAC: {(#id: Id,
             #yac: str,
             #result: str
           )}
 );
```

Figure 19: Type of view of Sybase database representing STS's

In order to store and reproduce fragment of DNA or probe it is inserted into an artificially produced chromosome known as a *vector*. A vector is formed by taking a chromosome from some small organism, such as yeast or bacteria, spliting it, and inserting the stretch of DNA of interest. The vector is then put back into the host organism, which will reproduce, thus reproducing the artificially inserted DNA. The process of reproducing fragments of DNA of interest in this manner is called *cloning.* The most common kind of vector, now used almost exclusively in human genome centers, is a *yeast artificial chromosome* (YAC). YACs have taken over from bacteria-based vectors because they can support much longer stretches of cloned DNA.

The data stored about a particular STS includes the sequences of the two primers of the STS and information about the origins of the primers, the name of the STS and the laboratory from which named it, the experimental conditions for each stage of the PCR amplification associated with the STS and the expected outcome of the PCR reaction, and cross references to various archival databases such as GDB. In addition data is stored on the YACs which were tested to see if their cloned DNA inserts contained the STS, and the results of the tests.

```
primitive srcdata == (
   #STS: {(#name: "stD22S43",
           #locus_symbol: "D22S43",
           #STS_length_lo: <#some: 103>,
           #STS_length_hi: <#some: 103>,
           #oligo1: "GTTCTGGGGAGTGGAGACTC",
           #oligo2: "TAACTGGGCTCTGATTCACC",
           #id: 1358,
           #annealing_temp_time: <#some: "60C">)},
   #STS_YAC:{(#id: 1358,
              #yac: "M1829:g-9",
              #result: "negative"),
             (#id: 1358,
              #yac: "M1925:g-12",
              #result: "positive")}
   );
```

Figure 20: Sample STS and YAC data for Chr22DB

The ACe22DB and Chr22DB Schemas

Rather than taking a schema to be a set of separate class definitions, the *WOL*++ prototype assumed that each database schema consisted of a single type: a tuple with an element for each class or table in the database. Figure 19 shows the type of the view of Chr22DB representing STS data. There are two tables representing STS's and YAC's. The two oligos determining an STS are represented by the attributes `#oligo1` and `#oligo2` which store their sequences as strings (e.g. "ACGGCTCGC..."). The notation `:~` represents an optional attribute, so `#STS_length_lo:~ int` can be considered to be a short hand for `#STS_length_lo: <|none:(), some: int|>`. Some sample source data for this schema is shown in figure 20.

The type of the target ACe database is shown in figure 19.1. Here the type constructor `ftype(f_sts)` represents the type of the range of the Skolem function `f_sts`: the actual type used to implement such Skolem functions is dependent on the database system, (strings in the

```
(#STS:  {(#key: ftype(f_sts),
          #GDB_id:~ str,
          #Oligo_1:~ ftype(f_oligo),
          #Oligo_2:~ ftype(f_oligo),
          #STS_length:~ (#1: int, #2:~ int),
          #Annealing_temp_time:~ str,
          #Phil_positive_YAC:~ ftype(f_yac),
          #Negative_YAC:~ f_type(f_yac)
        )},
 #YAC:  {(#key: ftype(f_yac),
          #Phil_positive_STS:~ str,
          #Negative_STS:~ str
        )},
 #GDB_id: {( #key: str,
             #locus_symbol:~ str,
             #Positive_STS:~ ftype(f_sts)
          )},
 #Oligo: {( #key: ftype(f_oligo),
            #sequence:~ str,
            #STS1:~ ftype(f_sts),
            #STS2:~ ftype(f_sts)
         )}
)
```

Figure 21: Type of ACe22DB schema representing STS's

case of ACeDB), but for type-checking purposes each Skolem function is considered to have its own distinct type. In fact these types may be considered to correspond to the classes of the database. In ACe each class has an implicit `#key` attribute which is required, and all other attributes are either optional or set valued. Figure 19.1 shows the translation of an ACe schema into the nested relational model: an actual ACe schema has a tree-like structure, as shown in figure 19.1. The left-most labels, preceded by question-marks, in this schema represent the classes of the schema. An "external reference" to an object of some class from within an object of another class, is marked by the class name followed by the keyword `XREF`. The keyword `UNIQUE` is used to mark single valued attributes, and all the other labels mark attributes or types at various levels of nesting. There is, in addition a interdependency between attributes on the same line of an ACe schema: for example the second integer in the `#STS_length` tuple can not be defined unless the first integer in the tuple is also defined.

```
//#STS#

?STS    General GDB_id ?GDB_id XREF Positive_STS  // use for D numbers
        PCR     Oligo_1  ?Oligo XREF STS1
                Oligo_2  ?Oligo XREF STS2
                STS_length      UNIQUE Int UNIQUE Int
                Annealing_temp_time  ?Text ?Text
        Positive Phil_positive_YAC      ?YAC XREF Phil_positive_STS
        Negative_results_here
        Negative Negative_YAC           ?YAC XREF Negative_STS

//#YAC#

?YAC    Positive Phil_positive_STS      ?STS XREF Phil_positive_YAC
        Negative_results_here
        Negative Negative_STS           ?STS XREF Negative_YAC


?GDB_id  locus_symbol ?Text
         Positive Positive_STS          ?STS XREF GDB_id

//#Oligo#

?Oligo  Sequence UNIQUE Text    // verbatim sequence - useful
        STS STS1 ?STS XREF Oligo_1
            STS2 ?STS XREF Oligo_2
```

Figure 22: ACe Schema for the ACe22DB Database

The Transformation Programs

Various different forms were tried for writing the transformation programs from Chr22DB to ACe22DB. These divided into two basic types: *"direct"* programs and *"external-reference"* programs. The transformation clauses of the direct version were all already in partial normal-form: that is, they contained no target atoms in their bodies. Consequently normalizing the direct versions of the program involved no sequences of multiple unfoldings, and resulted in the smallest possible number of normal form clauses. The time taken to normalize direct programs shows us how much time the system is taking on tasks such as parsing, type-checking, converting clauses to semi-normal form, building description clauses and so on. In a sense this provides a upper bound on the performance we might hope to get out of the system.

The *external reference* versions of the program included the various external reference constraints of the ACe schema, and used these to generate many of the attributes of certain classes, rather than deriving them directly from the source database. These version of the program lead to a considerable number of unfoldings in order to build partial normal form clauses, and also to a larger resulting number of clauses.

In addition the programs were tried using system generated Skolem functions and using externally defined functions in order to generate key attributes. Unlike Skolem functions, the normalization algorithm makes no assumptions about whether or not an externally defined function is injective. If `f` is an externally defined function then a clause can contain atoms `X = f(Y)` and `X = f(Z)` where `Y` and `Z` are distinct variables, whereas, if `f` were a Skolem function the system would force the variables `Y` and `Z` to be unified into a single variable. Versions of the program using external functions but with additional constraints representing that the external functions were injective were also tried, in order to test whether the observed differences between the performance of the system with programs based on Skolem functions and those based on external functions were indeed due to the injectivity of Skolem functions, and not due to implementational differences between the two.

The following is the *WOL++* code used to implement the direct version of the transformation using Skolem functions.

```
% Function symbol declarations

fun f_oligo;
fun f_sts;
fun f_yac;


KEY
X = Y <== X(#id = I) in #C22_STS, Y(#id = I) in #C22_STS;

KEY
X = Y <== X(#id = I, #yac = A) in #C22_STS_YAC,
          Y(#id = I, #yac = A) in #C22_STS_YAC;


KEY
ERROR
   <== (undef( #STS_length_lo ),
        #STS_length_hi = X) in #C22_STS;
```

```
% TRANSFORMATIONS
% Required fields in C22_STS

(#key = f_sts(N, I),
 #GDB_id = N,
 #Oligo_1 = f_oligo(O1),
 #Oligo_2 = f_oligo(O2)) in #STS
   <==
       (#name = N,
        #oligo1 = O1,
        #oligo2 = O2,
        #id = I) in #C22_STS;

% Optional fields in C22_STS

(#id = f_sts(N, I),
 #STS_length = (#1 = LO, #2 = HI) ) in #STS
   <==
       (#id = I,
        #name = N,
        #STS_length_lo = LO,
        #STS_length_hi = HI) in #C22_STS;

(#id = f_sts(N, I),
 #STS_length = (#1 = LO) ) in #STS
   <==
       (#id = I,
        #name = N,
        #STS_length_lo = LO) in #C22_STS;


% Oligo

(#key = f_oligo(S1),
 #sequence = S1,
 #STS1 = f_sts(N, I)) in #Oligo
   <==
       (#id = I,
        #name = N,
        #oligo1 = S1) in #C22_STS;

(#key = f_oligo(S2),
 #sequence = S2,
 #STS2 = f_sts(N, I)) in #Oligo
   <==
       (#id = I,
        #name = N,
        #oligo2 = S2) in #C22_STS;
```

```
% YAC

(#key = f_yac(S, Y),
 #Phil_positive_STS = f_(N, S)) in #YAC
   <==
        (#id = S,
         #yac = Y,
         #result = "positive") in #C22_STS_YAC,
        (#id = S,
         #name = N) in #C22_STS;

(#key = f_yac(S, Y),
 #Negative_STS = f_sts(N, S)) in #YAC
   <==
        (#id = S,
         #yac = Y,
         #result = "negative") in #C22_STS_YAC,
        (#id = S,
         #name = N) in #C22_STS;

% GDB_id

(#key = N,
 #locus_symbol = L,
 #Positive_STS = f_sts(N, I)) in #GDB_id
   <==
        (#name = N,
         #locus_symbol = L,
         #id = I) in #C22_STS;
```

This code involves a variety of notations that require explanation: statements of the form `fun f_sts;` are declarations of Skolem functions; lines starting with a `%` are comments, and the atom `ERROR` represents an invalid database state, and is equivalent to the atom False in *WOL*. Further the code uses *composite terms*: the term `X(#id = I, #yac = A)` may be considered as shorthand for the term `X` together with the atoms `X.#id = I`, and `X.#yac = A`. In general a composite term consists of a term followed by a list of atoms enclosed in paranthesis. The value of a composite term is the same as the value of its leading term, but it is also taken as asserting the following atoms. Further if an un-qualified attribute name occurs in the atom list of a composite term, then it is taken as refering to an attribute of the leading term. The leading term of a composite term may also be omitted, in which case it is assumed to be an implicit variable. For example the atom

```
(#name = N,
 #locus_symbol = L,
 #id = I) in #C22_STS
```

is shorthand for the atoms `_X.#name = N`, `_X.locus_symbol = I`, `_X.#id = I` and `_X in #C22_STS`, where `_X` is a system-generated variable.

For comparison the following version of the tranformation program makes use of external reference constraints, and of externally defined functions in order to generate keys.

```
% Function symbol declarations

fun f_oligo;
external f_YacName: str = "YacName( #1 )";
external f_StsKey: str = "StsKey( #1, #2 )";


%SOURCE KEY DEPENDENCIES

KEY
X = Y <== X(#id = I) in #C22_STS, Y(#id = I) in #C22_STS;

KEY
X = Y <== X(#id = I, #yac = A) in #C22_STS_YAC,
          Y(#id = I, #yac = A) in #C22_STS_YAC;

KEY
ERROR <== (undef( #STS_length_lo ),
           #STS_length_hi = X) in Src.#C22_STS;

% Same thing can't be an oligo1 and an oligo2

KEY
ERROR <==
        (#oligo1 = S) in #C22_STS, (#oligo2 = S) in #C22_STS;


% f_StsKey is one-to-one
KEY
X = Y <==
        X(#id = I1, #name = N1) in #C22_STS,
        Y(#id = I2, #name = N2) in #C22_STS,
        f_StsKey(N1, I1) = f_StsKey(N2, I2);

% f_YacName is one-to-one
KEY
X = Y <==
        (#yac = X) in #C22_STS_YAC,
        (#yac = Y) in #C22_STS_YAC,
        f_YacName(X) = f_YacName(Y);
```

```
% #yac is a key for #C22_STS_YAC
KEY
X = Y <== X(#yac = A) in #C22_STS_YAC,
          Y(#yac = A) in #C22_STS_YAC;


% XREF

(#key = K, #Phil_positive_YAC = Y) in #STS <==
   (#key = Y, #Phil_positive_STS = K) in #YAC;

(#key = K, #Negative_YAC = Y) in #STS <==
   (#key = Y, #Negative_STS = K) in #YAC;

(#key = S, #GDB_id = G) in #STS <==
   (#key = G, #Positive_STS = S) in #GDB_id;

(#key = S, #Oligo_1 = O) in #STS <==
   (#key = O, #STS1 = S) in #Oligo;

(#key = S, #Oligo_2 = O) in #STS <==
   (#key = O, #STS2 = S) in #Oligo;



% TRANSFORMATIONS

% STS

(#key = f_StsKey(N, I),
 #STS_length = (#1 = LO, #2 = HI) ) in #STS
   <==
       (#id = I,
        #name = N,
        #STS_length_lo = LO,
        #STS_length_hi = HI) in #C22_STS;

(#key = f_StsKey(N, I),
 #STS_length = (#1 = LO) ) in #STS
   <==
       (#id = I,
        #name = N,
        #STS_length_lo = LO,
        undef( #STS_length_hi )) in #C22_STS;
```

```
(#key = f_StsKey(N, I),
 #Annealing_temp_time = A) in #STS
   <==
       (#id = I,
        #name = N,
        #annealing_temp_time = A) in #C22_STS;


% Oligo

(#key = f_oligo( S1 ),
 #Sequence = S1,
 #STS1 = f_StsKey(N, I)) in #Oligo
   <==
       (#id = I,
        #name = N,
        #oligo1 = S1) in #C22_STS;

(#key = f_oligo( S2 ),
 #Sequence = S2,
 #STS2 = f_StsKey(N, I)) in #Oligo
   <==
       (#id = I,
        #name = N,
        #oligo2 = S2) in #C22_STS;


% YAC

(#key = K,
 #Phil_positive_STS = f_StsKey(N, S)) in #YAC
   <==
       (#id = S,
        #yac = Y,
        #result = "positive") in #C22_STS_YAC,
          K = f_YacName(Y),
       (#id = S,
        #name = N) in #C22_STS;

(#key = K,
 #Negative_STS = f_StsKey(N, S)) in #YAC
   <==
       (#id = S,
        #yac = Y,
        #result = "negative") in #C22_STS_YAC,
          K = f_YacName(Y),
       (#id = S,
        #name = N) in #C22_STS;
```

```
% GDB_id

(#key = N,
 #locus_symbol = L,
 #Positive_STS = f_StsKey(N, I)) in #GDB_id
   <==
       (#name = N,
        #locus_symbol = L,
        #id = I) in #C22_STS;
```

The lines starting with the keyword `external` are *external function declarations.* For example the declaration

```
       external f_YacName: str = "YacName( #1 )";
```

declares `f_YacName` to be a function symbol with range type `str`. Applications of `f_YacName` are not interpreted by the normalization algorithm, but are translated to the CPL code of the string of the `external` declaration. For example a term `f_YacName(X)` would be translated to the CPL expression `YacName(X)`.

Figure 23 compares some of the various programs used to implement the Chr22DB to ACe22DB transformation. The *STS.direct* program was a direct implementation of the transformation using externally defined functions, while the *STS.dirid* program is a direct version using Skolem functions. The *STS.xref*, *STS.constr* and *STS.id* versions of the program all make use of the ACe external reference constraints in order to instantiate the target tables. The *STS.xref* program uses external functions to generate keys, while the *STS.constr* program augments *STS.xref* with constraints asserting that the external functions are injective. The *STS.id* program uses Skolem functions to generate identities for each class, and, off the various programs, is the best approximation to a transformation program implemented using the full version of *WOL*.

The second column of the chart shows the time taken by the *WOL*++ implementation to parse, type-check and normalize the transformation programs and then to generate the corresponding CPL programs. The implementation was written using Standard ML of New Jersey. The times quoted were measured using the unix *timex* utility, and are the "user" times, that is the times taken by the process, rather than the "real" times which included other system activity and were generally about ten percent longer. The third column reprents the number of CPL primitive declarations in the resulting CPL program, and thus provide a measure of the size of the CPL program. In general there was one CPL primitive generated by each normal-form clause, plus some primitive handling Skolem functions, so this figure also provides a measure of the size of the normal-form *WOL*++ transformation program. The last column shows the time taken to load the resultant program into CPL and run it on some test data.

One at first surprising feature of these results is that the CPL and normal form programs generated by the external reference versions of the transformation are much larger than those generated by the direct versions. The reason for this is that there are more clauses defining each classes in the external reference versions: there are three clauses for the class `#STS` in

Program	Normalizing *WOL*++ Time	Number of CPL Primitives	Compiling CPL Time
STS.direct	5:19.61	61	1:59.02
STS.dirid	5:33.63	37	55.94
STS.xref	1:53:32.70	515	36:11.74
STS.constr	35:24.14	233	4:26
STS.id	37:28.16	205	10:33.01

Figure 23: Comparison of performance of various versions of the Chr22DB to ACe22DB transformation program

the *STS.dirid* program, while there are eight in the *STS.constr* program. These clauses can be combined in various ways in order to generate equivalent but destinct normal-form clauses. Because the attributes of the target classes are all optional, there are also normal-form clauses that only describe parts of a target value, constructed using subsets of these clauses. Having many alternative ways of deriving the same value in a target database is reflected by having many normal-clauses for the same class in the normalized program.

Another notable feature of these results is that constraints, and particularly a notion of identity, are essential in order to keep the performance of the system tolerable. The external reference version of the program without constraints took almost four times as long to normalize as the version with constraints, and this problem was found to become more severe when dealing with larger programs. An advantage of the full *WOL* language is that it imposes identity constraints on the source and target classes in a natural way, and because the identity is treated separately form other constraints, it can be implemented in a more efficient manner.

The times for compiling CPL code shown in the last column of figure 23 were in fact the times for the second *WOL*++ to CPL translation that was implemtmented. The first version of the translation created a CPL primitive corresponding to each normal-form clause, and then a CPL primitive for each target classes unioning the values generated by the relevent clauses. For example suppose we had a normal form program:

$$\begin{array}{rcl} X \in C,\ X.\#a = Y,\ X.\#b = Z & \Longleftarrow & W \in D,\ W.\#a = X,\ W.\#b = Z; \\ X \in C,\ X.\#a = Y,\ X.\#b = Z & \Longleftarrow & W \in E,\ W.\#a = X,\ W.\#c = Z; \\ X \in C,\ X.\#a = Y,\ X.\#b = Z & \Longleftarrow & W \in F,\ W.\#a = X,\ W.\#d = Z; \end{array}$$

where C is a target class and D, E and F are source classes. This would have been translated into the CPL program

```
primitive mk_C_1 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #b\Z) <- S.#D };
primitive mk_C_2 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #c\Z) <- S.#E };
primitive mk_C_3 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #d\Z) <- S.#F };
```

```
primitive mk_C == \S =>
        (mk_C_1 @ S) + (mk_C_2 @ S) + (mk_C_3 @ S);
```

The size of the last union primitive is linear in the number of clauses in the normal form program. However it was found that the time taken to compile load primitive declarations of this form in CPL was non linear in the size of the declaration. Consequently the time taken to compile the CPL generated by the *STS.dirid* program (37 clauses) using this first translation was 55 seconds, while the time for the *STS.id* program (205 clauses) was 51 minutes, and the time for the *STS.xref* program (515 clauses) was over five hours. Investigations showed that the main time sinkhole in the CPL loading process was in fact the time taken to type check expressions: the type-checking algorithm was clearler not designed to handle expressions of this size.

This problem was avoided by constructing the primitives generating a class in a "ladder", with each primitive unioning the values generated by one normal-form clause with the value of an application of the primitive before. For example, using the second translation, the above normal-form program would yield the CPL program

```
primitive mk_C_1 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #b\Z) <- S.#D };
primitive mk_C_2 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #c\Z) <- S.#E } + (mk_C_1 @ S);
primitive mk_C_3 == \S =>
        { (\#a:Y, #b:Z)| (#a:\Y, #d\Z) <- S.#F } + (mk_C_1 @ S);
```

In this case the size of each primitive declaration was constant (and small), yielding the somewhat improved times shown in figure 23.

Conclusions and Further Work

Database transformations arise from a wide variety of applications, and many approaches to such transformations have been proposed. Some of the more significant work in this area was surveyed in Part I. However existing approaches tend to focus on specific applications, and deal with restrictive data-models, rather than looking at the problems of performing general transformations between databases expressed using complex and heterogeneous data-models. ([1] is a partial exception, but deals with a limited data-model which can not express recursive or arbitrarily deeply nested data-structures). Further most existing work concentrates on the effect of transformations on database *schemas* and largely ignores the corresponding effects of transformations on the underlying data: at best these aspects of a transformation are described informally, and often they are entirely ambiguous. This is a serious problem with such work, since, as we have seen, there may be many ways that a particular transformation at the schema level can be reflected on the underlying data. In order for some work on database transformations to be meaningful, it is essential that the effects of the transformations considered on the underlying data be properly defined.

Some existing work has also focused on the problems of obtaining measures of correctness for database transformations [23, 30]. This work attempts to classify transformations in terms of whether they preserve the information capacity of the underlying databases. However the applicability of such notions to models involving object-identities or recursive data-structures has been curtailed by the lack of a proper understanding of the information capacity of such models. In practice, the utility of such notions of correctness is limited by the interaction of transformations and constraints: it is possible for a database transformation to be information preserving or "correct" because of the presence of some implicit constraints on the databases involved, but not to be recognized as such because the constraints have not been expressed, and may not be expressible in the relevant data-models. Further, it is frequently the case that transformations are selective and only deal with a part of a source database. Established notions of correctness are clearly to restrictive to be used in such cases.

We have argued that, in order to reason about the correctness, or relative correctness, of database transformations, it is necessary to have a precise understanding of the information capacity of the underlying data-models, and to have a formalism in which we can express both transformations and general database constraints, and which allows us to reason about the interactions between the two.

20.2 Contributions

In part II of this thesis we gave a detailed analysis of the information capacity of data-models involving object identity. This is a problem that has arisen frequently in the literature, but had not, to the author's knowledge, previously been properly explored. We argued that the information capacity of a data-model coincides precisely with the observational properties of instances of that model, and consequently is dependent on the language or interface available for querying the model.

We showed that, given a reasonably expressive query language equipped with a predicate for testing whether two object identities were equal, database instances are indistinguishable if and only if they are *isomorphic*. However it is doubtful whether object-identities should be considered to be directly comparable in this manner, and seems that such comparisons force us to distinguish between structures which intuitively represent the same data. Next we showed that, if a query language does not support any means of directly comparing object identities, then instances are indistinguishable if and only if they are *bisimilar*. However we proved that, in general, to test for bisimilarity of individual values in a database requires time bounded in the size of the database, and consequently that bisimulation does not provide an adequately efficient means of comparing values in a database. Finally we showed that useful observational indistinguishability relations lying between these two extremes can be derived from systems of *keys*, and that, if we restricted our attention to *acyclic* systems of keys, then the resulting observational equivalence relations on values can be efficiently computed.

In Part III we presented a declarative language, *WOL*, for expressing database transformations and constraints. The language is based on a data-model supporting nested set, record and variant constructors, as well as object-identities, and in section 15 we showed how it could be extended to data-models dealing with alternative collection types such as bags and lists. Consequently the data-model is sufficiently general that data representable in other established data-models can be embedded in it in a natural way. *WOL* can be used to express general structural transformations on databases, and also a wide variety of database constraints, including those normally supported by established data-models. Further, since *WOL* has a precise, formally defined semantics, it allows us to formally reason about the interactions between transformations and constraints. The syntax of *WOL* is based on Horn clause logic, but allows us to express partial specifications of large and complex data-structures, and therefore makes the programming of transformations and constraints over such data-structures feasible.

We presented a methodology for implementing a significant class of *WOL* transformation programs by first manipulating them into a *normal form* and then translating the normal-form transformation program into some underlying database programming language such as CPL. The restrictions on *WOL* programs necessary to make such implementation possible were relatively general and easy to check. In section 17 we presented a series of optimizations necessary in order to make this normalization process practically feasible, and in section 18 we showed that, by introducing intermediate data-structures and using two-stage transformations, potential exponential growth in the size of normalized transformation programs on databases involving multiple variant types can be avoided. In section 19 we described a prototype implementation based on a restricted version of the *WOL* language, and described our experiences with trials of this system.

20.3 Further Work

Fortunately there remain many questions related to the work of this thesis to which I have not yet been able to find satisfactory answers, many ideas that I have not yet properly explored, and a great deal of work that was left undone. Consequently it is necessary to consider these things to be *further work*, and to hope that I or another interested researcher will be able to investigate them at some point in the future. Some of the more immediate or more significant items of further work are described bellow.

In section 11 we attempted to define an alternative model for instances, based on regular trees, which more closely captured the observable properties of instances (assuming no means of directly comparing object identities). Such models had been proposed previously, for example in [2] but the details had not been presented. As was apparent in section 11 modeling unordered collections, such as sets or bags, using regular trees is somewhat difficult and unnatural: it requires the definition of a complex notion of equivalence, such as bisimulation, on top of the standard representation of regular trees. Consequently it is not clear that the regular-tree based model is satisfactory, and the problem of how to construct a natural mathematical model for the observable properties of instances remains open. It is my belief that non-well-founded sets might be used to provide such a model, and that this requires further investigation.

Section 19 described trials carried out with a prototype implementation of a restricted version of the *WOL* language (*WOL++*). It is clear that, in order to further test and validate these ideas, a full implementation of the *WOL* language incorporating all the optimizations of section 17 is desirable. Further trials, both of the existing prototype and of a future full version of the language, are also necessary. In addition the syntax of the *WOL* language presented here is somewhat cumbersome, and it would valuable to establish syntactic enhancements to the language in order to make transformation programs easier to develop and read. Such enhancements for the restricted language, *WOL++*, proved to be of great utility.

The work on two-stage transformations presented in section 18 is a comparatively recent development, and leaves a number of unanswered questions. Most notable is the need to either prove or disprove the conjecture 18.1: that it is always possible to rewrite a transformation program as an equivalent two-stage normal-form transformation program with size linear in the size of the original program. If this conjecture turns out not to be true in general then it would be desirable to find useful side-conditions or restrictions sufficient to make it true. Further, it seems clear that it should be possible to automate the process of generating two-stage transformations, and the necessary intermediary data-structures, from a single stage transformation program. Further investigations and the development of suitable algorithms is necessary.

Finally there are issues related to the use of data-models supporting finite extents of object-identities that deserve examination. In section 9.2 we showed that it was possible to compute bisimulation relations on values of instance using only structural recursion over sets. This result was surprising because bisimulation is something that one would normally expect to require general recursion in order to compute, and indeed would not be decidable in a more general computational setting. In fact we were saved by the notion of finite extents, and the fact that all values occurring in a database instance arise from one of the known finite extents of that instance. A natural question to ask is therefore, what other intuitively recursive or undecidable functions can be computed on such database instances by using the knowledge of finite extents? Further what is a natural semantics for recursively defined functions over such database instances, and how can they be computed in an efficient manner. Initial investigations into these questions have proved very promising (and, in fact, were described in the proposal for this thesis). However further investigation is needed.

Bibliography

[1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[2] 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.

[3] 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.

[4] Serge Abiteboul and Jan Van den Bussche. Deep equality revisited. In *Proc. 4th International Conference on Deductive Object-Oriented Databases*. Springer-Verlag, 1995. To appear.

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

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

[7] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, 1987.

[8] C. Batini and M. Lenzerini. A methodology for data schema integration in the entity-relationship model. *IEEE Transactions on Software Engineering*, SE-10(6):650–663, November 1984.

[9] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[10] 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.

[11] 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.

[12] 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.

[13] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *LNCS 580: Advances in Database Technology — EDBT '92*, pages 152–167. Springer-Verlag, 1992.

[14] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

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

[16] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. Technical Report MS-CIS-93-94/L&C 74, University of Pennsylvania, Philadelphia, PA 19104, December 1994.

[17] U. Dayal and H. Hwang. View definition and generalisation for database integration in Multibase: A system for heterogeneous distributed databases. *IEEE Transactions on Software Engineering*, SE–10(6):628–644, November 1984.

[18] C. Eick. A methodology for the design and transformation of conceptual schemas. In *Proceedings of the 17th International Conference on Very Large Databases, Barcelona, Spain*, pages 25–34, September 1991.

[19] F. Eliassen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20(4):25–29, December 1991.

[20] Larry Gonick and Mark Wheelis. *The Cartoon Guide to Genetics.* Harper Collins, 1991.

[21] N. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[22] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3), July 1985.

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

[24] 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.

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

[26] W. Kent. The breakdown of the information model in multi-database systems. *SIGMOD Record*, 20(4):10–15, December 1991.

[27] 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.

[28] M. Kifer and G. Laussen. F-logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proceedings of ACM-SIGMOD 1989*, pages 46–57, June 1989.

[29] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[30] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th International VLDB Conference*, pages 120–133, August 1993.

[31] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19, 1994.

[32] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Transactions on Software Engineering*, SE-13(7):785–798, July 1987.

[33] S. Navathe, R. Elmasri, and J. Larson. Integrating user views in database design. *IEEE Computer*, 19(1):50–62, January 1986.

[34] P. Pearson, N. Matheson, N Flescher, and R. J. Robbins. The GDB human genome data base anno 1992. *Nucleic Acids Research*, 20:2201–2206, 1992.

[35] D. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. *SIGPLAN Notices (Proc. OOOPSLA '87)*, 22(12):111–117, October 1987.

[36] John F. Roddick. Schema evolution in database systems — An annotated bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.

[37] A. Rosenthal and D. Reiner. Theoretically sound transformations for practical database design. In S. T. March, editor, *Entity-Relationship Approach*, pages 115–131, 1988.

[38] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, December 1991.

[39] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of data models as canonical models for federated databases. *SIGMOD Record*, 20(4):44–48, December 1991.

[40] P. Shoval and S. Zohn. Binary-relationship integration methodology. *Data and Knowledge Engineering*, 6:225–249, 1991.

[41] Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object oriented database. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming*, pages 392–415. MIT Press, Cambridge, Massachusetts, 1987.

[42] S. Spaccapietra and C. Parent. Conflicts and correspondence assertions in interoperable dbs. *SIGMOD Record*, 20(4):49–54, December 1991.

[43] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.

[44] M. Tresch and M. Scholl. Schema transformation without database reorganization. *SIGMOD Record*, 22(1):21–27, March 1993.

[45] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems I*. Computer Science Press, Rockville, MD 20850, 1989.

[46] S. Widjojo, R. Hull, and D. S. Wile. A specificational approach to merging persistent object bases. In Al Dearle, Gail Shaw, and Stanley Zdonik, editors, *Implementing Persistent Object Bases*. Morgan Kaufmann, December 1990.

[47] S. Widjojo, D. S. Wile, and R. Hull. Worldbase: A new approach to sharing distributed information. Technical report, USC/Information Sciences Institute, February 1990.

[48] G. Wiederhold and X. Qian. Modeling asynchrony in distributed databases. *Proc. 1987 International Conference on Data Engineering*, pages 246–250, 1987.

[49] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

Index