Types With Extents:
On Transforming and Querying Self-Referential Data-Structures
(Thesis Proposal)

MS-CIS-95-21
Logic and Computation 92

Anthony Kosky

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department

Philadelphia, PA 19104-6389

1995

Types with Extents:
On Transforming and Querying Self-Referential Data-Structures

(Thesis Proposal)

Anthony Kosky

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389
Email: kosky@saul.cis.upenn.edu

28 February 1995

Disclaimer: The following is a working proposal paper for a PhD thesis and does not comprise a finished thesis. Notation and formalisms are subject to change without notice. Comments, corrections and personal abuse are all welcome, and should be sent to the author via email.
The author takes no responsibility for any damage or confusion incurred during the reading and contemplation of concepts contained in this paper. Cerebral protection should be worn at all times.
All data-structures in this paper are purely fictitious. Any resemblance to other data-structures, either living or dead, is purely coincidental.

Abstract

The central theme of this paper is to study the properties and expressive power of data-models which use type systems with *extents* in order to represent recursive or self-referential data-structures. A standard type system is extended with *classes* which represent the finite extents of values stored in a database. Such an extended type system expresses constraints about a database instance which go beyond those normally associated with the typing of data-values, and takes on an important part of the functionality of a database schema. Recursion in data-structures is then constrained to be defined via these finite extents, so that all values in a database have a finite representation.

The idea of extending a type system with such classes is not new. In particular [2] introduced a type system and data models equivalent to those used here. However such existing work focuses on the expressive power of systems which allow the dynamic creation of recursive values, while we are concerned more with the properties of querying and manipulating databases containing known static extents of data-values.

This paper consists of three parts. In part I we look at the problem of expressing transformations and constraints over a model based on *object identities.* A declarative language based on Horn-clause logic is introduced in which we can express a very general family of constraints and transformations. A *normal form* for transformations is defined and it is shown that transformation specifications expressed in the language which satisfy certain syntactic restrictions can be converted into equivalent transformation specification in normal form. The normal form transformations can then be converted into an appropriate DBPL for implementation.

In part II we present a more detailed study of data-models based around such an extended type system. A second data model, based of *regular trees* is introduced. It is shown that this second data-model is a finer model than the first object-identity based model, and that under certain assumptions about the operators available on object identities, the second data model is observably equivalent to the first. It is also shown that, under different assumptions about the operators on object identities, any two non-isomorphic instances in the second model are observationally distinguishable, and that other assumptions yielding useful observational properties between these two extremes are also possible.

In part III we study the evaluation of recursive function definitions over such data-structures. We show that, in general, such function definitions may have many possible solutions, and identify the desirable or intuitive solutions as those which can be computed *constructively.* We also show that, by making use of the known finite domains of such functions, we can compute these solutions in a manner which is guaranteed to terminate.

I have deferred conclusions and discussion of further work until part IV where I briefly state the additional work I believe needs to be done in these areas and directions for future research. In addition there are many omitted proofs and technical details that need to be written up, and perhaps a need for further examples.

Contents

1 Introduction

There is a natural analogy between the relationship between values and types in programming languages and that between instances and schemas in databases. Though the exact interpretation of these terms may vary, speaking broadly, a type describes a set of values with similar structure, while a schema describes a set of admissible instances for a database. However, while a database instance may be considered to be to be an unusually complex value, a schema may describe much more than the possible structure of these values. In particular a schema may describe a number of *constraints* which must be satisfied in order for an instance to be valid.

The kinds of constraints expressible in a schema depend upon the particular data-model and DBMS being considered. However perhaps the most important kind of constraint in a schema, common to all established database systems, is to describe a number of finite sets, representing the data stored in the database, and the relationships between these sets. In a relational model these sets would be the relations (sets of tuples) in a database, while in an object oriented model they might be the classes (sets of objects).

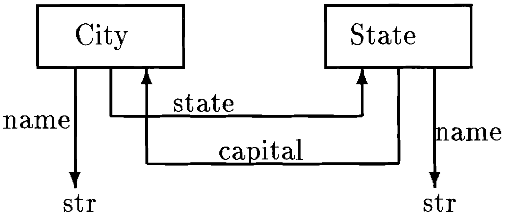


Figure 1: A simple database schema

The important distinguishing feature of databases, as opposed to more general programming environments, is that all values originate from one of these known sets, each of which is known to have a finite *extent*. It may be that new values are built from the values in these sets, but still they provide the anchor points from which the database is accessed. This is true whether the access to the database is in order to enforce constraints, or carry out transformations, as well as when querying or updating. We will borrow object-oriented terminology ([7]) and refer to these sets as *classes*, though the concepts are also common to other kinds of data-model.

For example figure 1 illustrates a schema for a database of Cities and States, in some indeterminate data-model. The schema shows that that database consists of two classes: a class of *Cities* and a class of *States*, and that each City has two components, a *name* and a state to which the city belongs, while each state also has two components, a *name* and a city which is it's capital. However the schema also implies some extra information: that the state of each *City* is in the *States* class of the database, and that the capital of each *State* is in the *City* class of the database. It is this sort of information that cannot be represented by a normal type system.

In this work we will describe a model (or a number of related models) in which the type system is extended to incorporate these classes. By doing so we hope to achieve a system which captures an important part of a database schema while maintaining the utility and simplicity of a conventional type system ([13, 31, 16]). The extended type system expresses some important information, namely the existence of sets with finite extents from which all values are sourced, which is not normally present in a type system. Our thesis is that, by basing database programming languages, query languages, transformation systems and so on on such an extended type system and making use of this additional information, we can gain significant expressive power over languages based on conventional type systems (such as CPL[12]).

An important example of this arises in recursive data-structures. Though recursive data-structures exist in some programming languages (such as streams in ML), programming with them requires use of lazy-evaluation and potentially non-terminating functions. In databases we know that all values have some finite static representation, and so recursive data-structures must make use of some finite number of place-holders (object identities or some other equivalent concept) and a cyclic construction. We know that such place-holders are taken from some finite extent, and can use this knowledge to compute functions on recursive data-structures in databases, where we would not expect to be able to define equivalent (terminating) functions in more general programming systems.

Another purpose of this work is to study the semantics of data-models based on object-identity. Object-identities occur naturally in object-oriented and semantic data-models ([7, 5, 21]), and have been shown to enhance the expressive power of purely value based models and languages ([2, 20]). They provide an abstract model for our intuitions about how complex structures, and particularly cyclic structures, are represented in a database. There are a number of other equivalent concepts, such as systems of equations over variables ([15]), or ML-style references ([25]), which basically amount to the same thing. Object-identities are considered to be internal to a database system, and not directly observable, thus leading to the question of when two database instances, based on object identities, are *observably equivalent*.

The concept of observable equivalence represents the idea that two database values or instances can not be distinguished using the some latent query language. Consequently observational equivalence is dependent on those predicates and operators that we consider to be available. For example, if we consider no comparison operators to be available on object identities, then we get a model equivalent to that based on *regular trees.* If we have an equality predicate on object identities, testing whether two object identities are the same, then we get a system where two values are equivalent if and only if one can be obtained from the other by applying an isomorphism of object identities.

An important class of observational equivalences, in between these two, can be obtained using systems of *keys.* By making such systems of keys primitive in a query language we can make a value-oriented language while achieving much of the efficiency of an object-identity oriented language. Further suitable systems of keys can be used to control the creation of object-identifiers ([20]), so that we can we can have a query language which supports the creation of object identifiers, but avoids the potential for non-terminating computations present in IQL([2]).

Part I

Database Transformations and Constraints

2 Transformations and Constraints of Databases with Object Identity

2.1 Schemas and Constraints

A *schema* represents the type structure of a database together with some constraints on the valid instances of the database. Precisely which type structures and which constraints can be expressed in a schema is dependent on the data-model being considered. The most fundamental constraints expressed by a schema are the declarations of a number of finite extents which comprise the database. However in general there are many more complicated and general constraints that one wishes to assert about a particular database.

For example let us consider the schema illustrated in figure 1. This diagram describes a database with two finite extents: one of *Cities* and the other of *States*. Every City in the database has a value associated with it, which is a pair consisting of its *name* and its *state*. Every State has a value associated with it which is a pair consisting of its *name* and a City which is its *capital*.

We will make use a data-model in which such extents are expressed as part of the type system. The model, which is equivalent to that used in [2], incorporates a nested relational type system, similar to that of [3], with the addition of *class types*. These class types are used to represent the *extents* present in the database. We will define a *schema* as consisting of a set of classes, representing the extents of a particular database, together with an association of types to classes, representing the types of the values associated with the objects in each class.

For example a schema for the database just described will have classes *City* and *State*, and will associate the type $(name : str, state : State)$ with the class *City* and the type $(name : str, capital : City)$ with *State*. This means that every City in the database has a record associated with it, with a *name* attribute of type string, and a *state* attribute which is a State in the database.

Most database systems will also support a number of other different kinds of constraints as primitive in the data-model. For example relational databases will often support keys and sometimes functional and inclusion dependencies ([36]), while semantic models might incorporate various kinds of cardinality constraints([21]). In general such collections of constraints form a rather ad hoc selection, included because of their utility in the particular examples of databases that the designer of the system had in mind, rather than because of any theoretical justification. With the increasing complexity of data-models, and of the tasks to which databases are being applied, it is apparent that there are many important and necessary constraints which fall outside these predetermined classes, and further, that it is difficult to anticipate the kinds of constraint that will arise.

For example, in our Cities and States database, we would want to impose a constraint that the capital City of a State is in the State of which it is the capital. This is an example of an *inclusion dependency* and we can express it as

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

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

$$X.population < Y.population \Longleftarrow X \in City, Y = X.state$$

These constraints are stated in a language called *WOL* which we will develop rigorously in section 5. This language is based on Horn clause logic. Horn clauses have been seen to provide a basis for simple but powerful programming languages (Prolog) ([37]) and database query languages (Datalog, ILOG)([36, 20]). They are easy to understand and reason about, and require only a small number of inference rules to build a complete proof system. Further they lend them selves naturally to a variety of paradigms for computation.

Our language is based around 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. Though it wouldn't be difficult in practice to extend *WOL* with operators to express such constraints, our experience indicates that the need for such constraints is uncommon, and their use is often somewhat contrived.

The language can express general database transformations, and in later sections we will see how it can be extended in order to define recursive functions over complex data-structures.

2.2 Database Transformations

Transformations of databases are required to meet numerous distinct needs: moving data from one or more distinct databases to an incompatible target database; incremental changes in database schemas, requiring data to be mapped from the old schema to the new one; and data entry, where the format and organization of the data as entered may be very different from that of the prescribed database schema. Existing work in this area includes systems which allow access to heterogeneous distributed database systems by means of queries against a unified view ([34, 40]), work on integrating database schemas ([33, 22, 26, 9]) which are useful primarily in schema design, and work on schema evolution ([32]). In addition there has been work done on analyzing whether transformations preserve or reflect information capacity of the underlying databases ([23, 24]). Also of relevance is the work on restructuring nested relational types presented in [1]. In this work a system for constructing rewrite rules using complex patterns was proposed, and a semantics of these rules in terms of the underlying data was given. The most significant difference between this work and our work on *WOL* is our ability to deal with transformations over recursive types and self-referential data-structures, and our attention to implementation techniques for such transformations. In addition the predicates of *WOL* are more powerful than the patterns of [1], for example allowing inequality predicates, and the ability to give partial descriptions of a transformation using *WOL* clauses gives an increase in expressive power.

Our work with database transformations arising in the Human Genome Project ([17]) indicates that there is a need to implement transformations as actual mappings of data from one database to an other: for example when populating a database with a newly evolved schema using the data already stored in an old version of the database, or when incorporating data from various archival databases into a local databases. In such cases the original data may be widely distributed and difficult to access, so that the unified view approach will be too costly, and it will be necessary to store copies of the data locally, and possibly restructure it, for efficient querying. Data stored in distinct heterogeneous databases may be represented in different and incompatible ways: a concept modeled as a top level relation in one database might be represented by a set valued attribute in another database, and some kind of flag in yet another database. Schema integration techniques, even those which do consider the semantics of the underlying data, require such incompatibilities to be resolved, and suitable restructurings to be applied, before they can be applied. Further many of the transformations that arise in practice are not information capacity preserving, but nevertheless require formal specification and analysis. None of the existing work mentioned fits all these needs.

Suppose that, in addition to our database of Cities and States, we have access to a second database containing information about European Cities and their Countries. A possible schema for such a database is shown in figure 2. We might want to transform the data stored

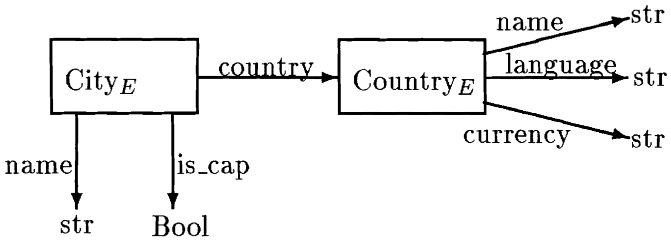


Figure 2: A schema for a database of European cities and countries

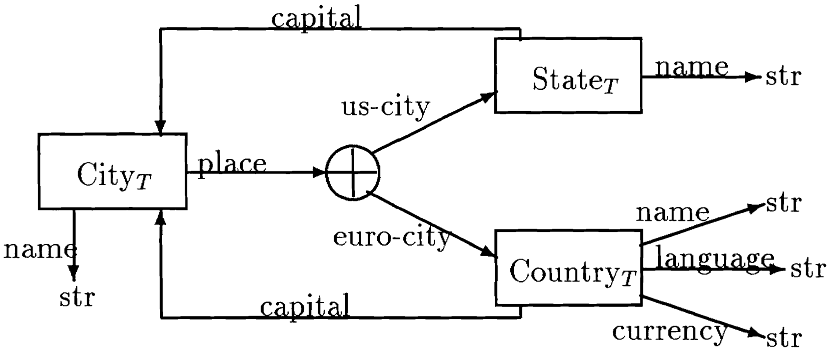


Figure 3: A schema for a combined database of European and American countries

in both of these databases into a common database, so that we can query a single class of Cities. Suppose our *target* database had the schema shown in figure 3.

We can use the language *WOL* to describe these transformations. If we rename the classes of our first schema to $City_A$ and $State_A$, then we can write clauses to express the population of the classes $City_T$, $State_T$ and $Country_T$ of the new database as:

$$\begin{aligned}
Y \in City_T, Y.name = X.name &\Longleftarrow X \in City_A \\
Y \in City_T, Y.name = X.name &\Longleftarrow X \in City_E \\
Z \in State_T, Z.name = X.name &\Longleftarrow X \in State_A \\
&Z \in Country_T, Z.name = X.name, \\
&Z.language = X.language, Z.currency = X.currency
\end{aligned}$$

$$\Longleftarrow \quad X \in \mathit{Country}_A$$

These clauses state that, for every object of class City_A in the first database there is a corresponding object with the same *name* in the class City_T in the target database, and so on. However we also need clauses to describe how the *place* and *capital* attributes in the target database are populated. *place* is not too difficult:

$$\begin{array}{rcl} X.\mathit{place} = \mathit{ins}_{\text{us-city}}(Y) & \Longleftarrow & X \in \mathit{City}_T,\ Y \in \mathit{State}_T,\ U \in \mathit{City}_A,\ V \in \mathit{State}_A \\ & & X.\mathit{name} = U.\mathit{name},\ Y.\mathit{name} = V.\mathit{name},\ V = U.\mathit{state} \\ X.\mathit{place} = \mathit{ins}_{\text{euro-city}}(Y) & \Longleftarrow & X \in \mathit{City}_T,\ Y \in \mathit{Country}_T,\ U \in \mathit{City}_E,\ V \in \mathit{Country}_E \\ & & X.\mathit{name} = U.\mathit{name},\ Y.\mathit{name} = V.\mathit{name},\ V = U.\mathit{country} \end{array}$$

(Here *ins* is a constructor for variant types).

However populating the *capital* attribute of the class *Country* is more difficult. Note that there is a basic incompatibility between the representation of capital cities in our City and Country database, which are marked by a Boolean flag, *is_cap*, and the *capital* attribute of the class $\mathit{Country}_T$ in our target database. We can express this correspondence in *WOL* by:

$$\begin{array}{rcl} X.\mathit{capital} = Y & \Longleftarrow & X \in \mathit{Country}_T,\ Y \in \mathit{City}_T,\ Y.\mathit{place} = \mathit{ins}_{\text{euro\_city}}(X) \\ & & U \in \mathit{City}_E,\ U.\mathit{name} = Y.\mathit{name},\ U.\mathit{is\_cap} = \mathit{True} \end{array}$$

This states that for any Country X and City Y in our target database, if X is the country of Y and there is a city U in our database of European cities and countries (U is of class City_E), such that U corresponds to Y in the target database and U has its *is_cap* attribute set to **T**, then Y is the capital city of X. It is this sort of structural manipulation between alternative representations of data that the *WOL* language is particularly well suited to, and is where existing schema manipulation and integration techniques fall short.

In order for clauses such as these to successfully specify a transformation, we require certain additional information about the target database. In particular we need to know that the objects of classes City_T, State_T and $\mathit{Country}_T$ are all uniquely determined by their *name* attributes. This is an example of some key dependencies which are easy to express in *WOL:*

$$X = Y \Longleftarrow X \in \mathit{City}_T,\ Y \in \mathit{City}_T,\ X.\mathit{name} = Y.\mathit{name}$$

This illustrates one of the main principles behind the design of the language *WOL*: *database constraints and transformations are fundamentally linked, and should be expressed and reasoned about in a common formalism.* Firstly constraints on a target database play an important part in determining transformations: keys or functional dependencies will uniquely identify values being inserted into a target database, while inclusion dependencies will cause additional parts of the database to be populated. But transformations also play a role in asserting constraints, both on the source and target databases.

In the example above, each Country in the the target database must have a capital city. However there is no such constraint implied on our source Cities and Countries database. The transformation specification then implies a constraint

$$Y \in \mathit{City}_E,\ Y.\mathit{country} = X,\ Y.\mathit{is\_cap} = \mathit{True} \Longleftarrow X \in \mathit{Country}_E$$

This says that for every X of class $\mathit{Country}_E$ there is a Y of class City_E such that Y is a capital city and Y has country X. Having derived such a constraint from our transformation specification, we can then test whether our source database satisfies this constraint, and consequently ensure that the transformation will be well defined, prior to actually carrying out the transformation.

2.3 Implementing transformations

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 6 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.

The work described in section 6 is an extension of work done by myself in collaboration with people working at the Philadelphia Genome Center for Chromosome 22, at the University of Pennsylvania, on a language called *TSL* and an associated transformation system ([17]). TSL could be regarded as a restriction of the language *WOL* to a standard nested relational data-model. It also makes use of Skolem functions, as in [20], in order to simulate object identities.

TSL was designed in order to address various database transformation problems encountered in a Human Genome Project center. Some of these applications are described and motivated in section 3. The normal forms for TSL were defined for the case where the target database is flat relation and the source databases have arbitrary nested relational structure, and the normalization algorithm was implemented for this case. The normal form for *WOL* described in section 6 does not have such restrictions.

The language TSL has two alternative syntaxes: a *basic* syntax, similar to the syntax for *WOL* defined in section 5, and an *extended syntax.* The basic syntax is simpler and easier to manipulate and reason about, and was therefore used in the formal development and internal implementation of TSL, while the extended syntax is less cumbersome and more convenient to program with. For example the transformations from the Cities and Counties database to the target database describe above would look something like the following in extended TSL syntax:

$$\begin{aligned}
&(name = N, place = ins_{\mathrm{euro\_city}}(Y)) \in City_T \\
&\quad\Longleftarrow\quad (name = N, country = (name = C)) \in City_E,\ Y(name = C) \in Country_E;
\end{aligned}$$

$$\begin{aligned}
&(name = C, capital = X, language = L, currency = M) \in Country_T \\
&\quad\Longleftarrow\quad Y(name = C, language = L, currency = M) \in Country_E, \\
&\quad\quad (name = N, is\_cap) = True, country = Y) \in City_E, \\
&\quad\quad X(name = N) \in City_T;
\end{aligned}$$

It seems that some kind of extended syntax is also desirable for *WOL*, in order to make programming transformations more convenient and efficient. However I believe more experience is needed in programming transformations and constraints between complex data-structures in order to decide on the details of such an extended syntax.

The implementation of TSL consists of a parser (for the extended syntax), together with type and well-formedness checkers, a normalizing program, which tests a TSL program for recursion and converts it to normal form if it is non-recursive, and a translator which then translates the normal form TSL program into CPL. CPL is a query language developed at the University of Pennsylvania which has interfaces to SYBASE and a variety of other biological data-sources ([41]). Developing translators from normal-form TSL programs into other DBPLs is a straightforward task.

3 Application of Transformations in Human Genome Project Databases

The work on database transformations and constraints outlined in section 2, and described in detail in the following sections, was in part inspired by a series of collaborations with the Philadelphia Genome Center for Chromosome 22, at the University of Pennsylvania, and is intended to address the database transformation problems encountered in a Human Genome Project (HGP) center. In this section we will look at some of the transformation problems involved in maintaining Chr22DB: the laboratory notebook database for the Philadelphia Genome Center for Chromosome 22. In order to understand these problems, it will first be necessary to give some background on the Human Genome Project and the relevant biology.

The goal of the Human Genome Project is to sequence the 24 distinct chromosomes comprising the human genome. Each chromosome is composed of a long, double-stranded molecule of DNA (deoxyribonucleic acid) made up of complementary pairs of four different nucleotides or *bases* (A, G, C, T), arranged like beads on a string. *Sequencing* DNA means discovering the exact sequence of A's, C's, T's, and G's on the string. Although there are techniques for directly sequencing short DNA strings (approximately 400 bases), current methods are not practical for sequencing the entire genome (3 billion bases) at one time. Consequently the HGP has set mapping the chromosomes as a less ambitious intermediate goal. *Mapping* involves the ordering of identifiable DNA fragments as markers along the chromosome, and anchoring markers at known positions to serve as landmarks.

One of the major problems faced in HGP databases is rapid schema evolution and the resulting need to modify existing applications. New and better experimental techniques are constantly being developed and the experimental data being modeled is constantly changing, forcing evolution of the laboratory notebook database schema and related applications. This process must occur extremely rapidly, since investigators consult the database to plan and guide ongoing experimentation. Furthermore data integrity is crucial. However, the data is very complex, hierarchically organized, and contains an unusually large number of links among tables (inclusion dependencies). This gives rise to a number of complex, non-standard constraints that need to be specified and enforced in order for the data to be correct.

Another major problem is that access to multiple, heterogeneous remote databases and software packages is frequently needed to augment the contents of the laboratory notebook databases and to answer queries posed by researchers. These include archival databases, such as the nucleic acid sequence database, Genbank, the protein sequence data base, PIR [8], the biomedical bibliographic data base, Medline, and the human genome map data base, GDB [28]; a growing number of laboratory notebook databases; as well as software systems such as BLAST [6], FASTA [29], and Staden, which perform complex data analysis involving such computational problems as pattern-matching, search and string comparison. These databases include flat-relational databases (Sybase), object-oriented databases (Object Store, GemStone), complex-relational databases (ASN.1), and personal-computer-based databases.

This heterogeneity in schemas and models within the HGP is likely to persist. As data complexity increases, different databases may capture only partial, and perhaps significantly different views of the data as a whole; as analysis tasks increase in complexity beyond simple queries, it is often necessary to organize the data to optimize a specific application to achieve acceptable system performance. Thus, we find numerous independent structurings of the same or similar information. The GenBank family is a case in point: there is the "standard" flat-file version with numerous trivial syntactic variants, a relational version developed at the Los Alamos National Laboratory [14], the ASN.1 version developed at NCBI, a relational version developed from the ASN.1 version by the Philadelphia Center for Chromosome 22 [19], and at least one knowledge base version, also developed within our group [27], which transforms the data from a sequence entry view to a biological concept view. Each of these has its own advantages and disadvantages that include issues of representation, query language expressiveness, and portability, among others.

A recent report of a Department of Energy Informatics "summit" [18] underscores these problems and indicates that they are pervasive to HGP databases. The report listed a number of simple queries that were impossible to answer with the current data sources, because the sources are distributed among various databases, programs and structured files, and there is no effective technique for combining them.

An important part of this is the problem of *transforming data* into some form that is understandable by users, a query language, or an applications program. The problem of schema or data evolution calls for flexible tools for rapidly re-mapping databases. We need a principled approach to data transformations: transformations between schemas in a single data model (as with schema evolution), between different data models (as with data entry screens, and as in the Genbank family of databases), or across multiple data models (as in the integration of data from multiple sources). We believe that the formalism and tools outlined in section 2 and described in detail in the following sections provide a solution to these problems.

3.1 A Databaser's View of the Biological Background

The data and schemas in the archival and laboratory notebook databases for the HGP are highly complex and difficult to understand, especially for those who know little to nothing about molecular biology. We will therefore start off by explaining a bit about what is being modeled and what some of the terms used mean.

The HGP's intermediate goal is *mapping*: ordering markers (fragments of DNA) along the human chromosome and locating them at known positions. A variety of techniques are used to anchor markers to specific locations on the chromosome. For the sake of simplicity, we will consider only one: *physical mapping* using cloned probes and Sequence Tag Sites (STS's).

The chromosome of interest is cut randomly into overlapping pieces of experimentally manipulable size (50,000-1 million bases). These pieces are then reassembled into a linear ordering representing their order in the original DNA string. To discover the relative ordering of fragments, it is crucial to be able to ascertain when the sequence of two pieces of DNA overlaps, that is, when the pieces come from neighboring sites in the original string. Sequence overlap between two pieces of DNA can be detected by showing that their sequences contain the sequence of a third, much shorter fragment, called a *probe*. The linear ordering on the pieces yields a linear ordering on the probes whose sequence is contained in them, and vice versa. The probes then become the desired map landmarks and may be used to sequence areas of special interest, such as regions thought to be related to inheritable disease.

Physical mapping and its relationship to DNA sequence is illustrated in Figure 4[1]. At the top of this figure, a chromosome is depicted with the banding patterns visible under a microscope, which themselves function as landmarks at the coarsest level of granularity. Vertical lines denote markers (probes). Horizontal lines denote larger, overlapping DNA fragments whose sequence contains marker sequence. Below, the sequence of a tiny substring of DNA is shown.

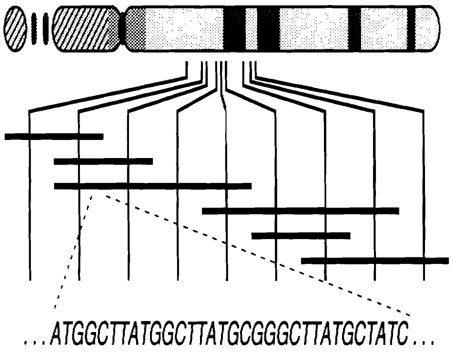


Figure 4: Physical Mapping of a Chromosome

Two types of probes used in physical mapping and represented in Chr22DB are: 1) Cloned probes and 2) Sequence Tag Sites (STS's). Cloned probes are actual physical reagents stored

[1] This figure was made by David Searls

in freezers, and STS's are information stored in a database. In what follows, we briefly describe some of the information maintained about probes by Chr22DB.

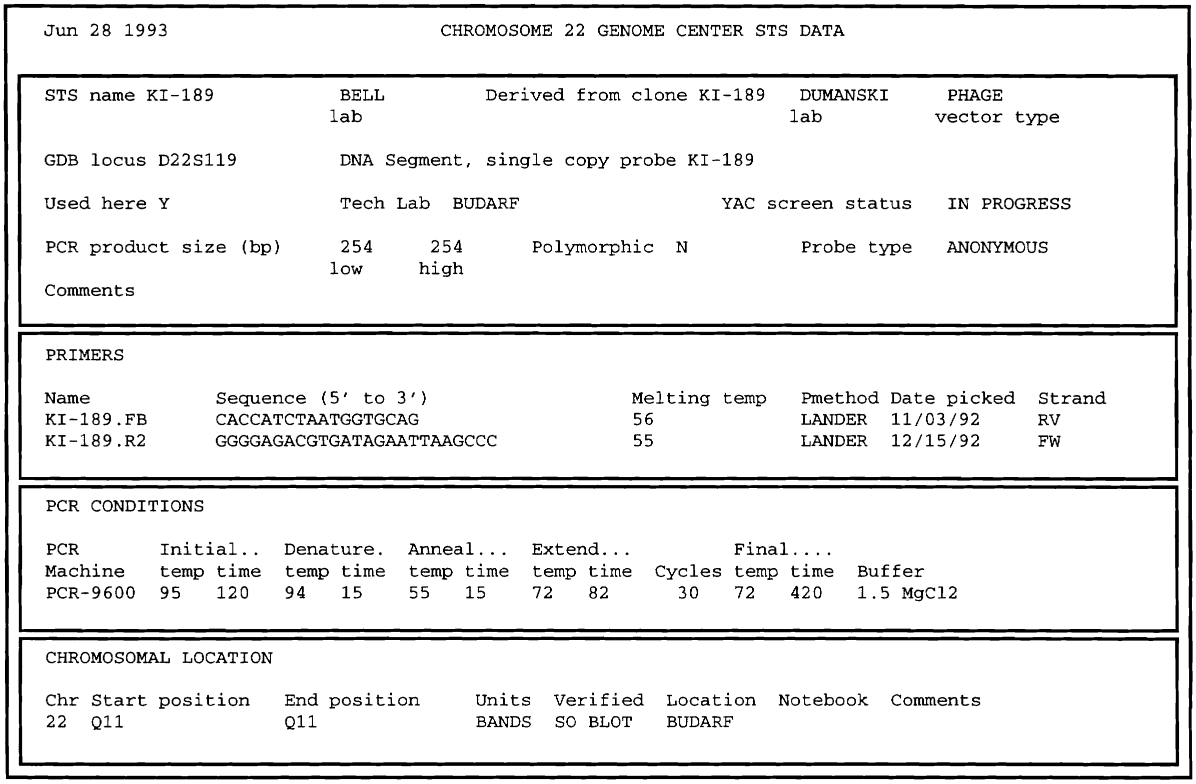
Cloned Probes. In cloning, a fragment or *interval* of human DNA is inserted into carrier or *vector* DNA in bacterial or yeast cells. When the host cells are cultured, many exact replicas of the human DNA are produced, to be used in future experiments.

Sequence Tag Sites (STS's). An STS is an interval of DNA defined by a *primer pair*: a pair of sequenced nucleic acid intervals used as primers to start a chemical reaction called *amplification by the polymerase chain reaction* (PCR amplification). The entire reaction comprises several stages, each proceeding at a different temperature. An amplification reaction will not occur unless the primer sequences are found, properly spaced, within the test sequence; therefore, a successful reaction demonstrates sequence containment. Important data items about an STS are: its name, including the laboratory which named it; the name, sequence, and melting temperature of each of the primers; the expected size range of the amplified product; the temperature and time required for each stage of the process (PCR conditions); a cross-reference to GDB; the name of the cloned probe from which the primers were derived; and the chromosomal location of the site.

3.2 A Sample Database Transformation

The data in Chr22DB comes from a variety of sources: archival databases such as GDB, preexisting spreadsheet databases, object-oriented laboratory notebook databases from other centers, as well as directly from experiments being carried out at the center. Importing data from these sources involves data transformations, and has to date been done largely by hand. Though not particularly glamorous, data entry is a special case of a data transformation, and provides a good illustration of some of the complexity of structural transformations. Rewriting the data entry applications is enormously time consuming since application generator tools can not handle the complexity of the data involved, and the modification has to be done by hand.

In data entry applications, the data is captured on a screen form that provides a specialized view of the underlying database. The view and the database may differ widely in structure, and the application must map between these two schemas. An example of a form used to enter STS lab notebook data is shown in Figure 5. It consists of a complex relation (`STS`) with three sub-relations (`Primers`, `PCR_conditions`, `Location`). Since each screen enters a single STS, there will be one row in the `STS` relation, two rows in the `Primers` relation denoting the primer pair, and multiple rows in the `PCR_conditions` and `Location` relations.



```
Jun 28 1993                              CHROMOSOME 22 GENOME CENTER STS DATA

STS name KI-189           BELL        Derived from clone KI-189  DUMANSKI     PHAGE
                         lab                                    lab         vector type

GDB locus D22S119         DNA Segment, single copy probe KI-189

Used here Y               Tech Lab  BUDARF                YAC screen status   IN PROGRESS

PCR product size (bp)     254     254     Polymorphic  N         Probe type   ANONYMOUS
                         low     high
Comments

PRIMERS

Name           Sequence (5' to 3')                   Melting temp   Pmethod Date picked  Strand
KI-189.FB      CACCATCTAATGGTGCAG                    56             LANDER  11/03/92     RV
KI-189.R2      GGGGAGACGTGATAGAATTAAGCCC             55             LANDER  12/15/92     FW

PCR CONDITIONS

PCR       Initial..  Denature.  Anneal...  Extend...         Final....
Machine   temp time  temp time  temp time  temp time  Cycles temp time  Buffer
PCR-9600  95   120   94   15    55   15    72   82      30   72   420   1.5 MgCl2

CHROMOSOMAL LOCATION

Chr Start position   End position     Units  Verified  Location  Notebook  Comments
22  Q11              Q11              BANDS  SO BLOT   BUDARF
```

Figure 5: STS Data Entry Screen

Data entered at the data-entry screen must be transformed to the underlying (relational) Chr22DB database. A conceptual (EER) schema of the relevant portion of Chr22DB, drawn using ERDRAW [35], is shown in Figure 6; this is merely introduced to convey the linkages between relations rather than to give a precise semantics of the schema.

In this database transformation a complex relation with nested subrelations is flattened into a standard relational schema with value-based pointers linking related tables. The atomic attributes of the top-level screen relation are distributed over 6 relational tables in the target schema: `names`, `lab`, `material`, `interval`, `na_interval`, and `STS`. The `Primers` subrelation is decomposed into 5 target tables: `na_interval`, `interval`, `material`, `primer`, `sequence`. The two name fields in the entry screen (`STS_name` and `GDB_locus`) are mapped to two separate rows in the target `names` table, which are linked by the internal identifier of the object being inserted. We will view each of these tables as a set of objects in our model, the value of each object being a simple tuple.

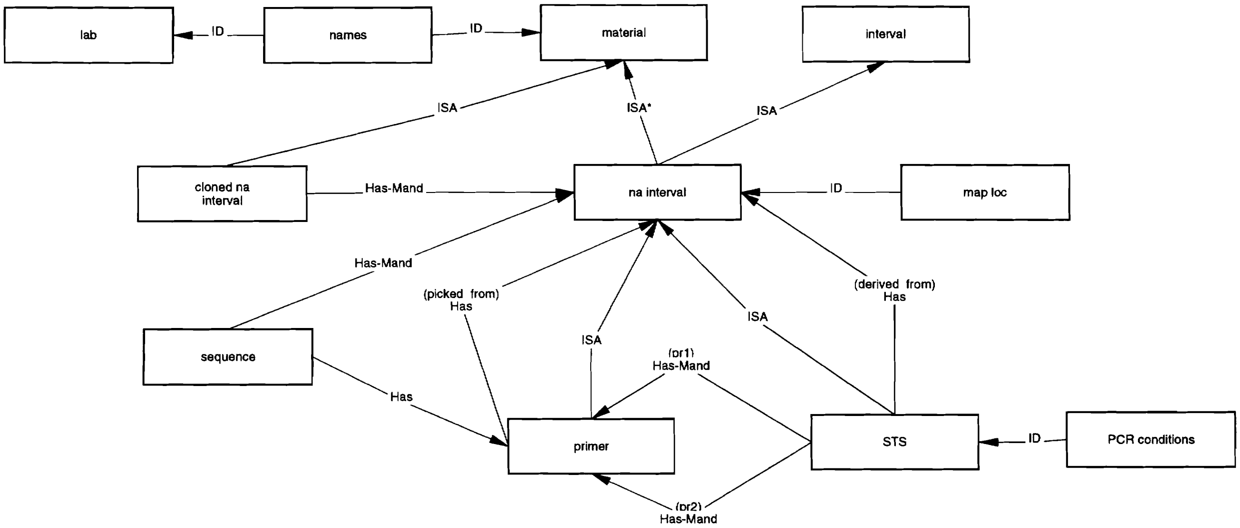


Figure 6: EER Schema of Target Database for STS's.

In order to accomplish the data transformations, appropriate insert statements must be generated. The normalized target schema relies on internal system-generated identifiers to accomplish the links among related tables. These correspond to the object-identifiers of our model.

Database integrity constraints

To maintain data integrity, the transformed data must conform to the integrity constraints of the target database. Preeminent are key and inclusion dependency constraints, but more complex constraints may also hold. For example, each material must have at least one GDB name (i.e., $X \in \texttt{names}$, $X.\texttt{lab\_code} =$ "GDB") and at least one non-GDB name (i.e, $X \in \texttt{names}$, $X.\texttt{lab\_code} \neq$ "GDB").

There are also simple integrity constraints requiring the inclusion of objects identities is certain tables. For example for every `primer id` in the `STS` table there is a corresponding entry in the `primer` table. We could express such a constraint as a clause

$$P \in \texttt{primer} \Longleftarrow S \in \texttt{STS},\ X.\texttt{pr1\_primer\_id} = P$$

However this is not necessary since such constraints are implicit in the type system of our model.

There are many constraints which do need to be expressed explicitly however, including some which go beyond the traditional functional and existence dependencies of the relational model. For example that each material has exactly one GDB name:

$$\begin{aligned} X = Y \quad \Longleftarrow \quad & X \in \texttt{names},\, X.\texttt{material\_id} = M,\, X.\texttt{lab\_code} = \text{``GDB''} \\ & Y \in \texttt{names},\, Y.\texttt{material\_id} = M,\, Y.\texttt{lab\_code} = \text{``GDB''} \end{aligned}$$

And that a public name cannot be a GDB name:

$$\mathsf{False} \Longleftarrow X \in \texttt{names},\, X.\texttt{public\_name} = \text{``Yes''}, X.\texttt{lab\_code} = \text{``GDB''}$$

Transformation clauses

The transformation is specified by the constraints on the database together with a number of transformation clauses which describe the relation between the data source (the STS input screen) and various tables in the target database, Chr22DB.

The following is a transformation clause generating part of the STS relation of the schema shown in Figure 6 from the data entry screen shown in Figure 5:

$$\begin{aligned} & S = Mk^{\mathsf{STS}}(PI1, PI2), S.\texttt{pr1\_primer\_id} = PI1, \\ & S.\texttt{pr2\_primer\_id} = PI2, \\ & S.\texttt{PCR\_prod\_size\_lo} = SL, \\ & S.\texttt{PCR\_prod\_size\_hi} = SH \\ & \quad \Longleftarrow \; X \in \texttt{STS\_screen}, \\ & \qquad P1 \in X.\texttt{primers},\, P1.\texttt{pname} = PN1 \\ & \qquad P2 \in X.\texttt{primers},\, P2.\texttt{pname} = PN2 \\ & \qquad X.\texttt{PCR\_prod\_size\_lo} = SL, \\ & \qquad X.\texttt{PCR\_prod\_size\_hi} = SH, \\ & \qquad PI1 \in \texttt{Primer},\, PI1.\texttt{pname} = PN1, \\ & \qquad PI2 \in \texttt{Primer},\, PI2.\texttt{pname} = PN2, \\ & \qquad PI1 < PI2 \end{aligned}$$

There are several points about this clause that deserve comment. Firstly notice that the operator Mk^{STS} is used to generate object identities for the STS table from pairs of identities of the Primer table. Also although the STS_screen relation has only one attribute primers, it occurs in two separate atoms in the description of a tuple in the STS_screen relation. This is because the attribute is *set valued* and each of the two atoms asserts the presence of a different tuple in the primers sub-relation.

The body of this clause makes use of the target database relation primer in order to look up the primer_id's. The tuples for this relation are in turn generated by another clause:

$$\begin{array}{l}
PI = Mk^{\texttt{Primer}}(PN), \\
PI.\texttt{melting\_temp} = MT,\ PI.\texttt{pick\_method} = PM, \\
PI.\texttt{date\_picked} = DP, PI.\texttt{strand} = ST \\
\qquad \Longleftarrow\ X \in \texttt{STS\_screen},\ P \in X.\texttt{primers}, \\
\qquad\qquad P.\texttt{pname} = PN,\ P.\texttt{melting\_temp} = MT, \\
\qquad\qquad P.\texttt{pmethod} = PM,\ P.\texttt{date\_picked} = DP, \\
\qquad\qquad P.\texttt{strand} = ST
\end{array}$$

In order to implement this transformation it is necessary to unfold clauses like this, in order to get equivalent clauses in *normal form*: that is clauses that refer only to source relations in their bodies and only to target relations in their heads. Clauses of this form can be processed in one-pass without referring to the target database.

The normal-form clauses are built by combining and unfolding clauses of a transformation, in order to form clauses which provide a complete description of a tuple in the target database in terms of the elements of the source

If it is possible to build only a partial description of a tuple for some relation, then it follows that the transformation program is not complete.

For example a normal-form clause for the STS table in the transformation from the STS data-entry screen (Figure 5) to Ch22DB (Figure 6) formed from the clauses above would be:

$$\begin{array}{l}
S = Mk^{\texttt{STS}}(PI1, PI2), \\
S.\texttt{pr1\_primer\_id} = PI1, \\
S.\texttt{pr2\_primer\_id} = PI2, \\
S.\texttt{PCR\_prod\_size\_lo} = SL, \\
S.\texttt{PCR\_prod\_size\_hi} = SH \\
\qquad \Longleftarrow\ X \in \texttt{STS\_screen}, \\
\qquad\qquad P1 \in X.\texttt{primers},\ PN1 = P1.\texttt{pname}, \\
\qquad\qquad P2 \in X.\texttt{primers},\ PN2 = P2.\texttt{pname}, \\
\qquad\qquad X.\texttt{PCR\_prod\_size\_lo} = SL, \\
\qquad\qquad X.\texttt{PCR\_prod\_size\_hi} = SH, \\
\qquad\qquad PI1 = Mk^{\texttt{Primer}}(PN1),\ PI2 = Mk^{\texttt{Primer}}(PN2), \\
\qquad\qquad PI1 < PI2
\end{array}$$

Notice that this clause gives a complete description of a tuple in the STS relation, and does not call on any of the target relations in the body of the clause. In particular the calls to the **primer** relation which were in the body of the previous clause have been replaced by applications of the operator $Mk^{\texttt{Primer}}$ in order to create primer identities.

4 A Data Model with Extents and Object Identity

In this section we will define the data-model on which the language *WOL* is based. The model is basically equivalent to that described in [2] and its type system incorporates *class types* which are used to represent the finite *extents* present in a database.

In part II we will examine this data-model in more detail, in particular studying the observational properties of the model, and will show that under certain assumptions about the predicates available on object identities, it is observation-ally equivalent to a much coarser model based on *regular trees.*

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

4.1 Types and Schemas

Definition 4.1: Assume a finite set of *classes* $\mathcal{C}$, ranged over by $C, C', \ldots$, and a countable set of *attribute labels*, $\mathcal{A}$, ranged over by $a, a', \ldots$. The set $\mathit{Types}^{\mathcal{C}}$ of **types** over $\mathcal{C}$ is then given by the following abstract syntax:

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

where $\underline{b}$ is some *base type.* The notation $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ represents a *record type* with attributes $a_1, \ldots, a_k$ of types $\tau_1, \ldots, t_k$ respectively, while $\langle\!|a_1, \tau_1, \ldots, a_k : \tau_k|\!\rangle$ represents a *variant type.*

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

$$\mathcal{C}_A \equiv \{City_A, State_A\}$$

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

$$\begin{aligned} \mathcal{S}_A(City_A) &\equiv (name : str, state : State_A) \\ \mathcal{S}_A(State_A) &\equiv (name : str, capital : City_A) \end{aligned}$$

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

For the combined schema of figure 3 the classes would be

$$\mathcal{C}_T \equiv \{City_T, State_T, Country_T\}$$

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

$$\begin{aligned} \mathcal{S}_T(City_T) &\equiv (name : str, place : \langle\!| us\_city : State_T, euro\_city : Country_T |\!\rangle) \\ \mathcal{S}_T(State_T) &\equiv (name : str, capital : City_T) \\ \mathcal{S}_T(Country_T) &\equiv (name : str, language : str, currency : str, capital : City_T) \end{aligned}$$

■

We can view schemas as directed graphs, with classes and other type constructors as their nodes. Note that, given this view of a schema, 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 4.2, any recursive data-structures in an instance must have a finite representation via the object-identifiers of these classes.

4.2 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.

In order to represent cyclic or recursive data-structures in a finite manner it is necessary to use some mechanism such as object identities, pointers, variables and recursive equations, and so on. In fact all of these are simply different representations of the same basic concept, and the choice of which style of representation to use is simply a matter of personal preference. My decision to use object identities is simply because I find it the most intuitive of these alternatives, and because it fits in nicely with actual implementations of object-based databases. However the results of this section, and of later sections, should be easily adaptable to other such alternative mechanisms.

For each base type $\underline{b}$, assume a 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[2] 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

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

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

Definition 4.2: For each type τ define $[\![\tau]\!]\sigma^{\mathcal{C}}$ by

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

■

Definition 4.3: A database **instance** of schema S consists of a family of *object sets*, $\sigma^{\mathcal{C}}$, and for each $C \in \mathcal{C}$ a mapping

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

■

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

Example 4.2: We will describe an instance of the schema introduced in example 4.1.

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

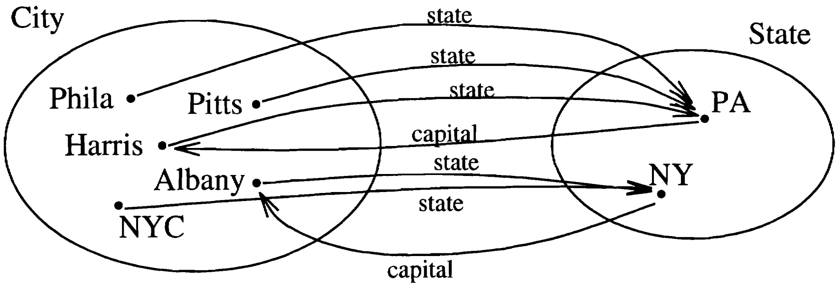


Figure 7: A database instance

Our object identities are:

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

and the mappings are

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

and

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

This defines the instance illustrated in figure 7. ■

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 database programming language 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, $f^C : \sigma^C \rightarrow \sigma'^C$, $C \in \mathcal{C}$ then we can extend $f^{\mathcal{C}}$ to mappings $f^\tau : [\![\tau]\!]\mathcal{I} \rightarrow [\![\tau]\!]\mathcal{I}'$ as follows:

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

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 4.4: 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 $\mathcal{I}'$. ■

4.3 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. In part II we will analyze the observational properties of this model in more detail, and show how various assumptions about the predicates available on object identities effect their observational properties. However, in order to do transformations over databases with object identity, we need some way of uniquely identifying an object. 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 4.5: 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 $\mathcal{K}_{\mathbf{I}}^{\mathcal{C}}$ from instances, $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$, of $\mathcal{S}$ to families of functions

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

for each $C \in \mathcal{C}$. ■

Example 4.3: Consider the first schema described in example 4.1. 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 $\mathcal{K}_{\mathcal{I}}^{\mathcal{C}}$ are given by

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

■

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 $o \in \sigma^C$,

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

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

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

Definition 4.6: 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 4.3 has nodes *City* and *State*, and a single edge $(City, State)$.

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

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

Keyed Schema

Definition 4.7: A **keyed schema** is a pair consisting of a schema S and a key specification $\mathcal{K}^{C}$ on 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 $\mathcal{I}$ of $\mathcal{S}$ is said to **satisfy** the keyed schema $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$ if and only if for any class $C \in \mathcal{C}$ and any $o_1, o_2 \in \sigma^{C}$, if $\mathcal{K}^{C}_{\mathcal{I}}(o_1) = \mathcal{K}^{C}_{\mathcal{I}}(o_2)$ then $o_1 = o_2$. ■

For example the instance described in example 4.2 satisfies the first schema of example 4.1 and the key specification from example 4.3.

5 A Logic for Constraints and Transformations

In this section we will give a rigorous definition of the language *WOL* introduced informally in section 2. In section 5.1 we will define the syntax for *WOL*, and in section 5.2 we will give a denotational semantics for *WOL* based on the model of section 4.

5.1 Syntax

We will assume a simply keyed schema, $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$, with classes $\mathcal{C}$, and will define our language, $WOL^{\mathcal{SK}}$, relative to this schema.

As before we will assume a countable set of constant symbols ranged over by $c^{\underline{b}}$ for each base type $\underline{b}$, 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 5.1: The set of **terms** for S, $Terms^{\mathcal{S}}$, ranged over by $P, Q, \ldots$, is given by the abstract syntax:

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

■

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 will occur 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:

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

Definition 5.2: Given a type context Γ, the relation $\Gamma \vdash : \subseteq Terms^S \times Types^C$ 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 4.1, if $\Gamma(X) \equiv City$, then $\Gamma \vdash !X : (name : str, state : State)$ and $\Gamma \vdash \pi_{name}(!X) : str$.

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 5.3: The set of **atoms** for S, $Atoms^S$, ranged over by $\phi, \psi, \ldots$ is given by the abstract syntax:

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

■

The atoms $P\dot{=}Q$, $P\dot{\neq}Q$, $P\dot{\in}Q$ 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 5.4: An atom ϕ is said to be **well-typed** by a type context Γ iff

1. $\phi \equiv P\dot{=}Q$ 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 occuring 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.

Range restriction

The concept of *range-restriction* is used to ensure that every term in collection of atoms is bound to some value occuring 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.

Definition 5.5: Suppose Φ is a set of atoms, and P is an *occurrence* of a term in Φ. Then P 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 Q is a range restricted occurrence of a term in Φ;
4. P occurs in a term $Q \equiv ins_a P$, where Q is a range-restricted occurrence of a term in Φ;
5. $P \equiv !Q$ where Q is a range-restricted occurrence of a term in Φ;

6. Φ contains an atom $P \doteq Q$ or $Q \doteq P$ or $P \dot{\in} Q$, where Q is a range-restricted occurrence of a term in Φ;

7. $P \equiv X$, a variable, and there is a range-restricted occurrence of X in Φ.

■

Note: It is important here to distinguish between syntactic terms, and occurrences of those terms in a set of atoms. 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

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

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

A term occurrence may be identified by the atom in which the term occurs together with a path within the parse tree of that atom.

Clauses

Definition 5.6: 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 5.7: 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 5.1: If Φ is a well-formed set of atoms then there is a unique type context Γ such that $\mathit{dom}(\Gamma) = \mathit{Var}(\Phi)$ and every atom in Φ is well-typed by Γ. ■

Corollary 5.2: 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 occuring in a well-formed clause.

If Φ is a well-formed set of atoms and P is a term occuring in Φ, we write $\Phi \vdash P : \tau$ to mean $\Gamma \vdash P : \tau$ where Γ is the unique minimal type context which well-types Φ.

Example 5.1: Let us first add some additional attributes to our running example. Consider the schema $\mathcal{S}$ with classes

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

and

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

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:

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

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

$$\begin{aligned} &Y \notin Y.\mathit{neighbors} \Longleftarrow Y \in \mathit{State} \\ &Y \in Z.\mathit{neighbors} \Longleftarrow Y \in \mathit{State}, Z \in Y.\mathit{neighbors} \end{aligned}$$

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:

$$X.\mathit{popl} \leq X.\mathit{state.popl} \Longleftarrow X \in \mathit{City}$$

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

5.2 Semantics

In this section we will define a semantics for $\mathit{WOL}^{\mathcal{S}}$ in terms of the model defined in 4.3.

Semantics of terms

Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance of $\mathcal{S}$. An $\mathcal{I}$-environment, ρ, is a partial function from *Var* to $\mathbf{D}(\mathcal{I})$. We write $\mathit{Env}(\mathcal{I})$ 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 5.8: We define the semantic operator $\llbracket\cdot\rrbracket^{\mathcal{I}} : \mathit{Terms}^{\mathcal{S}} \rightarrow \mathit{Env}(\mathcal{I}) \rightarrow \mathbf{D}(\mathcal{I})$ by:

$$\begin{aligned}
\llbracket C\rrbracket^{\mathcal{I}}\rho &\equiv \sigma^{C} \qquad\qquad — C \in \mathcal{C} \\
\llbracket c^{\underline{b}}\rrbracket^{\mathcal{I}}\rho &\equiv \overline{c} \qquad\qquad — c^{\underline{b}} \text{ a constant symbol} \\
\llbracket \pi_a P\rrbracket^{\mathcal{I}}\rho &\equiv \begin{cases} (\llbracket P\rrbracket^{\mathcal{I}}\rho)a & \text{if } \llbracket P\rrbracket^{\mathcal{I}}\rho \in (\mathcal{A} \overset{\sim}{\rightarrow} \mathbf{D}(\mathcal{I})) \\ & \text{and } a \in \mathit{dom}(\llbracket P\rrbracket^{\mathcal{I}}\rho) \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket \mathit{ins}^{a} P\rrbracket^{\mathcal{I}}\rho &\equiv (a, \llbracket P\rrbracket^{\mathcal{I}}\rho) \\
\llbracket !P\rrbracket^{\mathcal{I}}\rho &\equiv \begin{cases} \mathcal{V}^{C}(\llbracket P\rrbracket^{\mathcal{I}}\rho) & \text{if } \llbracket P\rrbracket^{\mathcal{I}}\rho \in \sigma^{C} \text{ for some } C \in \mathcal{C} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket \mathit{Mk}^{C}(P)\rrbracket^{\mathcal{I}}\rho &\equiv \begin{cases} o & \text{if } \llbracket P\rrbracket^{\mathcal{I}}\rho \in \llbracket \kappa^{C}\rrbracket\mathcal{I} \text{ and } o \in \sigma^{C} \\ & \text{such that } \mathcal{K}^{C}(o) = \llbracket P\rrbracket^{\mathcal{I}}\rho \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket X\rrbracket^{\mathcal{I}}\rho &\equiv \begin{cases} \rho(X) & \text{if } X \in \mathit{dom}(\rho) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

■

For example, for the key specification of example 4.3 and instance of example 4.2,

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

and

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

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

Proposition 5.3: If $\Gamma \vdash P : \tau$ and ρ satisfies Γ then, if $[\![P]\!]^{\mathcal{I}}\rho$ is defined then $[\![P]\!]^{\mathcal{I}}\rho \in [\![\tau]\!]\mathcal{I}$. ■

Semantics of atoms

For each auxillary 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 5.9: We define the semantic operator $[\![\cdot]\!]^{\mathcal{I}} : Atoms^{\mathcal{S}} \rightarrow Env(\mathcal{I}) \rightarrow \{\mathbf{T}, \mathbf{F}\}$ by:

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

■

Semantics of clauses

In a clause, any variables occuring in the body of the clause are taken to be universally quantified, while any additional variables occuring 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.

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 $i = 1, \ldots, l$.

Example 5.2: For the instance of example 4.2, suppose the environment ρ is given by

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

then

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

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

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

■

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.

5.3 Semi-normal forms

The language WOL^S 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 transformations and recursive function definitions 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 5.10: 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 the terms of an snf atom using some predicate other than $\doteq$ will contain only variables as terms.

Lemma 5.4: 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.

Example 5.3: The atom $X \doteq Y.state.capital$ is equivalent to the snf atoms

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

■

Definition 5.11: 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, $X \in var(\Phi)$ and $Y \in var(\Phi)$;

4. if $X, Y, Z \in var(\Phi \cup \Psi)$ 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 $X, Y, Z \in var(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq ins_a Y$ and $X \doteq ins_b Z$ then $a \equiv b$ and $Y \equiv Z$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq ins_{a_i} Z$ and $Y \doteq ins_{a_i} Z$ 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 $X, Y, Z \in var(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq !Z$ and $Y \doteq !Z$ then $X \equiv Y$;

7. if $X, Y, Z \in var(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq Mk^C Z$ and $Y \doteq Mk^C Z$ then $X \equiv Y$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq Mk^C Y$ and $X \doteq Mk^C Z$ then $Y \equiv Z$;

8. if $X, Y \in var(\Phi \cup \Psi)$ and $\Phi \cup \Psi$ contains the atoms $X \doteq c^{base}$ and $Y \doteq c^{\underline{b}}$ then $X \equiv Y$; and if $\Phi \cup \Psi$ contains the atoms $X \doteq c^{\underline{b}}$ and $X \doteq d^{\underline{b}}$ then $c^{\underline{b}} \equiv d^{\underline{b}}$;

9. if $\Phi \cup \Psi$ contains an atom $X \dot{\neq} Y$ then $X \not\equiv Y$;

10. for any $X, Y \in var(\Phi \cup \Psi)$, $\Phi \cup \Psi$ does not contain both of the atoms $X \dot{\in} Y$ and $X \dot{\notin} Y$; and

11. Φ does not contain the atom False, and if Ψ contains the atom False then it contains no other atoms.

■

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 **F** 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 5.5: For any clause Δ, there is an equivalent clause Δ', unique up to the choice of variables, such that Δ' is in snf. ■

Example 5.4: 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. ■

6 Database Transformations

In this section we will show how our language, WOL^S, 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: that is it is *non-recursive* .

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 two or more 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.

6.1 Partitioning schemas and instances

Suppose $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are schemas with disjoint 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 6.1: 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 $\mathcal{K}_{\mathcal{I}}^C \equiv \mathcal{K}_i^C{}_{\mathcal{I}/\mathcal{S}_i}$ 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, $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$. However we will ensure our definitions apply equally well to the cases of 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.

6.2 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 $\mathcal{S}$ is a schema, partitioned into two schemas $\mathcal{S}_{Src}$ and $\mathcal{S}_{Tgt}$ as described earlier, and Φ is a set of $WOL^{\mathcal{SK}}$ atoms. A term P occuring 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 occurs in a term $Q \equiv !P$ 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 $P \doteq Q$ or $P \dot{\in} Q$ where Q is a target term in Φ.

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

Definition 6.2: There are three kinds of $WOL^{\mathcal{S}}$ 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) Ψ contains only target terms;
   (b) $\Phi \cup \Psi$ contains no *negative* target atoms: that is, no atoms of the form $P \not\in Q$ or $P \neq Q$ where P or Q are target terms;
   (c) for any variable $X \in var(\Phi \cup \Psi)$, 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 occuring in Ψ such that $X \not\in Var(\Phi)$.

■

So 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. 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

$$1 \in X.a \Longleftarrow X \in C, Y \in C, X.a = \overset{\circ}{Y}.a$$

where C is a class with corresponding type $\tau^C \equiv (a : \{int\})$. 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.

Example 6.1: For the schemas of the Cities and Countries databases described earlier, suppose we split the description of the instantiation of the $Country_T$ class over several transformation clauses:

$$X = Mk^{Country_T}(N),\ X.language = L \quad \Longleftarrow \quad Y \in Country_E,\ Y.name = N,\ Y.language = L$$

$$X = Mk^{Country_T}(N),\ X.currency = C \quad \Longleftarrow \quad Z \in Country_E,\ Z.name = N,\ Z.currency = C$$

Combining these clauses gives

$$\begin{aligned} X = Mk^{Country_T}(N),\ X.language = L,\ X.cu\underset{\circ}{r}rency = C \\ \Longleftarrow \quad Y \in Country_E,\ Y.name = N,\ Y.language = L \\ Z \in Country_E,\ Z.name = N,\ Z.currency = C \end{aligned}$$

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

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

$$X = Y \Longleftarrow X \in Country_E \, Y \in Country_E \, X.name = Y.name$$

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

$$\begin{aligned} X &= Mk^{Country_T}(N),\, X.language = L,\, X.currency = C \\ \Longleftarrow &\quad Y \in Country_E,\, Y.name = N,\, Y.language = L,\, Y.currency = C \end{aligned}$$

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 $Country_T$:

$$X = Mk^{Country_T}(X.name) \Longleftarrow X \in Country_T$$

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

$$\begin{aligned} X &= Mk^{Country_T}(N),\, X.language = L,\, X.currency = C,\, X.name = N \\ \Longleftarrow &\quad Y \in Country_E,\, Y.name = N,\, Y.language = L,\, Y.currency = C \end{aligned}$$

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

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

Example 6.2: We will take the first schema described in examples 4.1 and 4.3 to be our source schema. Our target schema will be given by:

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

and

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

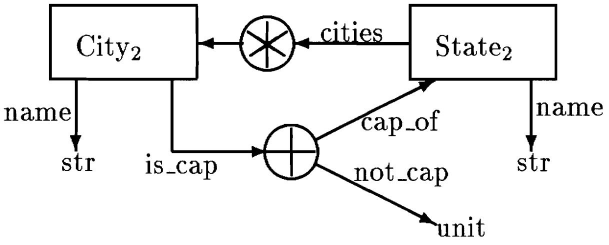


Figure 8: A target schema for Cities and States

with key specification

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

The schema is illustrated in figure 8. 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

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

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

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

$$\begin{array}{l}
W \in Y.cities,\ W.name = C \impliedby \\
\qquad Z \in City_A,\ Z.name = C,\ Y \in State_2, Y.name = Z.state.name
\end{array}$$

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 $City_2$:

$$X = Mk^{State_2}(X.name) \impliedby X \in State_2$$

$$\begin{aligned} X = Mk^{City_2}(W),\ \pi_{name}W = X.name,\ \pi_{state}W = Y \\ \impliedby X \in City_2,\ Y \in State_2,\ X \in Y.cities \end{aligned}$$

$$\begin{aligned} &Y \in State_2, X \in Y.cities \impliedby X \in City_2 \\ &X = Y \ = \ X \in State_2, Y \in State_2, Z \in X.cities, Z \in Y.cities \end{aligned}$$

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. ∎

6.3 Transformations of instances

Definition 6.3: Suppose that **Tr** is a transformation program from schema $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$, and that $\mathcal{I}_{Src}$ is an instance of $\mathcal{S}_{Src}$. Then an instance $\mathcal{I}_{Tgt}$ of $\mathcal{S}_{Tgt}$ is said to be a **Tr-transformation** of $\mathcal{I}_{Src}$ iff, for each clause $(\Psi \impliedby \Phi) \in$ **Tr**, $\mathcal{I}_{Src} \cup \mathcal{I}_{Tgt}$ satisfies $\Psi \impliedby \Phi$. ∎

Unfortunately the **Tr**-transformation of an instances 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 addition 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 6.4: 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 $dom(\rho) = Var(\Phi)$ and $[\![\Phi]\!]^{\mathcal{I}}\rho = \mathbf{T}$ there is *at most one* extension of ρ, ρ' say, such that $dom(\rho') = Var(\Psi, \Phi)$ and $[\![\Psi]\!]^{\mathcal{I}}\rho' = \mathbf{T}$. ∎

So a clause in 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 6.5: Suppose $\mathcal{I}$ and $\mathcal{I}'$ are instances of a schema $\mathcal{S}$, and f^C is a family of injective functions, $f^C : \sigma^C \rightarrow \sigma'^C$. 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 6.1: If $\mathcal{I}$, $\mathcal{I}'$ and $\mathcal{I}''$ are instances of $\mathcal{S}$ then

1. $\mathcal{I} \preceq \mathcal{I}$ — the relation $\preceq$ is symmetric;
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 $\mathcal{I} \cong \mathcal{I}'$.

■

Definition 6.6: 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 6.3: We will describe the transformation specified by the transformation program in example 6.2 on the instance of $\mathcal{S}_{Src}$ defined in example 4.2.

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

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

and

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

■

6.4 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 be 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 6.7: 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:

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

$$\frac{\vdash (C, \mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)}{\vdash (C, \mu.\pi_{a_i}) : \tau_i} \qquad \frac{\vdash (C, \mu) : \llparenthesis a_1 : \tau_1, \ldots, a_k : \tau_k \rrparenthesis}{\vdash (C, \mu.ins_{a_i}) : \tau_i}$$

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.

Suppose P is a term occuring in a well-formed set of atoms, Φ. We say P has term path (C, μ) in Φ, written $\Phi \vdash P : (C, \mu)$, if

1. $P \equiv C$ and $\mu = \epsilon$ (the empty string), or
2. $P \equiv !Q$ and $\mu = \mu'.!$ and $\Phi \vdash Q : (C, \mu')$, or
3. $P \equiv \pi_a Q$ and $\mu = \mu'.\pi_a$ and $\Phi \vdash Q : (C, \mu')$, or
4. P occurs in a term $Q \equiv ins_a(P)$ in Φ, and $\mu = \mu'.ins_a$ and $\Phi \vdash Q : (C, \mu')$, or
5. Φ contains an atom $P \dot{\in} Q$ and $\mu = \mu'.\dot{\in}$, and $\Phi \vdash Q : (C, \mu')$, or
6. Φ contains an atom $P \doteq Q$ and $\Phi \vdash Q : (C, \mu)$.

Example 6.4: Consider the set of atoms

$$\begin{aligned} \Phi \ \equiv\ & W \in Y.cities, W.name = N, Z \in City_A, Z.name = N, \\ & Y \in State_2, Y.name = Z.state.name \end{aligned}$$

for the schema of the previous examples. Here the term N has two term paths: $\Phi \vdash N : (City_A, \dot{\in}!\pi_{\text{name}})$ and $\Phi \vdash N : (State_2, \dot{\in}!\pi_{\text{cities}}\dot{\in}!\pi_{\text{name}})$. ■

The following lemma tell us that the typing rules for term paths and for terms occuring 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 6.2:

1. If Φ is a well-formed set of atoms and $\Phi \vdash P : (C, \mu)$ in Φ and $\Phi \vdash P : \tau$ 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 occuring in Φ such that $\Phi \vdash P : (C, \mu)$. Further there is a *unique* (up to variable renaming) smallest set of semi-normal form atoms, Φ, such that $\Phi \vdash X : (C, \mu)$ for some $X \in \mathit{Var}(\Phi)$.

■

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 associate a particular term with some 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.

Definition 6.8: Suppose **Tr** is a transformation program, and Θ is a set of atoms. Then Θ is said to be **characterized** in **Tr** by a set of variables $\{X_1, \ldots, X_k\}$ iff $\{X_1, \ldots, X_k\} \subseteq \mathit{Var}(\Theta)$ and for any instance $\mathcal{I}$ satisfying **Tr**, and for any $\mathcal{I}$-environment ρ such that $\mathit{dom}(\rho) = \{X_1, \ldots, X_k\}$, there is *at most one* extension ρ' of ρ such that $\mathit{dom}(\rho') = \mathit{Var}(\Theta)$ and $[\![\Theta]\!]^{\mathcal{I}}\rho' = \mathbf{T}$. ■

Definition 6.9: Suppose **Tr** is a transformation program, and (C, μ) a term path. A characterizing formula for (C, μ) in **Tr** is a set of atoms Θ together with a set of variables $\{Y_1, \ldots, Y_n\} \subseteq \mathit{Var}(\Theta)$ and a distinguished variable $X \in \mathit{Var}(\Theta)$, such that $\{Y_1, \ldots, Y_n\}$ characterize Θ and $\Theta \vdash X : (C, \mu)$.

We write $\Theta_X(Y_1, \ldots, Y_n)$ for the characterizing formula 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 $\{Y_1, \ldots, Y_N\} = \mathit{Var}(\Theta)$. However there are also some more useful examples of characterizing formula.

Example 6.5: A characterizing formula for the term path $(\mathit{State}', \dot{\in})$ in the transformation program of example 6.2 would be

$$\Theta_X(N) \;\equiv\; X \in \mathit{State}_2,\, \pi_{name}(!X) = N$$

since the program implies the target constraint

$$X = Y \;\Longleftarrow\; X \in \mathit{State}_2,\, Y \in \mathit{State}_2,\, \pi_{name}(!X) = N,\, \pi_{name}(!Y) = N$$

A characterizing formula for the term path $(\mathit{City}', \dot{\in})$ would be

$$\Theta'_X(N, Z) \;\equiv\; X \in \mathit{City}_2,\, \pi_{name}(!X) = N,\, Z \in \mathit{State}_2,\, X \in \pi_{cities}(!Z)$$

The following lemma tells us that the characterizing formulas for any other target terms paths could be formed from these two. ■

Lemma 6.3:

1. If $\vdash (C, \mu) : (a_1 : \tau_1, \ldots, a_k : \tau_k)$ and $\Theta_Y(V)$ is a characterizing formula for (C, μ), then $\Theta'_X(V) \equiv (\Theta_Y(V) \cup \{X \doteq \pi_{a_i} Y\})$ is a characterizing formula for $(C, \mu.\pi_{a_i})$.
2. If $\vdash (C, \mu) : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ and $\Theta_Y(V)$ is a characterizing formula for (C, μ), then $\Theta'_X(V) \equiv (\Theta_Y(V) \cup \{\mathit{ins}_{a_i} X \doteq Y\})$ is a characterizing formula for $(C, \mu.\mathit{ins}_{a_i})$.
3. If $\vdash (C, \mu) : D$, $D \in \mathcal{C}$, and $\Theta_Y(V)$ is a characterizing formula for (C, μ), then $\Theta'_X(V) \equiv (\Theta_Y(V) \cup \{X \doteq !Y\})$ is a characterizing formula for $(C, \mu.!)$.

■

Consequently, if we are describing a database instance, or a transformation, we only need characterization formula 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

$$\Theta_X(Z) \equiv (X \in C, X = \mathit{Mk}^C(Z))$$

is a characterization formula for the term path $(C, \dot{\in})$. This gives us a useful special case: if every set type occuring 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 $\mathcal{S}$.

Normal Forms

Definition 6.10: Suppose that **Tr** is a transformation program, with target constraints $\mathbf{Tr}^T$. A clause $\Psi \Longleftarrow \Phi$ is in **normal form** for **Tr** iff

1. $\Psi \Longleftarrow \Phi$ is in semi-normal form;
2. Φ contains no target terms;
3. For any target term path (C, μ), if $\Phi \cup \Psi \vdash X : (C, \mu)$ for some $X \in \mathit{Var}(\Phi \cup \Psi)$, and $\Phi \cup \Psi \vdash (C, \mu) : \tau$ where τ is *not* a set type, then $\Psi_X(Y_1, \ldots, Y_k)$ is a characterizing formula for (C, μ) in $\mathbf{Tr}^T$ for some variables $\{Y_1, \ldots, Y_k\} \subseteq \mathit{Var}(\Phi)$;
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 $Y \in \mathit{Var}(\Phi \cup \Psi)$; and
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.

■

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 6.6: The following are normal-form clauses equivalent to the transformation clauses of example 6.2:

$$Y \in \mathit{State}', W = !Y, N = \pi_{name} W \; \Longleftarrow \; X \in \mathit{State}, U = !X, N = \pi_{name} U$$

$$\begin{aligned} Y \in \mathit{State}', W \in \mathit{City}', U = !Y, N = \pi_{name} U, V = !W, C = \pi_{name} V,\\ T = \pi_{iscap} V, T = \mathit{ins}_{yes}(), S = \pi_{cites} U, W \in S\\ \Longleftarrow \; X \in \mathit{State}, Q = !X, N = \pi_{name} Q, Z \in \mathit{City}, R = !Z, C = \pi_{name} R,\\ Z = \pi_{state} Q, X = \pi_{capital} R \end{aligned}$$

$$\begin{aligned} Y \in \mathit{State}', W \in \mathit{City}', U = !Y, N = \pi_{name} U, V = !W, C = \pi_{name} V,\\ T = \pi_{iscap} V, T = \mathit{ins}_{no}(), S = \pi_{cites} U, W \in S\\ \Longleftarrow \; X \in \mathit{State}, Q = !X, N = \pi_{name} Q, Z \in \mathit{City}, R = !Z, C = \pi_{name} R,\\ Z = \pi_{state} Q, O = \pi_{capital} R, X \neq O \end{aligned}$$

■

6.5 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 6.11: A **description** 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 or atoms of the form $X \not\in Y$;
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 $Y \doteq X.a_i$ 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 $\mathit{ins}_{a_i} Y \doteq X$.

■ 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 6.6, 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, though each target variable can match at most one variable from the unifying clause.

Definition 6.12: 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 $\mathcal{U}(\Psi) \Longleftarrow \mathcal{U}(\Phi)$.

Lemma 6.4: If Ξ and Δ are clauses and $\mathcal{U}$ is a unifier from Δ to Ξ, then $\Xi \models \mathcal{U}(\Xi)$. ■

Unfolding

Let Ξ and Δ be 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

$$\begin{aligned} \Psi &\equiv \mathcal{U}(\Psi) \\ \Phi &\equiv (\mathcal{U}(\Phi_t) \setminus \Psi_u) \cup \Phi_u \end{aligned}$$

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 $\Phi \subseteq \Phi^* \subseteq \mathcal{U}(\Psi_t) \cup \Phi_u$, 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 Ψ 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.

Definition 6.13: Ξ is said to be **unfoldable** on $\Delta, \mathcal{U}$ 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. For each clause $\Psi \Longleftarrow \Phi \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U}), \mathcal{U}(\Phi_t) \not\subseteq \Phi$; and
3. For each clause $\Psi \Longleftarrow \Phi \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$, Ψ is characterized by $\mathit{Var}(\Phi)$.

■ 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 6.5: If $\mathcal{U}$ is a unifier from Δ to Ξ and $\Psi \Longleftarrow \Phi \in \mathit{Unfold}(\Xi, \Delta, \mathcal{U})$, then $\Delta, \Xi \models \Psi \Longleftarrow \Phi$. ■

6.6 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 occuring 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 variables at each point at which values are potentially being created. For each variable a record is kept of which transformation clauses have *"touched"* that variable 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 variable 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. Ξ_i is unfoldable on $C_i, \mathcal{U}_i$ and $D_{i+1} \in$ *Unfold*$(D_i, C_i, \mathcal{U}_i)$ for $i = 0, 1, 2, \ldots$.

A **decoration** of an unfolding sequence is a sequence of maps, $\delta_0, \delta_1, \ldots, \delta_i, \ldots$, such that

1. $\delta_i : \text{var}(\Xi_i) \to \mathcal{P}_{fin}(\mathbf{Tr})$ for $i = 0, 1, 2, \ldots$,
2. $\delta_0(Z) = \emptyset$ for each $Z \in \text{var}(\Xi_0)$, and
3. $\delta_{i+1}(Z) = \begin{cases} \delta_i(Z) & \text{if } Z \in \text{var}(\Xi_i) \setminus dom(\mathcal{U}_i) \\ \{\Delta_i\} \cup \bigcup\{\delta_i(W) \mid W \in dom(\mathcal{U}_i)\} & \text{otherwise} \end{cases}$
for $i = 0, 1, 2, ldots$

Proposition 6.6: Suppose $\Xi_0, \Delta_0, \Xi_1, \Delta_1, \ldots, \mathcal{U}_0, \mathcal{U}_1, \ldots$ is an infinite unfolding sequence for **Tr**. Then there is a k such that, for every $X \in dom(\mathcal{U}_k)$, $T_k \in \delta_k(X)$. ■

Proof outline: We assume that we have a an infinite unfolding sequence, and a clause $\Delta \in \mathbf{Tr}$ such that Δ occurs infinitely often in the unfolding sequence, and for each occurrence of Δ there is variable X in the domain of the corresponding unifier, $X \in dom(\mathcal{U}_k)$ say, such that $\Delta \notin \delta_k(X)$. We prove that there is then an infinite unfolding sequence that does not involve Δ.

If the proposition is false then we can repeatedly apply the above result, to get that there is an infinite unfolding sequence with unfolding clauses taken from the empty set of clauses, which is clearly false. ■

This proposition leads to a fine definition of recursion, for which, as we will see, tests can be built into out normalization algorithm.

Definition 6.14: Suppose **Tr** is a transformation program from $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$. Then **Tr** is said to be **recursive** iff there is an unfolding sequence for **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 St, and $\Delta_k \in \delta_k(X)$ for each $X \in dom(\mathcal{U}_k)$. ■

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 on unfolding.

In addition we need syntactic restrictions on programs in order to ensure that any variants or object-identities generated by the program are fully specified by the program. We say that a transformation program, **Tr**, is **safe** iff every clause in **Tr** is deterministic with respect to **Tr** and, for any clause $\Delta \in \mathbf{Tr}$, $\Delta \equiv \Psi \Longleftarrow \Phi$, instance $\mathcal{I}$ and environment ρ such that $[\![\Psi, \Phi]\!]^{\mathcal{I}}\rho = True$, and $X \in Var(\Psi) \setminus Var(\Phi)$ we have

1. if $\Delta \vdash X : C, C \in \mathcal{C}$, then there is a clause $\Delta' \in \mathbf{Tr}$, $\Delta' \equiv \Psi' \Longleftarrow \Phi'$, and an environment ρ' such that $[\![\Phi']\!]^{\mathcal{I}}\rho' = True$ and there is an atom $(Z \doteq !Y) \in \Psi'$ and $\rho'(Y) = \rho(X)$;
2. if $\Delta \vdash X : (a_1 : \tau_1, \ldots, a_n : \tau_n)$ then for each a_i there is a clause $\Delta' \in \mathbf{Tr}$, $\Delta' \equiv \Psi' \Longleftarrow \Phi'$,

and an environment ρ' such that $[\![\Phi']\!]^{\mathcal{I}}\rho' = \mathit{True}$ and there is an atom $(Z \doteq \pi_{a_i} Y) \in \Psi'$ and $\rho'(Y) = \rho(X)$;

3. if $\Delta \vdash X : \langle\!| a_1 : \tau_1, \ldots, a_n : \tau_n |\!\rangle$ then for some a_i there is a clause $\Delta' \in \mathbf{Tr}$, $\Delta' \equiv \Psi' \Longleftarrow \Phi'$, and an environment ρ' such that $[\![\Phi']\!]^{\mathcal{I}}\rho' = \mathit{True}$ and there is an atom $(Y \doteq \mathit{ins}_{a_i} Z) \in \Psi'$ and $\rho'(Y) = \rho(X)$.

Proposition 6.7: Given any non-recursive, safe transformation program **Tr** from $\mathcal{S}_{Src}$ to $\mathcal{S}_{Tgt}$, if **Tr** is complete then there is an equivalent transformation program **Tr**$'$ such that **Tr**$'$ is in normal form. Further there is an algorithm which will compute such a **Tr**$'$ if **Tr** is complete, or terminate reporting that **Tr** is not complete otherwise. ■

Such a **Tr**$'$ can be computed by taking the maximal unfolding sequences of the description clauses for $\mathcal{S}_{Tgt}$ with clauses from **Tr**.

7 Implementation Issues

The ideas and methods presented in section 6 suggest a plan for implementing transformation programs specified in *WOL*. Transformation programs written in *WOL* would first be translated into semi-normal form (definition 5.11). The snf transformation program would then be normalized using an algorithm based on proposition 6.7, and finally the normal form transformation program is translated into code in an appropriate DBPL, such as CPL, which can then be executed. Such an architecture is shown diagrammatically in figure 9. In fact this corresponds to the architecture for the established implementation of the more restrictive language *TSL* (see section 2.3). Certain constraints and type information relevant to a *WOL* transformation program may be derived directly from the meta-data stored of the various relevant databases, as well as being input by the user in a (possibly extended) *WOL* syntax.

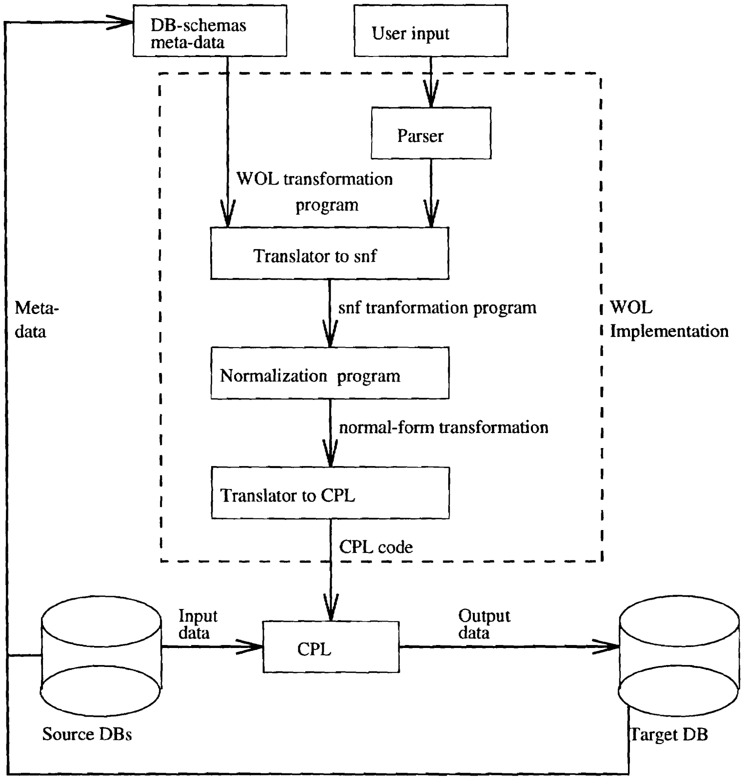


Figure 9: Architecture for proposed implementation of *WOL* transformations

The algorithm for normalizing a transformation program suggested by proposition 6.7 would work by first generating a set of *description clauses*, then generating all the possible unfolding sequences, and collecting together the resulting normal form clauses. If recursion was detected while generating the unfolding sequences then the algorithm would halt and report that the transformation program is recursive.

However such an algorithm, if implemented naively, would not be tolerably efficient. This is because there are too many possible unfolding sequences and too many possible unifiers at each stage. It is necessary to take various steps to make the search space smaller, and the implementation more efficient.

In this section we will briefly discuss various issues relevant to making a practical implementation of *WOL* transformation programs. These include optimizations to the normalizing algorithm, and restrictions on the relevant target database schemas and constraints necessary to make the resultant transformation code suitably efficient. Some of these optimizations have already been incorporated in the existing implementation of TSL, others represent improvements to the existing implementations, and others are factors relevant only to the more general transformations expressible in *WOL*.

7.1 Optimizing the Normalization Algorithm

The algorithm suggested by proposition 6.7, in a naive form, would be:

1. Generate a set of description clauses;
2. For each description clause generate maximal unfolding sequences, while testing for recursion;
3. If recursion is detected then raise a "recursive transformation program" error;
4. Return final clauses from unfolding sequences which are in normal form.

However generating all possible maximal unfolding sequences would be extremely inefficient, and so it is necessary to find ways of reducing our attention to a subset of relevant unfolding sequences.

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 unfolding clauses

If we have multiple unfolding clauses, it may be possible to apply the clauses in a number of different orders and get equivalent unfolding sequences. For example suppose we had a target clause

$$\Xi \equiv (\Psi \Longleftarrow X = C, Y \in X, Z =!Y, U = \pi_a Z, V = \pi_b Z)$$

and two transformation clauses

$$\begin{aligned}\Delta_1 &\equiv (X_1 = C, Y_1 \in X_1, Z_1 =!Y_1, U_1 = \pi_a Z_1 \Longleftarrow \Phi_1)\\ \Delta_2 &\equiv (X_2 = C, Y_2 \in X_2, Z_2 =!Y_2, V_2 = \pi_b Z_2 \Longleftarrow \Phi_2)\end{aligned}$$

where Φ_1 and Φ_2 contain no target atoms. Then we can unfold first with the clause Δ_1, removing the atom $U = \pi_a Z$, and then with the clause Δ_2 removing the remaining atoms, or we can unfold first with the clause Δ_2, removing the atom $V = \pi_b Z$, and then with the clause Δ_1, removing the remaining atoms. The results of these two unfolding sequences are equivalent. It is clear that the number of equivalent unfolding sequences is potentially exponential in the number of transformation clauses. We therefore need to find a way of avoiding generating multiple equivalent transformation sequences.

In order to solve this problem we must assume that there is an ordering on the transformation clauses of a program. Suppose our transformation clauses are $\Delta_1, \ldots, \Delta_k$. We then use the following algorithm.

```
Repeat
    For $i = 1, \ldots, k$ do
        Unfold on $\Delta_i$ as many times as possible;
        Ignore unfoldings which do not bind an atom which was
        not present during a previous attempt to unfold on $\Delta_i$;
    od
Until no more unfoldings are possible
```

Initially every atom in the target clause counts as being "not present in a previous attempt to unfold" on each clause. In later iterations of the *repeat* loop, only unfoldings which bind newly introduced atoms are applied. Note that it is possible to unfold on the same target clause multiple times.

7.2 Maintaining Characterizing Sets of Variables

Our definition of valid unfoldings (definition 6.13) requires that the clause resulting be characterized by the variables in the body of the clause. In general this is a difficult condition to test, and we need to make some simplifying assumptions on our target constraints.

We assume that the only target constraints that introduce non-trivial characterizing formulae are those that define the keys of classes. In particular we assume there are no target constraints of the form $X \doteq Y \Longleftarrow \Phi$ in the transformation program, though there may be constraints of the form $X \doteq Mk^C(Y) \Longleftarrow \Phi$. (Constraints of the first form may still exist, but they will be used only to ensure integrity of the target database, and not to determine a transformation). Any non-trivial of characterizing formula are therefore derived using target constraints of this second form, together with the lemma 6.3.

Rather than compute characterizing sets of variables for each clause in an unfolding sequence, we propose computing minimal characterizing sets of variables on the description clauses, and then *tracing* these sets of variables through an unfolding sequence.

Suppose $\mathcal{U}$ is a unifier from a clause Δ to Ξ, and $V \subseteq Var(\Xi)$ is a set of variable. Then we define

$$\mathcal{U}(V) \equiv \{\mathcal{U}(X) \mid X \in V \cap dom(\mathcal{U})\} \cup (V \setminus dom(\mathcal{U}))$$

If $\Xi_0, \Delta_0, \ldots, \Xi_k, \Delta_k, \Xi_{k+1}$, $\mathcal{U}_0, \ldots, \mathcal{U}_k$ is an unfolding sequence, such that $\Xi_i \equiv (\Psi_i \Longleftarrow \Phi_i)$ for $i = 1, \ldots, k+1$, and $V \subseteq Var(\Phi_0)$, then we say that the unfolding sequence **preserves** V iff

$$\mathcal{U}_i \circ \ldots \circ \mathcal{U}_0(V) \subseteq Var(\Phi_{i+1})$$

for $i = 0, \ldots, k$.

It follows that for any valid unfolding sequence, $\Xi_0, \Delta_0, \ldots, \Xi_k, \Delta_k, \Xi_{k+1}$, $\mathcal{U}_0, \ldots, \mathcal{U}_k$, where Ξ_0 is a description clause, there must be a set of minimal characterizing set of variables for Ξ_0, say V, such that the unfolding sequence preserves V. Having computed the characterizing sets of variables for a description clause, it is easy to ensure this condition is satisfied by any unfolding sequences we compute.

7.3 Restrictions on Applicable Schemas

If we are to continue to use CPL, or an extension of CPL, as a target language for implementing transformations, then it is will be necessary to restrict the possible target schemas so that the transformations can be expressed in CPL in a reasonable manner. In particular CPL is not well suited to handling transformations where the target involves deeply nested sets. This is because the values which are associated with each object identity must be created be a single expression, rather than being created incrementally.

For example, suppose our target schema contained a class C with associated type

$$\tau^C \equiv (a : int, b : \{int\})$$

and the key specification for C was

$$\mathcal{K}^C(u) \equiv (\mathcal{V}^C(u))(a)$$

That is, the attribute a functions as the key for C. Suppose our source database contains a class D, with $\tau^D \equiv int$, and that we have the transformation clauses

$$\begin{array}{rcl} X \in C, X.a = Y & \Longleftarrow & Z \in D, Y = !Z \\ Y \in X.b & \Longleftarrow & Z \in D, Y = !Z, X \in C \end{array}$$

Then to apply both of these clauses individually would require $2n$ accesses to the source databases, whereas combining the two in a CPL-style comprehension would require n^2 accesses to the source database, n being the size of the database. In general the number of accesses in a comprehension expression will be exponential in the level of nesting of sets.

I believe a reasonable restriction to impose on target databases is that all set types occuring in the target database schema must be either of the form $\{C\}$ for some class C, or $\{\underline{b}\}$ for some base type $\underline{b}$. This would also simplify the computing of characterizing sets of variables.

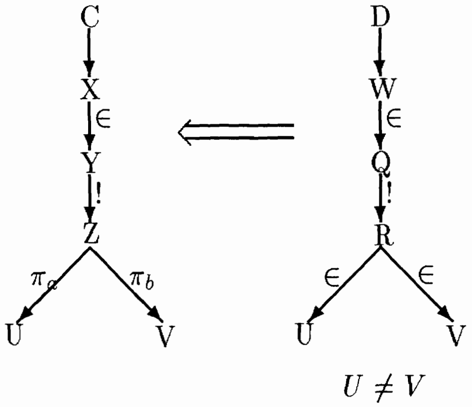
7.4 Tree Representation of Clauses

The existing implementation of TSL represents clauses as pairs of sets snf atoms. I believe significant improvements in the unification and unfolding algorithms could be gained by making use of the range-restriction conditions and representing clauses as pairs of *forests*, together with some extra atoms. Such forests would have variables for the nodes of the trees, with the root of each tree being marked by a class, and edges marked with the symbols $\dot{\in}$, !, $\pi_a, \pi_b, \ldots$ and so on. Atoms of the forms $X \doteq Y$, $X \dot{\neq} Y$ and $X \dot{\notin} Y$ would still be represented as additional sets of atoms.

For example the snf 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



Such a representation of clauses should provide significant improvements in the searching for possible unifiers between clauses, and also make it possible to reject invalid unifiers at an earlier stage.

Though it has not so far seemed to be worth rewriting the TSL implementation in order to use this representation, I believe that it would be worth incorporating in a future implementation of *WOL*. However doing so would make it more difficult to re-use code from the existing TSL implementation.

8 Transformations of Alternative Collection Types

The type system we defined in 4.1 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 are all examples of a kind of categorical structure known as a *monad* (see [39]). Though the details of *monads* are not important here, the important conclusion is simply that it is natural to use the same, or similar constructs to program over sets, bags and lists. 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

$$\begin{aligned} X = Mk^{\mathrm{Person}}(N),\ X.name = N,\ Mk^{\mathrm{Person}}(C) \in X.children \\ \Longleftarrow\ \ P \in Parents,\ P.pname = N,\ P.cname = C \end{aligned}$$

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

$$\begin{aligned} Parents \ \equiv\ & [(pname = \text{“Susan”}, cname = \text{“Jeremy”}), \\ & (pname = \text{“Susan”}, cname = \text{“Chris”}), \\ & (pname = \text{“Val”}, cname = \text{“Alexander”}), \\ & (pname = \text{“Val”}, cname = \text{“Nicholas”})] \end{aligned}$$

then we would like our class *Person* to have objects

$$\sigma^{Person} = \{Susan, Jeremy, Chris, Val, Alexander, Nicholas\}$$

with associated values

$$\begin{aligned}
\mathcal{V}^{Person}(Susan) &= (name = \text{"Susan"}, children = [Jeremy, Chris]) \\
\mathcal{V}^{Person}(Val) &= (name = \text{"Val"}, children = [Alexander, Nicholas]) \\
\mathcal{V}^{Person}(Jeremy) &= (name = \text{"Jeremy"}, children = []) \\
\mathcal{V}^{Person}(Chris) &= (name = \text{"Chris"}, children = []) \\
\mathcal{V}^{Person}(Alexander) &= (name = \text{"Alexander"}, children = []) \\
\mathcal{V}^{Person}(Nicholas) &= (name = \text{"Nicholas"}, children = [])
\end{aligned}$$

.

8.1 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 we choose does not matter here, though later we'll be settling on the set of strings of natural numbers ordered lexicographically.

Definition 8.1: Suppose D is some set. A **list** over domain D is a partial function $l : \mathbf{L} \overset{\sim}{\rightarrow} D$ 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 $f : dom(l) \rightarrow dom(l')$ such that if $i, j \in dom(l)$, $i < j$, then $f(i) < f(j)$, and for every $i \in dom(l)$, $l(i) = l'(f(i))$.

Lemma 8.1: The relation $\approx$ is an equivalence on lists. ■

We will consider to 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.

8.2 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 5.2 and 5.4.

We will expand the definition of the semantic operator on types (definition 4.2) 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 5.9, 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

$$[\![P \dot{\in} Q]\!]^{\mathcal{I}}\rho \;\equiv\; \begin{cases} \mathbf{F} & \text{if } [\![P]\!]^{\mathcal{I}}\rho \neq ([\![Q]\!]^{\mathcal{I}}\rho)(i) \\ & \text{for all } i \in \mathit{dom}([\![Q]\!]^{\mathcal{I}}\rho) \\ \left\{n \mid ([\![Q]\!]^{\mathcal{I}}\rho)(n) = [\![P]\!]^{\mathcal{I}}\rho\right\} & \text{otherwise} \end{cases}$$

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

$$\begin{aligned} X = \mathit{Mk}^{\mathrm{Person}}(N),\ X.\mathit{name} = N,\ \mathit{Mk}^{\mathrm{Person}}(C) \in X.\mathit{children} \\ \Longleftarrow\quad P \in \mathit{Parents},\ P.\mathit{pname} = N,\ P.\mathit{cname} = C \end{aligned}$$

this would seem sufficient: we could take the clause to mean, if P is in list *Person* with precedence i, then $\mathit{Mk}^{Person}(C)$ 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

$$X \in L_1 \;\Longleftarrow\; Y \in L_2,\ Z \in L_3,\ \Phi$$

where L_1, L_2 and L_3 are lists, it is necessary to combine the precedences of the two $\in$ 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 8.2: 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 5.6), to say that a clause consists of two *ranked* sets of atoms: the *head* and the *body.*

Definition 8.3: 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 [("a","d"),("a","e"),("a","f"),("b","d"),("b","e"),("b","f"),("c","d"),("c","e"),("c","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 8.4: 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,

$$[\![\{X \in^1 Y.a,\ Z \in^2 Y.b,\ Z = ins_d W,\ U \in^3 W\}]\!]^{\mathcal{I}}_{(4.2.15)}\rho = \mathbf{T}$$

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 $[\![Z = ins_d W]\!]^{\mathcal{I}}\rho = \mathbf{T}$.

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 $dom(\rho) = Var(\Phi)$, 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 8.5: 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 II

Observable Properties of Models for Recursive Data-Structures

9 A Data Model Based on Regular Trees

As we remarked in section 4, 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. In later sections of part II we will see that the observable properties of a data-model are dependent on the particular predicates we consider to be available for comparing object-identities.

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 4.3, fails to capture.

In this section we will present a model sharing definition 4.1 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's fairly easy to form an intuitive understanding of regular trees, based on diagrammatic representations, to formulate them rigorously requires a surprising amount of care. 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.

9.1 Regular Trees

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 $\phi \in D$.

A *tree* is a mapping from a tree domain to some alphabet $\mathbf{A}$.

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

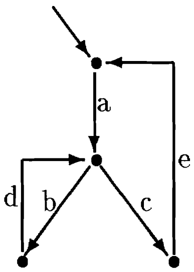
$$t.\phi(x) \equiv t(\phi x)$$

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 $\phi \in dom(t)$ such that $t' = t.\phi$.

Definition 9.1: A **regular tree** is a tree with only finitely many subtrees. ■

There several ways of constructing regular trees: one can view then 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:

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

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.

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 to be the largest relation such that, if $t_1 \asymp \tau_2$ then

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

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

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

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 $\phi \in dom(t_i)$.

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.

9.2 Trees of Types

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

1. the symbol $\in$,
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 $\underline{b}$ a disjoint countable set of *constant symbols*, $c^{\underline{b}}, \ldots$.

Definition 9.2: We define the mapping *TTree* from types to sets of regular trees over this alphabet to be the largest such that:

1. if $t \in \mathit{TTree}(\underline{b})$ then $t = \langle c^{\underline{b}} \epsilon \rangle$ for some constant symbol $c^{\underline{b}}$,
2. if $t \in \mathit{TTree}((a_1 : \tau_1, \ldots, a_k : \tau_k))$ then $t = \langle \pi_{a_1} t_1, \ldots, \pi_{a_k} t_k \rangle$ where $t_i \in \mathit{TTree}(\tau_i)$ for $i = 1, \ldots, k$.
3. if $t \in \mathit{TTree}(\llparenthesis a_1 : \tau_1, \ldots, a_k : \tau_k \rrparenthesis)$ then $t = \langle ins_{a_i}\ t' \rangle$ where $t' \in \mathit{TTree}(\tau_i)$ some $i \in 1, \ldots, k$.
4. if $t \in \mathit{TTree}(\{\tau\})$ then $t = \langle \in t_1, \ldots, \in t_k \rangle$ where $k \geq 0$ and $t_i \in \mathit{TTree}(\tau)$ for $i = 1, \ldots, k$.
5. if $t \in \mathit{TTree}(C)$ then $t = \langle St^C t' \rangle$ where $t' \in \mathit{TTree}(\tau^C)$.

■

$\mathit{TTree}(\tau)$ represents the set of all regular trees of type τ.

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 $\llparenthesis a_1 : \tau_1, \ldots, a_n : \tau_n \rrparenthesis$ 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 $\mathcal{S}_{Tgt}{}^{C}$, going to a tree of type τ^{C}.

9.3 Instances

Definition 9.3: An **instance**, $\omega^{\mathcal{C}}$, consists of a family of finite sets of regular trees, $\omega^{C} \subseteq$ *TTree*(C), for each $C \in \mathcal{C}$.

A regular tree, t, is said to be a **value of type** τ iff $t \in$ *TTree*(τ).

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 $\omega^{\mathcal{C}}$. ■

Example 9.1: Let us consider an instance for the schema described in example 4.1. The

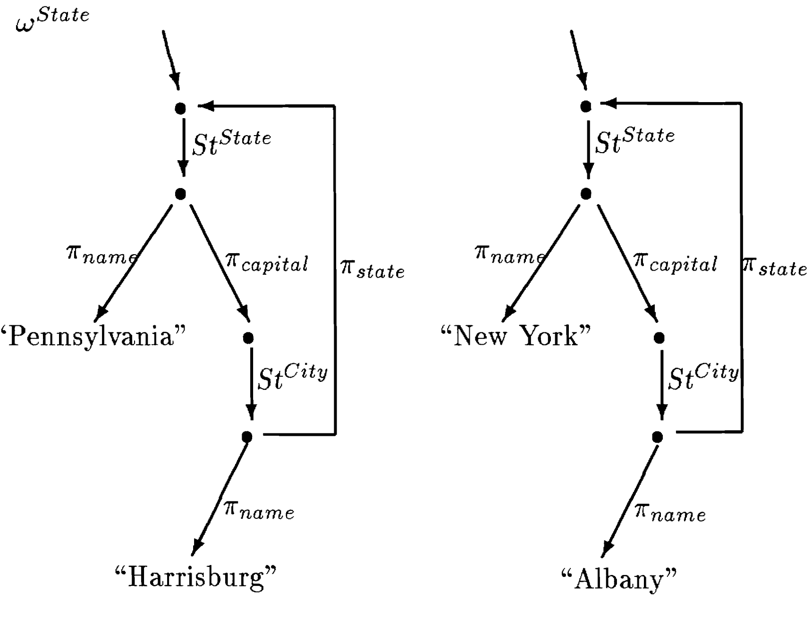


Figure 10: States from instance

instance consists of two sets, ω^{State} and ω^{City}. The set ω^{State} contains the regular trees shown in figure 10 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 11

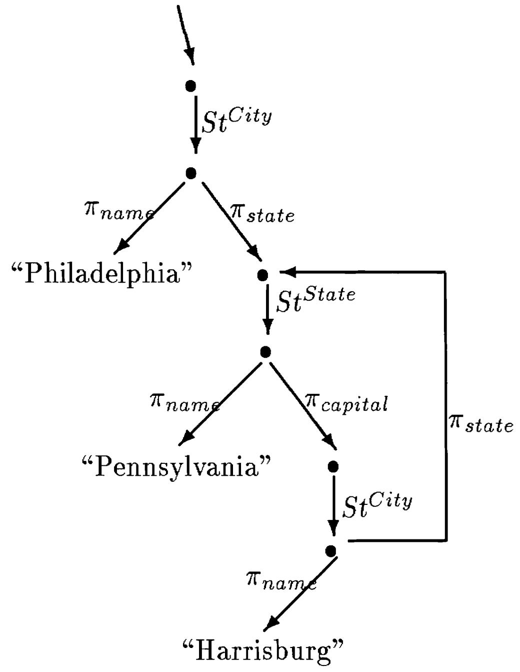


Figure 11: Cities from instance

representing the city *Philadelphia.* However, in order to be a valid instance, ω^{City} must at least contain regular trees for Harrisburg and Albany. ■

10 Bisimulation and Correspondence of Object Identities

It seems clear that the object-identity based model of definition 4.3 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, differing only in the choice of object identities.

We would like a semantic model where two instances are considered to be different if and only if they are distinguishable. However to talk about whether two instances are distinguishable assumes some latent language for querying the databases, and of course the notion of distinguishability is dependent on this language and the predicates available in it.

The regular-tree based model defined in 9.3 seems to provide such a semantic model if we assume that it is not possible to observe or compare object-identities/recursive-values directly: in particular, if it is not possible to test whether two object-identities are equal, or whether a particular object identity is included in a set of object identities. In this section we construct an equivalence relation $\approx$ on (object identity based) database values representing observational equivalence in this scenario. In section 10.1 we will show that the show how this equivalence gives rise to regular tree (definition 9.3) database instances. In section 11 we will show how these different equivalences/models correspond to different observational equivalences induced by varying the assumptions about which operators are available on object identities in a query language.

Definition 10.1: A **correspondence** between two families of object identifiers $\sigma^{\mathcal{C}}$ and $\sigma'^{\mathcal{C}}$ is a family of binary relations $\sim^{C} \subseteq \sigma^{C} \times \sigma'^{C}$.

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

■

A correspondence $\sim^{\mathcal{C}}$ is said to be **consistent** with instances $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ and $\mathcal{I}' = (\sigma'^{\mathcal{C}}, \mathcal{V}'^{\mathcal{C}})$ if for each $C \in \mathcal{C}$ and all $o \in \sigma^{C}$, $o' \in \sigma'^{C}$, if $o \sim^{C} o'$ then $\mathcal{V}^{C}(o) \sim^{\tau^{C}} \mathcal{V}'^{C}(o')$.

Lemma 10.1: 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 $\mathcal{I}'$. ■

Definition 10.2: Let $\mathcal{I}$, $\mathcal{I}'$ be instances of a schema S. Then we define the **bisimulation** correspondence, ${}_{\mathcal{I}}{\approx}_{\mathcal{I}'}$, 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 $o\,{}_{\mathcal{I}}{\approx}_{\mathcal{I}'}\,o'$,

■

Proposition 10.2: The relation $\approx$ is an equivalence on the set of all model 2 instances **I** of a schema S. ■

Proposition 10.3: 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 $[\mathcal{I}]_{\approx}$. ■

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

10.1 Mapping Between the two 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 definition 4.3 and the regular tree based instances of definition 9.3.

Mapping from object-identities to regular trees

Suppose $\mathcal{I} = (\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})$ is an instance of schema S. For each type τ we define the mapping $tree^{\tau}_{\mathcal{I}}$ from $[\![\tau]\!]\sigma^{\mathcal{C}}$ to model 1 values (regular trees), by:

$$\begin{aligned}
tree^{b}_{\mathcal{I}}(c^{\underline{b}}) &\equiv \langle c^{\underline{b}}\epsilon\rangle \\
tree^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\mathcal{I}}(x) &\equiv \langle \pi_{a_1} tree^{\tau_1}_{\mathcal{I}}(x(a_1)), \pi_{a_k} tree^{\tau_{a_k}}_{\mathcal{I}}(x(a_k))\rangle \\
tree^{\llparenthesis a_1:\tau_1,\ldots,a_k:\tau_k \rrparenthesis}_{\mathcal{I}}(a_i, x) &\equiv \langle ins_{a_i}\; tree^{\tau_i}_{\mathcal{I}}(x)\rangle \\
tree^{\{\tau\}}_{\mathcal{I}}(\{x_1,\ldots,x_k\}) &\equiv \langle \in tree^{\tau}_{\mathcal{I}}(x_1),\ldots,\in tree^{\tau}_{\mathcal{I}}(x_n)\rangle
\end{aligned}$$

$$\mathit{tree}^{C}_{\mathcal{I}}(o) \quad \equiv \quad \langle \mathit{St}^{C}\, \mathit{tree}^{\tau^{C}}_{\mathcal{I}}(\mathcal{V}^{C}(o))\rangle$$

The mapping *tree* from model 2 instances, **I**, to model 1 instances **M**, is given by

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

where

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

for $C \in \mathcal{C}$. It is easy to check that $\omega^{\mathcal{C}}$ is indeed a model-1 instance of $\mathcal{S}$.

Mapping from trees to object-identities

Suppose $\omega^{\mathcal{C}}$ is a model 1 instance of a schema S.

Assume we have some ordered, countably infinite set from which to pick object identifiers[3]. 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 $\mathit{inst}^{\tau}_{\omega^{\mathcal{C}}}$ from model 1 values of type τ ($\mathit{TTree}(\tau)$) to values from $[\![\tau]\!]\sigma^{\mathcal{C}}$ by

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

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

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

[3] a gumball machine

The mapping *inst* from model 1 instances **M** to equivalence classes of model 2 instances, $\mathbf{I}/\approx$ is given by

$$inst(\omega^{\mathcal{C}}) \equiv [(\sigma^{\mathcal{C}}, \mathcal{V}^{\mathcal{C}})]_{\approx}$$

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

Proposition 10.4: The mappings *tree* and *inst* provide an isomorphism between the set of bisimulation classes of model 1 instances and the $\approx$-equivalence classes of model 2 instances. ■

11 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 4.3. 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 strictly 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.

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 11.1 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 11.2 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 11.3 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 10.2, 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 12 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.

11.1 Queries and the language *SRI*(=)

The query language is described for a schema 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 two base types *unit* and *Bool*. (*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

$$\begin{aligned} \tau \quad ::= \quad & (a : \tau, \ldots, a : \tau) \mid \langle\!| a : \tau, \ldots, a : \tau |\!\rangle \mid \{\tau\} \mid \\ & \mathit{unit} \mid \mathit{Bool} \mid C \end{aligned}$$

General types

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

Definition 11.1: 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 4.1, it makes no sense to compare values of the second type since they will contain object identities taken from to distinct instances.

Definition 11.2: A **query** is a closed expression of ground type. ■

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. Consequently it will be possible to define an 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 $x^{\tau}, y^{\tau}, \ldots$. The syntax and typing rules are given in figure 12.

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

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

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 12: 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).

Example 11.1: For the schema of example 4.1, 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 again we use $e.a$ as shorthand for $\pi_a(!e)$.

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

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

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

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

■

Semantics of *SRI*(=)

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

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

$\mathbf{D}(\mathcal{I})$ by

$$V\llbracket \pi_a e \rrbracket \mathcal{I}\rho \;\equiv\; (V\llbracket e \rrbracket \mathcal{I}\rho)(a)$$

$$V\llbracket (a_1 = e_1, \ldots, a_k = e_k) \rrbracket \mathcal{I}\rho \;\equiv\; (a_1 \mapsto V\llbracket e_1 \rrbracket \mathcal{I}\rho, \ldots, a_k \mapsto V\llbracket e_k \rrbracket \mathcal{I}\rho)$$

$$V\llbracket \mathit{ins}_a e \rrbracket \mathcal{I}\rho \;\equiv\; (a, V\llbracket e \rrbracket \mathcal{I}\rho)$$

$$V\llbracket \mathit{case}\, e\; \mathit{of}\, a_1(x_1) \Rightarrow e_1, \ldots, a_k(x_k) \Rightarrow e_k \rrbracket \mathcal{I}\rho \;\equiv\; \begin{cases} V\llbracket e_1 \rrbracket \mathcal{I}(\rho[x_1 \mapsto u]) & \text{if } V\llbracket e \rrbracket \mathcal{I}\rho = (a_1, u) \\ \vdots\;\vdots & \\ V\llbracket e_k \rrbracket \mathcal{I}(\rho[x_k \mapsto u]) & \text{if } V\llbracket e \rrbracket \mathcal{I}\rho = (a_k, u) \end{cases}$$

$$V\llbracket \emptyset \rrbracket \mathcal{I}\rho \;\equiv\; \{\}$$

$$V\llbracket \mathit{add}(e_1, e_2) \rrbracket \mathcal{I}\rho \;\equiv\; \{V\llbracket e_1 \rrbracket \mathcal{I}\rho\} \cup V\llbracket e_2 \rrbracket \mathcal{I}\rho$$

$$V\llbracket \mathit{sri}(e_1, e_2, e_3) \rrbracket \mathcal{I}\rho \;\equiv\; f(u_1, f(u_2, \ldots f(u_n, v) \ldots)) \text{ where } \begin{array}{l} V\llbracket e_1 \rrbracket \mathcal{I}\rho = f \\ V\llbracket e_2 \rrbracket \mathcal{I}\rho = v \\ V\llbracket e_3 \rrbracket \mathcal{I}\rho = \{u_1, \ldots, u_n\} \end{array}$$

$$V\llbracket \lambda x \cdot e \rrbracket \mathcal{I}\rho \;\equiv\; (u \mapsto V\llbracket e \rrbracket \mathcal{I}(\rho[x \mapsto u]))$$

$$V\llbracket e_1 e_2 \rrbracket \mathcal{I}\rho \;\equiv\; (V\llbracket e_1 \rrbracket \mathcal{I}\rho)(V\llbracket e_2 \rrbracket \mathcal{I}\rho)$$

$$V\llbracket \mathit{tt} \rrbracket \mathcal{I}\rho \;\equiv\; \mathbf{T}$$

$$V\llbracket \mathit{ff} \rrbracket \mathcal{I}\rho \;\equiv\; \mathbf{F}$$

$$V\llbracket \mathit{if}(e_1, e_2, e_3) \rrbracket \mathcal{I}\rho \;\equiv\; \begin{cases} V\llbracket e_2 \rrbracket \mathcal{I}\rho & \text{if } V\llbracket e_1 \rrbracket \mathcal{I}\rho = \mathbf{T} \\ V\llbracket e_3 \rrbracket \mathcal{I}\rho & \text{otherwise} \end{cases}$$

$$V\llbracket x \rrbracket \mathcal{I}\rho \;\equiv\; \rho(x)$$

$$V\llbracket () \rrbracket \mathcal{I}\rho \;\equiv\; \emptyset$$

$$V\llbracket C \rrbracket \mathcal{I}\rho \;\equiv\; \sigma^C$$

$$V\llbracket !e \rrbracket \mathcal{I}\rho \;\equiv\; \mathcal{V}^C(V\llbracket e \rrbracket \mathcal{I}\rho) \text{ where } V\llbracket e \rrbracket \mathcal{I}\rho \in \sigma^C$$

$$V\llbracket e_1 =^C e_2 \rrbracket \mathcal{I}\rho \;\equiv\; \begin{cases} \mathbf{T} & \text{if } V\llbracket e_1 \rrbracket \mathcal{I}\rho, V\llbracket e_2 \rrbracket \mathcal{I}\rho \in \sigma^C \\ & \text{and } V\llbracket e_1 \rrbracket \mathcal{I}\rho = V\llbracket e_2 \rrbracket \mathcal{I}\rho \\ \mathbf{F} & \text{otherwise} \end{cases}$$

where $\emptyset$ denotes the unique value of type *unit*, and $\mathbf{T}$ and $\mathbf{F}$ are the values of type *Bool*.

Note that, for the semantics of *sri* 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 $V\llbracket e \rrbracket \mathcal{I}$ for the semantics of e in instance $\mathcal{I}$.

Example 11.2: For the instance described in example 4.2, and the first query of example 11.1,

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

and for the second query

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

$$= \{\text{“Philadelphia”}, \text{“Pittsburgh”}, \text{“Harrisburg”}\}$$

■

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 13.

Logical Operators

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

Predicates

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

Figure 13: Extensions to *SRI*(=)

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

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

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

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

In addition we use the shorthand notations

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

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'.

11.2 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 $\vdash e : \tau$, $V[\![e]\!]^{\tau}_{\mathcal{I}} = V[\![e]\!]^{\tau}_{\mathcal{I}'}$.

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

Theorem 11.1: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI*(=) if and only if they are isomorphic. ∎

Proof: The if part is by induction on *SRI* expressions.

For the only-if part, given an instance $\mathcal{I}$ we construct an expression $e_{\mathcal{I}}$ such that $\vdash e_{\mathcal{I}} : \mathit{Bool}$ 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 $\mathcal{I} = (\sigma^C, \mathcal{V}^C)$ is an instance of schema $\mathcal{S}$, such that

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

and

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

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

We will write $p^i[x^C_1, \ldots, x^C_{n^i}]$ for the *expression* formed by replacing each occurrence of $o_{m^i_r}$ by the variable x^C_r. Note that there is an implicit conversion of values into syntactic expressions here which can be carried out inductively in a straight forward manner.

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

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

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

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

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

Example 11.3: Returning to the instance $\mathcal{I}$ described in example 4.2, we construct $e_\mathcal{I}$ as

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

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 show 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 11.1. For example the constructors used in the proof of theorem 11.1 do not go beyond those found in the nested relational algebra of [12] or the calculus of [3] without the powerset operator. ■

Proposition 11.2: 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 value $e_{\mathcal{S}}$, depending only on $\mathcal{S}$, such that for any two instances $\mathcal{I}$ and $\mathcal{I}$, $V[\![e_{\mathcal{S}}]\!]\mathcal{I} = V[\![e_{\mathcal{S}}]\!]\mathcal{I}'$ iff $\mathcal{I}$ and $\mathcal{I}'$ are isomorphic. ■

Proof: Suppose there is such an e, and $\vdash e : \tau$. Clearly τ must be a ground type (contain no classes) for this to make sense. Hence, for any instances $\mathcal{I}$ and $\mathcal{I}'$, $[\![\tau]\!]\mathcal{I} = [\![\tau]\!]\mathcal{I}' = T$, where T is some finite set of values. So for any instance $\mathcal{I}$, $V[\![e]\!]\mathcal{I} \in T$. But T is finite and there are a countably infinite number of non-isomorphic instances (for any non-trivial schema). Hence result. ■

11.3 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 10.2, and, further, that the relation $\approx_{\mathcal{I}}$ on values of an instance $\mathcal{I}$ can be computed using *SRI*.

Indistinguishable instances in *SRI*

Proposition 11.3: Two instances, $\mathcal{I}$ and $\mathcal{I}'$, are indistinguishable in *SRI* if and only if $\mathcal{I} \approx \mathcal{I}'$. ■

Proof: It is clear that, for any instances $\mathcal{I}$ and $\mathcal{I}'$, if $\mathcal{I} \not\approx \mathcal{I}'$ then we can build an *SRI* query e such that $V[\![e]\!]\mathcal{I} \neq V[\![e]\!]\mathcal{I}'$. We will therefore concentrate on the *if* part of the proposition.

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, $V[\![e]\!]\mathcal{I}\mathcal{I} = V[\![e]\!]\mathcal{I}'$. Note that, for a *ground* type τ, $\approx^\tau$ coincides with equality. We will show that for any closed expression e, $V[\![e]\!]\mathcal{I}\mathcal{I} \approx V[\![e]\!]\mathcal{I}'$.

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. ■

Testing for correspondence in *SRI*

Proposition 11.4: Using *SRI* we can test for the bisimulation correspondence relation described in section 10, that is, for any type τ, and any values u and v, $u, v \in [\![\tau]\!]_{\mathcal{I}}$, we can form a function expression $\mathit{Cor}^{\tau} : (\tau \times \tau) \rightarrow \mathit{Bool}$ such that $V[\![\mathit{Cor}]\!]_{\mathcal{I}}(u, v) = \mathit{True}$ iff $u \approx_{\mathcal{I}} v$.

(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, 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*:

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

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 $\mathit{Check}^{\tau} : (\tau \times \tau) \rightarrow \{\{C \times C\}\}$ such that, if $\mathit{Check}^{\tau}(e, e') = X$, 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}.

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

$$\begin{aligned} & a_k(y_k) \Rightarrow \mathit{Check}^{\tau_k}(E)(x_k, y_k)) \\ \mathit{Check}^{\{\tau\}}(e, e') \;\; \equiv \;\; & \mathit{sri}(\lambda x \cdot \lambda Z \cdot \mathit{Map}(\lambda y \cdot \mathit{Map}(\lambda W \cdot \\ & \quad \mathit{UnionProd}(\mathit{Check}^{\tau}(x, y), W), Z), e'), \emptyset, e) \end{aligned}$$

The next step is a function *IterChk* : $\{C \times C\} \rightarrow \{\{C \times C\}\}$ 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 *Unfold* : $\{\{C \times C\}\} \rightarrow \{\{C \times C\}\}$ applies *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:

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

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 $\sum\{|C| \mid C \in \mathcal{C}\}$ dereferencing 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 $\mathit{sri}(f, e, u)$ is not defined if $|V[\![u]\!]_{\mathcal{I}}|$ is greater than N.

Proposition 11.5: It is not in general possible to compute the correspondence relations $\approx$ on N-bounded instances using the language SRI^N. That is, there exists a schemas $\mathcal{S}$ and type τ such that, for any expression *Cor* with $\vdash \mathit{Cor} : \tau \times \tau \rightarrow \mathit{Bool}$, either there is an N-bounded instance $\mathcal{I}$ and values u and v, $u, v \in [\![\tau]\!]\mathcal{I}$, such that $V[\![\mathit{Cor}]\!]_{\mathcal{I}}(u, v) = \mathbf{T}$ 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 $V[\![\mathit{Cor}]\!]_{\mathcal{I}}(u, v) = \mathbf{F}$ 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 dereferences of objects. Consequently it is enough to construct a schema with a recursive structure such that, for any constant k, we can construct an instance containing two objects which require $k + 1$ dereferences in order to distinguish between them. ∎

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

Applicability of results to other query languages

The results in this section are in terms of variants of a query language based on structural recursion on sets ([10]). From a theoretical standpoint the choice of the *SRI* languages is motivated by some ideas from category theory ([12]) which lead us to believe that *sri* is the most general reasonable predicate for defining functions over sets. From a more pragmatic point of view we can show that *SRI* is more expressive than other existing nested relational algebras ([12]) including the algebra with powerset operator of [3].

For most of the results in this section, in particular propositions 11.2, 11.5 and the "only if" parts of theorem 11.1 and proposition 11.3, we automatically get equivalent results for any less expressive query languages. To adapt results such as proposition 11.4 or the "if" parts of theorem 11.1 or proposition 11.3 to other query languages, it is necessary to check that the query languages in question are sufficiently expressive to describe the relevant expressions. To this end we observe that none of these results use the full generality of *SRI* or go beyond the operations we would expect to be present in a reasonable query language for these data-models.

12 Observable Properties of Object Identities with Keys

In sections 11 and 11.3 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 section 11.3 we saw that distinguishability under such a language coincided precisely with the bisimulation relation on instances, $\approx$, defined in 10.2, and in section 10.1 we saw that such an equivalence corresponded to an alternative, coarser data-model, namely that of *regular trees* defined in 9.3. In section 11.3 we saw that such a bisimulation equivalence relation, $\approx$, on values in an instance could be computed, but 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* introduced in section 4.3. A system of keys, as defined in 4.5, determines an equivalence relation on object identities:

12.1 Key correspondences

Let us recall the definition of of *key specifications* from 4.5.

Definition 12.1: Given a key specification, $\mathcal{K}^{\mathcal{C}}_{\mathbf{I}}$ and two instances $\mathcal{I}$ and $\mathcal{I}'$, we define the family of relations ${}_{\mathcal{I}\mathcal{I}'}\approx^{\tau}_{\mathcal{K}} \subseteq [\![\tau]\!]\mathcal{I} \times [\![\tau]\!]\mathcal{I}'$ to be the largest relations such that

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

2. if $x \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{(a_1:\tau_1,\ldots,a_k:\tau_k)}_{\mathcal{K}} y$ then $x(a_i) \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\tau_i}_{\mathcal{K}} y(a_i)$ for $i = 1, \ldots, k$,
3. if $(a_i, x) \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\langle\!|a_1:\tau_1,\ldots,a_k:\tau_k|\!\rangle}_{\mathcal{K}} (a_j, y)$ then $i = j$ and $x \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\tau_i}_{\mathcal{K}} y$,
4. if $X \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\{\tau\}}_{\mathcal{K}} Y$ then for each $x \in X$ there is a $y \in Y$ such that $x \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\tau}_{\mathcal{K}} y$ and for each $y \in Y$ there is an $x \in X$ such that $x \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\tau}_{\mathcal{K}} y$, and
5. for each $C \in \mathcal{C}$ and any $o \in \sigma^C$, $o' \in \sigma'^C$, if $o \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{C}_{\mathcal{K}} o'$ then $\mathcal{K}^{\mathcal{C}}_{\mathcal{I}}(o) \;{}_{\mathcal{I}\mathcal{I}'}\!\approx^{\kappa^C}_{\mathcal{K}} \mathcal{K}^{\mathcal{C}}_{\mathcal{I}'}(o')$.

■

Note: For any schema 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 $C \in \mathcal{C}$,

$$\mathcal{K}^{\mathcal{C}}_{\mathcal{I}} \equiv \mathcal{V}^C$$

then the relations $\approx_{\mathcal{K}}$ and $\approx$ relations are the same.

Given any instance $\mathcal{I}$, we write ${}_{\mathcal{I}}\!\approx^{\mathcal{C}}_{\mathcal{K}}$ for ${}_{\mathcal{I}\mathcal{I}}\!\approx^{\mathcal{C}}_{\mathcal{K}}$, or omit the $\mathcal{I}$ altogether when it is clear from the context.

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

Proposition 12.1: If $\mathcal{K}^{\mathcal{C}}$ is a key specification then, for any instance $\mathcal{I}$ and each type τ, $\approx^{\tau}_{\mathcal{K}}$ is an equivalence relation. ■

Definition 12.2: An instance $\mathcal{I}$ is said to be **consistent** with a key specification $\mathcal{K}^{\mathcal{C}}_{\mathbf{I}}$ iff for each $C \in \mathcal{C}$, any $o, o' \in \sigma^C$, if $o \approx^{C}_{\mathcal{K}} o'$ then $\mathcal{V}^C(o) \approx^{\tau^C}_{\mathcal{K}} \mathcal{V}^C(o')$. ■

Note that consistence is a more general condition than that of an instance *satisfying* a keyed schema, from definition 4.7. This is in line with the more general equivalences on values in an instance we have developed since then.

Given two instances of a simply keyed schema, $(\mathcal{S}, \mathcal{K}^{\mathcal{C}})$, say $\mathcal{I}$ and $\mathcal{I}'$, we say $\mathcal{I}$ is $\mathcal{K}$-equivalent to $\mathcal{I}'$, and write $\mathcal{I} \approx_{\mathcal{K}} \mathcal{I}'$ iff

1. For each $C \in \mathcal{C}$, each $o \in \sigma^C$ there is an $o' \in \sigma'^C$ such that $o \approx^{\mathcal{C}}_{\mathcal{K}} o'$, and for each $o' \in \sigma'^C$ there is an $o \in \sigma^C$ such that $o \approx^{\mathcal{C}}_{\mathcal{K}} o'$; and
2. For each $C \in \mathcal{C}$, $o \in \sigma^C$ and $o' \in \sigma' C$, if $o \approx^{\mathcal{C}}_{\mathcal{K}} o'$ then $\mathcal{V}^C(o) \approx^{\tau^C}_{\mathcal{K}} \mathcal{V}'^C(o')$.

12.2 Keyed Schema

Recall the definition of *keyed schema* from 4.7, that is a keyed schema consists of a schema, $\mathcal{S}$ together with a key specification $\mathcal{K}^{\mathcal{C}}$ on $\mathcal{S}$.

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

Lemma 12.2: 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 ${}_{\mathcal{I}\mathcal{I}'}\!\approx_{\mathcal{K}}$ is a consistent correspondence between $\mathcal{I}$ and $\mathcal{I}'$. ■

Proposition 12.3: 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 $\mathcal{I} \approx \mathcal{I}'$. ■

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 12.1: Let us consider the schema from example 4.1 once again, and the schema 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.

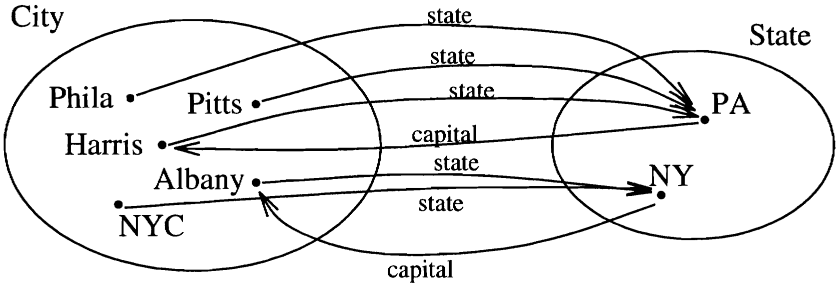


Figure 14: A database instance

Let us now construct a new instance, $\mathcal{I}'$, and compare it to the instance $\mathcal{I}$ defined in example 4.2. $\mathcal{I}'$ 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 14.

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

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

Figure 15: A summary of the operators considered and the resulting observational equivalences

12.3 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

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

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 $\approx_{\mathcal{K}}^{\tau}$ for each type τ.

2. We cannot in general find a formula to compute $\approx_{\mathcal{K}}^{\tau}$ on N-bounded values in SRI^N for any N.

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

Proposition 12.4: For any simply keyed schema $(\mathcal{S}, \mathcal{K})$ there is an M such that for any $N \geq M$, and any type τ, $\approx_{\mathcal{K}}^{\tau}$ can be computed on N-bounded values using $SRI^N(\mathcal{K})$. That is, for each type τ, there is a formula $Cor_{\mathcal{K}}^{\tau}$ of $SRI^N(\mathcal{K})$ such than $\vdash Cor_{\mathcal{K}}^{\tau} : \tau \times \tau \rightarrow Bool$ and for any two N-bounded values $u, v \in [\![\tau]\!]\mathcal{I}$, $\mathcal{V}[\![Cor_{\mathcal{K}}^{\tau}]\!]_{\mathcal{I}}(u, v) = \mathbf{T}$ iff $u \approx_{\mathcal{K}}^{\tau} v$. ■

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. Systems of *keys* generate various observational equivalences lying between these two. It follows that acyclic key specifications provide us with an efficient means of comparing recursive values which incorporate object identities, without having to examine the object identities directly. These results are summarized in figure 15.

Part III

Computing Recursive Functions with Finite Domains

13 Recursive Functions

Our type system and data-model allow us to express recursive types in such a way that the values of these types have a finite representation, and also so that the *extent* or range of values associated with a type is finite. Such recursive data structures may admit infinite series of unfoldings, and thus can be represented as trees of infinite depth.

Though finite, the representation of such data structures can be arbitrarily large, and it is not possible, a priori, to tell what level of unfolding is necessary in order to examine all parts of the data-structure. Consequently, when defining functions and operators on such recursive data structures, it is natural to use *recursive function definitions*, and so the question of how to evaluate such function definitions arises.

In this section we will show that, though such recursive function definitions have an intuitive meaning over cyclic data-structures, they will in general be undefined under the *least-fixed-point* semantics approach of conventional programming language theory. Further, if the recursive function definitions are interpreted as equations, they will admit many solutions which are not intuitively meaningful in addition to those that are. We will then demonstrate that the intuitively solutions to a system of recursive function definitions are those that can be formed *constructively*, and that this insight yields an algorithm for evaluating such functions.

In section 14 we will provide a formal semantics for recursive functions of finite domains based on the techniques of [38], and show that this semantics coincides with our intuitions for the meanings of such functions. We will also claim that the constructive solutions to any such function definitions can be computed using structural recursion Φ over the finite extents of a database, rather than general recursion.

For example let us consider the simple recursive schema shown in figure 16. Here we have a single class, *Person*, with attributes representing the name, age and set of children of a person:

$$\tau^{Person} \equiv (name : str,\ height : int,\ children : \{Person\})$$

If we were to define a function which returned the maximum height of the child of a person, we could do so by first taking the *children* attribute of a person, and then using structural

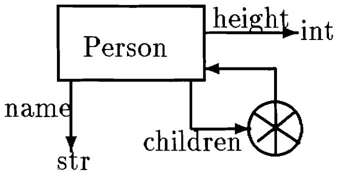


Figure 16: A simple recursive schema

recursion on this set. However, if we wanted to find the maximum height of an *ancestor* of a person, we would need to write a recursive function definition in order to unfold the nested *children* attributes:

$$max\_an\_ht(X) \equiv max\{X.height,\ max\{max\_an\_ht(Y)|Y \in X.children\}\}$$

This schema allows for cyclic data-structures: a Person could be their own ancestor. Suppose (ignoring the physical impossibility of such an occurrence) that our database contained a person, say *Jenny*, who was a member of her own children set. Then evaluating the function *max_an_ht* using a conventional least-fixed-point approach would yield a bottom or *undefined* result. But, even in such an instance, the set of ancestors of *Jenny* would be finite, and so intuitively the function should be well-defined and should yield the maximum height for any person in the set of ancestors of *Jenny*.

In general, using conventional methods from programming language theory to evaluate recursively defined functions over such cyclic structures will lead to non-terminating or undefined results, even though, intuitively, the structures are finite and so such functions have an apparent and seemingly computable solution.

In section 10 we saw that, given the knowledge that all object identities in a database resided in some known finite *extents*, and using recursion over those extents, we could compute a bisimulation relation on the values of a database, even though this relation could not be computed without using this additional knowledge. The question arises as to whether such a technique can be applied to more general recursive functions, thus allowing us to evaluate such functions using structural induction on the finite extents of the database.

In the following sections we will show how more general recursively defined functions over finite extents can be computed in a terminating manner. We will provide a denotational semantics for such function definitions, and show that the computable solutions coincide with those provided by this semantics. We will also show that the recursive function definitions may admit a number of solutions, but that only certain of these solutions are actually meaningful, and that these coincide with the solutions formed in a *constructive* manner. We will also present a semantic characterization of these meaningful solutions, and try to find an operational characterization for them.

Representing recursive data-structures

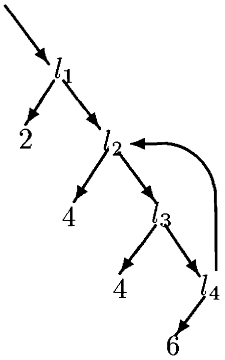
The use of object identities, or some other equivalent mechanism, allows us to create *recursive* or *self-referential* data-structures and represent them in a finite way, so that the *extent* or range of values associated with any type remains finite. This differs from the case in general programming languages, where recursive structures have potentially infinite values (such as streams).

Example 13.1: Let us consider a single class *Rlist*, and the schema

$$\mathcal{S} : \mathit{Rlist} \mapsto \mathit{int} \times \mathit{Rlist}$$

(recall that we write $\tau \times \tau'$ as an abbreviation for the record type $(\#1 : \tau, \#2 : \tau')$ and (P, Q) as an abbreviation for $(\#1 \mapsto P, \#2 \mapsto Q)$).

An example recursive list would be:



Here the corresponding instance is given by $\sigma^{RList} = \{l_1, l_2, l_3, l_4\}$, and

$$\begin{array}{rcl}
\mathcal{V}^{RList}(l_1) & \equiv & (2, l_2) \\
\mathcal{V}^{RList}(l_2) & \equiv & (4, l_3) \\
\mathcal{V}^{RList}(l_3) & \equiv & (4, l_4) \\
\mathcal{V}^{RList}(l_4) & \equiv & (6, l_2)
\end{array}$$

This could be thought of as encoding an infinite sequence, 2,4,4,6,4,4,6,4,4,6,.... Nevertheless the instance is finite, and consists of only four object identities. ■

Recursive function definitions

In general a system of recursive function definitions will be of the form

$$\begin{aligned} f_1(x) &\equiv M_1[f_1, \ldots, f_k](x) \\ \vdots & \quad \vdots \\ f_k(x) &\equiv M_k[f_1, \ldots, f_k](x) \end{aligned}$$

where M_i are some expressions involving $f_1, \ldots, f_k$ with free variable x. Details of the operations available will be deferred till later.

A simple example, for the list above would be

$$f(x) \equiv min(x.\#1, f(x.\#2))$$

which seems to define a function that returns the minimum element of a recursive list (an object of the class *Rlist* defined before).

However even with this simple example there are problems, in that it fails to define a unique solution. An obvious solution, for the instance shown earlier, would be $f(l_1) = 2$ and $f(l_2) = f(l_3) = f(l_4) = 4$. However an equally valid solution would be $f(l_1) = f(l_2) = f(l_3) = f(l_4) = -20$. So the question arises as to what makes the first solution a *better* or *more natural one*?

One possible answer is that it's the *greatest* solution to the equation. However this is a rather ad-hoc characterization: different functions with different domains and ranges are unlikely to have such a natural ordering on their range. Consequently such a characterization of best solutions would involve defining an ordering on the possible solutions, dependent on the particular function and it's intended meaning, rather than merely its domain and its range.

A more convincing answer is that the first solution is formed *constructively*, while the second seems to be merely an arbitrary choice of number. This hints at a paradigm for computing solutions to such recursive functions. To explore this more thoroughly it is best to reformulate the function definition a bit.

Unfolding the definition of *min* gives the equation:

$$f(x) \equiv \text{if } x.\#1 \leq f(x.\#2) \text{ then } x.\#1 \text{ else } f(x.\#2)$$

or, in written in a "logic-programming style":

$$\begin{aligned} f(x) = x.\#1 &\Longleftarrow x.\#1 \leq f(x.\#2) \\ f(x) = f(x.\#1) &\Longleftarrow f(x.\#2) < x.\#1 \end{aligned}$$

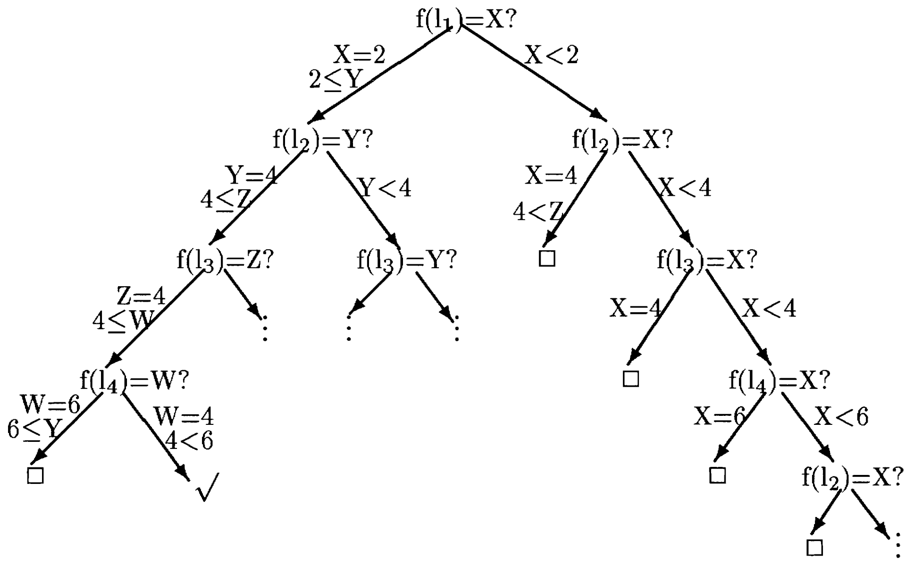


Figure 17: Search tree for $f(l_1) = X$?

and searching for a solution for f can be thought of as querying against these two clauses.

Suppose we wished to evaluate f on l_1. We would do this by trying to solve the query $f(l_1) = X?$, where X is a logical variable. Using the first clause gives the binding $X = 2$, together with the assumption $2 \leq f(l_2)$. This then leads to a new query $f(l_2) = Y?$.

Using the second clause leads to the assumptions $X < 2$ and $X = f(l2)$, and hence to a new query $f(l_2) = X?$. We can then use the two clauses to try to find solutions for each these new queries.

A search tree is shown in figure 17. At each stage in generating this search tree we generate some new assumptions, and some new queries, leading to the next stage of the tree. A branch will terminate if either the new assumptions added contradict those already made, or the branch leads to no new queries. In the later case all the logical variables in the branch would be instantiated, and all the assumptions true for those instantiations of variables, thus leading to a solution.

Alternatively a branch may not terminate at all, as is the case with the right most branch of this search tree. This is where the finiteness of the domain of our function comes in. Because there are only finitely many objects to which f can be applied it follows that a non terminating branch will eventually repeat a query. At this point we can tell that the branch will not lead to a constructive solution (in the case of figure 17 the right-most branch represents an infinite set of solutions), and stop searching it.

Example 13.2: Figure 18 illustrates a schema for a database of *Cities* and *Flights*. The class *City* has a *name* attribute and a set-valued *flights_from* attribute, which points to a set of all the flights originating from that City, while *Flights* has attributes *flt#*, *cost* and *destination*.

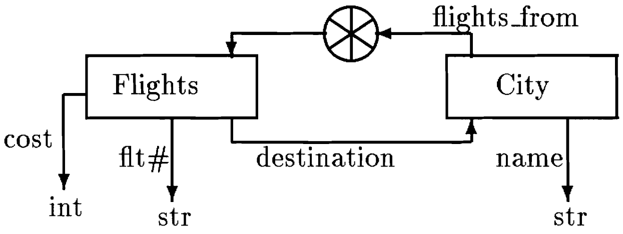


Figure 18: A schema for a database of Cities and Flights

A possible instance for this schema is illustrated in figure 19.

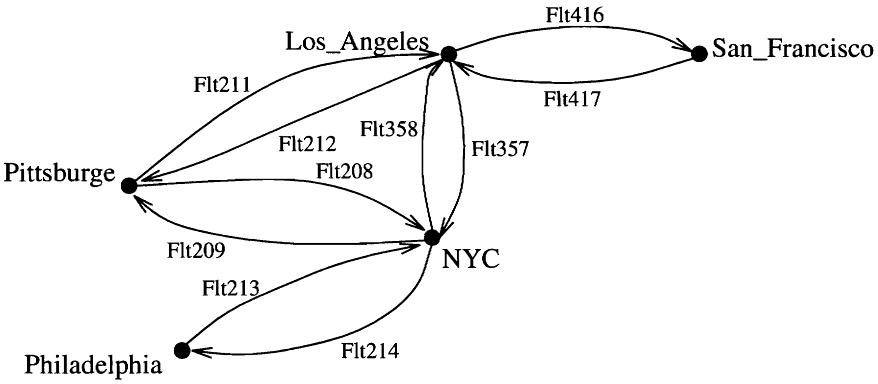


Figure 19: An instance of the Cities/Flights database

We might wish to write a function on pairs of Cities in this database which returns the lowest possible cost for flying between those cities:

$Min\_Cost(X, Y) \equiv$ if $X \doteq Y$ then 0 else

$$\min\{F.cost + Min\_Cost(F.destination, Y) \mid F \dot{\in} X.flights\_from\}$$

Such a function would not terminate if evaluated using a least-fixed-point approach, or a prolog-style depth-first search because of loops in the instance. However again it is clear that it should have a solution, and that there are only finitely many possible proofs of well-defined solutions, corresponding to loop-free paths in the instance, the paths incorporating loops can not give rise to well-defined or constructive solutions.

An equivalent function could be defined in a logic-programming style as:

$$\begin{array}{l}
Min\_Cost(X, Y) = 0 \Longleftarrow X = Y \\
Min\_Cost(X, Y) = Min\_Cost\_F(X.flights\_from, Y) \Longleftarrow X \neq Y \\
Min\_Cost\_F(W, Y) = F.cost + Min\_Cost(F.destination, Y) \Longleftarrow \\
\quad F \in X,\ (F.cost + Min\_Cost(F.destination, Y)) < Min\_Cost\_F(Del(F, W)) \\
Min\_Cost\_F(W, Y) = Min\_Cost\_F(Del(F, W), Y) \Longleftarrow \\
\quad F \in X,\ (F.cost + Min\_Cost(F.destination, Y)) \geq Min\_Cost\_F(Del(F, W))
\end{array}$$

Here $Del(P, Q)$ represents the set Q with any occurrences of the element P removed from it.

Because there are only finitely many pairs of Cities, we can construct a search tree which at each stage follows one of the flights out of the first City argument, so that each path in the search tree is guaranteed to end either in a solution or a contradictory set of assumptions, or to recursively test the same pair of Cities. ■

This last example brings out an important feature of such an algorithm for evaluating functions: it must be able to store partial instantiations while searching for a solution. For example, at some stage in the algorithm we might make an assumption:

$$\begin{array}{rcl}
Min\_Cost(Philadelphia, San\_Francisco) & = & (\mathcal{V}^{Flights}(Flt211))(cost) + \\
 & & Min\_Cost(Los\_Angeles, San\_Francisco)
\end{array}$$

but not yet have made any assumptions about what the value of $Min\_Cost(Los\_Angeles, San\_Francisco)$ should be.

Before formalizing such an algorithm however, it is necessary to provide a formal semantics for such function definitions, so that we can prove our algorithm satisfies such a semantics.

13.1 A logic for recursive functions

In this section we will extend our language of section 5 in order to form a logic, $WOL^{\mathcal{SF}}$, in which to express recursive function definitions.

Since we are interested in functions with finite domains only, we must first define a restricted form of types which have finite extents, so that these restricted types will be the domains of our functions.

Definition 13.1: We define the set of **finite types** over a set of classes $\mathcal{C}$, ranged over by $\nu, \ldots$, to be given by abstract syntax:

$$\begin{array}{rcl} \nu & ::= & \{\nu\} \\ & | & (a : \nu, \ldots, \nu) \\ & | & \langle\!| a : \nu, \ldots, a : \nu |\!\rangle \\ & | & C \\ & | & \mathit{unit} \end{array}$$

■

Note that finite types are a restriction of the types defined in 4.1 in which the only base type is *unit*. We will assume that the domain of type *unit*, $\mathbf{D}^{unit}$, is a single element set. It would be possible to allow other base types with finite domains, such as *Bool* in finite types, but these can be regarded as variants of elements of type *unit*.

Lemma 13.1: For any finite type, ν, the denotation of ν, $[\![\nu]\!]^{\mathcal{I}}$, as defined in 4.2, is a finite set. ■

We assume a set $\mathcal{F}$ of function symbols, and for each $f \in \mathcal{F}$ an associated domain, ν^f, and range τ^f, where ν^f is a finite type and τ^f is a general type.

For example, for the function *Min_Cost* described in example 13.2, the domain of *Min_Cost* would be the finite type $\nu^{Min\_Cost} \equiv (\#1 : City, \#2 : City)$ and the range would be the type $\tau^{Min\_Cost} \equiv int$.

The purpose of finite types here is simply to ensure that the domains of the functions considered are finite. Alternatively, we could have explicitly limited the values from the denotation of a type that we are interested in to being those that occur somewhere in the database instance. Our semantics for recursive function definitions should easily still work in this case.

Syntax of $WOL^{\mathcal{SF}}$

We will assume just one base type $\underline{b}$ with an infinite corresponding domain $\mathbf{D}^{\underline{b}}$. We will assume an infinite set of constant symbols $\mathbf{K}$, ranged over by $c^{\underline{b}}, \ldots$, representing values of type $\underline{b}$.

We extend the $WOL^{\mathcal{S}}$ terms defined in section 5.1 with

$$\begin{array}{rclll} P & ::= & C & \text{— class} \\ & | & c^{\underline{b}} & \text{— constant symbol} \\ & | & X & \text{— variable} \\ & | & \pi_a P & \text{— record projection} \\ & | & ins_a P & \text{— variant insertion} \\ & | & !P & \text{— dereferencing} \\ & | & f(P) & \text{— function application} \\ & | & Del(P,Q) & \text{— set deletion} \end{array}$$

The function application terms $f(P)$ here and the terms $Del(P,Q)$ are the only new terms; the rest are included as a reminder. The Del operator is necessary in order to do recursion over sets. We add the new typing rules

$$\frac{\Gamma \vdash P : C^f}{\Gamma \vdash f(P) : \tau^f} \qquad \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \{\tau\}}{\Gamma \vdash Del(P,Q) : \{\tau\}}$$

to those listed in section 5.1.

The syntax for atoms is the same as in section 5.1 except that the types of additional predicate symbols are simplified: we assume additional predicate symbols of the form p^r where r is the arity of p^r.

Semantics of $WOL^{\mathcal{SF}}$

An $\mathcal{F}$-environment, $\xi^{\mathcal{F}}$, for instance $\mathcal{I}$, is a set of functions, $\xi^f : [\![\nu^f]\!]\mathcal{I} \rightarrow [\![\tau^f]\!]\mathcal{I}$. We write $\mathrm{FEnv}^{\mathcal{F}}(\mathcal{I})$ for the set of $\mathcal{F}$-environments for $\mathcal{I}$.

Intuitively we can see that the $\mathcal{F}$-environments for $\mathcal{I}$ will be the exact solutions to our function definitions.

We alter our semantic operators for terms, $[\![\cdot]\!]_{\mathcal{I}} : Terms^{\mathcal{S}} \rightarrow Env(\mathcal{I}) \rightarrow \mathrm{FEnv}^{\mathcal{F}}(\mathcal{I}) \rightarrow \mathbf{D}(\mathcal{I})$, and for atoms, $[\![\cdot]\!]_{\mathcal{I}} : Atoms^{\mathcal{S}} \rightarrow Env(\mathcal{I}) \rightarrow \mathrm{FEnv}^{\mathcal{F}}(\mathcal{I}) \rightarrow \{\mathbf{T}, \mathbf{F}\}$ to take an $\mathcal{F}$-environment as an additional parameter. In particular we define

$$[\![f(P)]\!]^{\mathcal{I}}\rho\xi \;\equiv\; \xi^f([\![P]\!]^{\mathcal{I}}\rho\xi)$$

Also we define the semantics of the Del operator by:

$$[\![Del(P,Q)]\!]^{\mathcal{I}}\rho\xi \;\equiv\; ([\![Q]\!]^{\mathcal{I}}\rho\xi) \setminus \{[\![P]\!]^{\mathcal{I}}\rho\xi\}$$

In the following we will assume that $\mathcal{F}$ consists of a single function symbol f, in order to simplify things. However things should go through just as well with multiple function definitions.

13.2 Function definitions and programs

We will limit our attention to a special form of clauses for function definitions, A *function definition clause* is a clause of the form

$$f(x) \doteq e \;\Longleftarrow\; \Phi$$

where e does not contain any function symbols and $var(e) \cup \{x\} \subseteq var(\Phi)$.

Definition 13.2: A **program** consists of a finite set of function definition clauses, $\Delta_i \equiv (f_i(x) \doteq v_i) \Longleftarrow \Phi_i$, $i = 1, \ldots, k$, such that, for any instance $\mathcal{I}$, function symbol $f \in \mathcal{F}$, $\mathcal{F}$-environment ξ, and object-identity $o \in \sigma^{C^f}$, there is an environment ρ, and at least one clause, Δ_i such that $f_i \equiv f$, $\rho(x) = o$ and $[\![\Phi_i]\!]_{\mathcal{I}}\rho\xi$ is true. ■

The conditions are to ensure that, for each function symbol, and any computed partial solution, there will always be an applicable function definition clause, and so the program will not define partial functions. Any function definition of the form $f(x) \equiv M[f](x)$, where M may involve *if-then-else* expressions, will be equivalent to a program of this form.

Example 13.3: For the schema described in example 13.1, take the program **Pr** given by the clauses

$$\begin{aligned}
\Delta_1 &\equiv f(x) \doteq \pi_1(!x) \Longleftarrow \pi_1(!x) \mathrel{\dot{\leq}} f(\pi_2(!x)) \\
\Delta_2 &\equiv f(x) \doteq y \Longleftarrow y \doteq f(\pi_2(!x)), y \mathrel{\dot{<}} \pi_1(!x)
\end{aligned}$$

(this is the same function as described in section 13 adjusted to satisfy our requirements for function definition clauses)

Take the $\mathcal{F}$-environment ξ given by $\xi^f(l_1) = 2, \xi^f(l_2) = \xi^f(l_3) = \xi^f(l_4) = 4$. Then ξ satisfies the program **Pr**. ■

14 Topological systems of solutions for recursive function equation

We have already seen that an $\mathcal{F}$-environment represents a solution to a set of function equations, but we have also seen that a set of function equations may have many such solutions, and that, in computing the solutions to such equations, it is necessary to reason about partial solutions.

We will use the techniques of [38] to build a *topological system* to represent such partial solutions. The $\mathcal{F}$-environments will be the *points* of our topological system, and the *opens* will correspond to the partial or computed solutions. The opens will form an algebraic structure known as a *frame.* A frame, F, is a complete lattice such that F has all finite meets/conjunctions and all (possibly infinite) joins/disjunctions, and finite meets distribute over infinite joins.

Frames

We will briefly reproduce those definitions and results concerning frames that are necessary for our purposes here. For further details see [38].

Definition 14.1: A **frame** is a complete lattice, $(F, \wedge, \vee)$ such that

1. If $S \subseteq F$, S finite, then $\bigwedge S \in F$ — F has all finite conjunctions;
2. If $S \subseteq F$ then $\bigvee S \in F$ — F has all infinite disjunctions;
3. If $S \subseteq F$, $u \in F$, then $(\bigvee S) \wedge u = \bigvee\{v \wedge u \mid v \in S\}$ — finite conjunctions distribute over disjunctions.

■

One of the most important features of frames is that, despite their infinitary nature, they can be presented in the same manner as a finitary algebra.

A *presentation* of a frame consists of a set of generators G, and set of relations R (R is a set of equations relating expressions built out of the operators of the frame, the generators, G and some variables). A *model* for a presentation $(G|R)$, is a frame A, together with an interpretation g_A of each of the generators $g \in G$, such that each of the relations in R hold. A frame, A, is *presented* by a presentation $(G|R)$ iff it is a model for $(G|R)$, and for any other model for $(G|R)$, B, there is a unique frame homomorphism $f : A \to B$, such that $fg_A = g_B$ for each $g \in G$.

Proposition 14.1: Given a presentation $(G|R)$, there is a frame A, unique up to isomorphism, such that A is presented by $(G|R)$. ■

The set of elements $g_A, g \in G$ are known as the **sub-basics** of A. A **basic** of A is a finite conjunction of sub-basics.

Proposition 14.2: If A is a frame presented by $(G|R)$, then every element of A is the disjunction of some set of basics in A. ■

Definition 14.2: A **topological system** consists of a set P, a frame F, and a relation $\models \subseteq P \times F$ such that

1. $a \models \bigwedge S$ iff $a \models x$ for all $x \in S$, and
2. $z \models \bigvee S$ iff $a \models x$ for some $x \in S$.

■

Note that, if F is a frame presented by $(G|R)$ and $\models' \subseteq P \times G$ then there is a unique $\models \subseteq P \times F$ such that $(P, F, \models)$ is a topological system and $a \models g_F$ iff $a \models' g$.

14.1 A topological system of function solutions

Suppose we have a set of function symbols $\mathcal{F}$, and an instance $\mathcal{I}$. We define the *sub-basics* for $\mathcal{F}$ and $\mathcal{I}$, $\mathbf{SB}^{\mathcal{F}}(\mathcal{I})$, to be the set of triples of the form $fo \mapsto u$, where $f \in \mathcal{F}$, $o \in [\![\nu^f]\!]\mathcal{I}$ and $u \in [\![\tau^f]\!]\mathcal{I}$. Each sub-basic represents an atomic statement that one of our functions takes a particular value on a particular object-identity.

Definition 14.3: We define $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$ to be the frame generated by $\mathbf{SB}^{\mathcal{F}}(\mathcal{I})$ together with the laws

1. $(fo \mapsto u) \wedge (fo \mapsto v) = \bot$ if $u \neq v$.
2. $\bigvee\{fo \mapsto u \mid u \in [\![\tau^f]\!]\mathcal{I}\} = \top$

■

The first of these laws can be thought of as saying that every function takes at most one value on a particular object-identity. The second law states that each function symbol is total, or assigns some value to every object identity in its class.

The least element of the frame, $\bot = \bigvee \emptyset$ represents a contradictory or unsatisfiable set of solutions. The top element, $\top = \bigwedge \emptyset$ represents the set of all possible solutions.

Definition 14.4: We define the relation $\models \subseteq \mathrm{FEnv}^{\mathcal{F}}(\mathcal{I}) \times \mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$ to be such that $(\mathrm{FEnv}^{\mathcal{F}}(\mathcal{I}), \mathbf{Fr}^{\mathcal{F}}(\mathcal{I}), \models)$ is a topological system, and $\xi \models (fo \mapsto u)$ iff $\xi^f(o) = u$. ■

Example 14.1: Let us consider the schema and instance in example 13.1, and $\mathcal{F}$ containing just one function symbol f, with domain *Rlist* and range *int*.

The sub-basic $fl_1 \mapsto 2$ would describe all those $\mathcal{F}$-environments in which f takes the value 2 on the object identity l_1.

The basic $fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4$ describes exactly one $\mathcal{F}$-environment, in fact the solution to the example described in section 13.

The open u given by

$$\begin{aligned} u \quad \equiv \quad & (fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4) \\ & \vee (fl_1 \mapsto 2 \wedge fl_2 \mapsto 3 \wedge fl_3 \mapsto 3 \wedge fl_4 \mapsto 3) \\ & \vee \bigvee \{(fl_1 \mapsto i \wedge fl_2 \mapsto i \wedge fl_3 \mapsto i \wedge fl_4 \mapsto i) \mid i \leq 2\} \end{aligned}$$

describes an infinite set of $\mathcal{F}$-environments, in fact the set of all solutions to the function equations described in example 13.3. ■

14.2 Opens and Clauses

In this section we define an entailment relation between the opens (the elements of our frame $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$) and the formulas and clauses of $\mathit{WOL}^{\mathcal{SF}}$.

Definition 14.5: If ϕ is a formula of $\mathit{WOL}^{\mathcal{SF}}$, ρ is an $\mathcal{I}$-environment, and $u \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$, then we say u *entails* ϕ, ρ, and write $u \rhd \phi, \rho$ iff for any $\mathcal{F}$-environment, ξ, if $\xi \models u$, then $[\![\phi]\!]_{\mathcal{I}} \rho \xi = \mathbf{T}$.

If Φ is a set of atomic formula, then we write $u \rhd \Phi, \rho$ iff $u \rhd \phi, \rho$ for every $\phi \in \Phi$.

If $\Psi \Longleftarrow \Phi$ is a clause in $\mathit{WOL}^{\mathcal{SF}}$, and $u \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$ an open, then we write $u \rhd (\Psi \Longleftarrow \Phi)$ iff for any $\mathcal{I}$-environment, ρ, such that $\mathit{dom}(\rho) = \mathit{Var}(\Phi)$, if $u \rhd \Phi, \rho$ then there is an extension of ρ, ρ', such that $u \rhd \Psi, \rho'$. ■

Example 14.2: Recall the two clauses of example 13.3:

$$\begin{aligned} \Delta_1 \quad & \equiv \quad f(x) \doteq \pi_1(!x) \Longleftarrow \pi_1(!x) \dot{\leq} f(\pi_2(!x)) \\ \Delta_2 \quad & \equiv \quad f(x) \doteq y \Longleftarrow y \doteq f(\pi_2(!x)), y \dot{<} \pi_1(!x) \end{aligned}$$

Both of these clauses are entailed by the open u of example 14.1:

$$\begin{aligned} u \quad \equiv \quad & (fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4) \\ & \vee (fl_1 \mapsto 2 \wedge fl_2 \mapsto 3 \wedge fl_3 \mapsto 3 \wedge fl_4 \mapsto 3) \\ & \vee \bigvee \{(fl_1 \mapsto i \wedge fl_2 \mapsto i \wedge fl_3 \mapsto i \wedge fl_4 \mapsto i) \mid i \leq 2\} \end{aligned}$$

In fact this open has the important property that, for any other open v, $v \rhd \Delta_1$ and $v \rhd \Delta_2$ iff $v \leq u$. (Here we use $\leq$ to denote the partial order induced by the frame operators of $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$: that is, $v \leq u$ iff $v \vee u = u$). In other words, u is the largest open which entails both Δ_1 and Δ_2. ■

14.3 Declarative semantics of programs

In this section we will define a declarative semantics for the programs defined in section 13.2.

For an instance $\mathcal{I}$, we define the mapping $Assert_{\mathcal{I}}$ from $WOL^{\mathcal{SF}}$ clauses to $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$ by:

$$Assert_{\mathcal{I}}(\Delta) \;\equiv\; \bigvee\{u \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I}) | u \rhd \Delta\}$$

Clearly, for any *program clause* Δ, $Assert_{\mathcal{I}}(\Delta) \rhd \Delta$, and for any $u \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$, if $u \rhd \Delta$ then $u \leq Assert_{\mathcal{I}}(\Delta)$. So $Assert_{\mathcal{I}}(\Delta)$ is the greatest or most general element of the frame satisfying Δ.

Definition 14.6: Given a program $\mathbf{Pr}$, $\mathbf{Pr} = \{\Delta_1, \ldots, \Delta_n\}$, we define

$$Assert_{\mathcal{I}}(\mathbf{Pr}) \equiv Assert_{\mathcal{I}}(\Delta_1) \wedge \ldots \wedge Assert_{\mathcal{I}}(\Delta_n)$$

■

Proposition 14.3: For any program, $\mathbf{Pr} = \{\Delta_1, \ldots, \Delta_n\}$, $Assert_{\mathcal{I}}(\mathbf{Pr}) \rhd \Delta_i$ for $i = 1, \ldots, n$, and, for any $u \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$, if $u \rhd \Delta_i$ for $i = 1, \ldots, n$, then $u \leq Assert_{\mathcal{I}}(\mathbf{Pr})$. ■

Recall that any element of a frame can be expressed as a disjunction of basic elements. In particular $Assert_{\mathcal{I}}(\mathbf{Pr}) = \bigvee S$ where S is some set of finite conjunctions of sub-basics. Hence $Assert_{\mathcal{I}}(\mathbf{Pr})$ can be thought of as a disjunction of all possible solutions to the program $\mathbf{Pr}$.

Example 14.3: Take the program $\mathbf{Pr}$ defined in example 13.3. For this program

$$\begin{array}{rcl} Assert_{\mathcal{I}}(\mathbf{Pr}) & = & (fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4) \\ & & \vee(fl_1 \mapsto 2 \wedge fl_2 \mapsto 3 \wedge fl_3 \mapsto 3 \wedge fl_4 \mapsto 3) \\ & & \vee \bigvee\{(fl_1 \mapsto i \wedge fl_2 \mapsto i \wedge fl_3 \mapsto \wedge fl_4 \mapsto) \mid i \leq 2\} \end{array}$$

■

14.4 A proof-tree oriented semantics for programs

In this section we will describe an alternative view of the semantics of our programs, still using our topological system, which attempts to capture the intuitive notion of proof trees for recursive functions presented in section 13.

First we define the operator *Mod* by

$$Mod(u, \Phi, \rho) \equiv \bigvee\{v | v \leq u,\ v \rhd \Phi, \rho\}$$

So $Mod(u, \Phi, \rho)$ is the largest element of $\mathbf{Fr}^{\mathcal{I}}(\mathcal{I})$ which is less than u and entails Φ, ρ. If $Mod(u, \Phi, \rho) = \bot$ we say that u is *incompatible* with Φ, ρ.

We define the operator *Rel* to return the set of environments *relevant* to a particular open and program clause. *Rel* takes as parameters an open, u, a program clause, Δ, and an object identity, o, and returns all environments, ρ, such that u is compatible with the body of Δ and ρ, and ρ maps the first variable of Δ to o:

$$Rel(u, \Delta, o) \equiv \{\rho \in Env(\mathcal{I}) \mid Mod(u, \Phi, \rho) \neq \bot, \rho(x) = o\}$$

where $\Delta \equiv (f(x) \doteq e \Longleftarrow \Phi)$.

We define the operator $Apply_{\mathbf{Pr}}$ which takes an open, and returns a set of opens representing the effect of applying one clause to the first open.

$$\begin{aligned} Apply_{\mathbf{Pr}}(u) \quad \equiv \quad & \Big\{\bigvee\{Mod(u, \Phi_i, \rho) \wedge (f_i(\rho x) \mapsto [\![v_i]\!]_{\mathcal{I}}\rho) \mid \rho \in Rel(u, \Delta_i, o)\} \Big| \\ & \Delta_i \in \mathbf{Pr},\ o \in \sigma^{C^{f_i}},\ Rel(u, \Delta_i, o) \neq \emptyset\Big\} \end{aligned}$$

Next we define a series of sets of opens, $App^0_{\mathbf{Pr}}, App^1_{\mathbf{Pr}}, \ldots$, representing the stages of building a proof tree.

$$\begin{aligned} App^0_{\mathbf{Pr}} \quad & \equiv \quad \{\top\} \\ App^{i+1}_{\mathbf{Pr}} \quad & \equiv \quad App^i_{\mathbf{Pr}} \cup \bigcup\{Apply_{\mathbf{Pr}}(u) | u \in App^i_{\mathbf{Pr}}\} \end{aligned}$$

The following lemma may be thought of as saying that every proof tree constructed in this manner will terminate.

Lemma 14.4: For any program $\mathbf{Pr}$ and instance $\mathcal{I}$, there is an N such that $App^i_{\mathbf{Pr}} = App^N_{\mathbf{Pr}}$ for any $i \geq N$. ■

This construction represents the "largest possible" proof tree: intuitively any proof of a particular solution should be super-imposable on it.

Finally, we extract the *leaves* or finished opens from the $App^i_{\mathbf{Pr}}$s:

$$\begin{aligned} App^{\infty}_{\mathbf{Pr}} \quad \equiv \quad & \{v | v \in App^i_{\mathbf{Pr}} \text{ for some } i, \\ & \forall \Delta_j \in \mathbf{Pr} \forall o \in \sigma^{C^{f_j}} \forall \rho \in Rel(v, \Delta_j, o) \cdot Mod(u, \Phi_j, \rho) \leq (f_j o \mapsto [\![e_j]\!]_{\mathcal{I}}\rho)\} \end{aligned}$$

where $\Delta_j \equiv (f_j(x)\dot{=}e_j \Longleftarrow \Phi_j)$. So $App^{\infty}_{\mathbf{Pr}}$ consists of the opens occuring in some $App^{i}_{\mathbf{Pr}}$ for which there are no more applicable clauses.

Example 14.4: We will continue with the instance described in example 13.1 and the function definitions from example 14.3. To compute $App^{1}_{\mathbf{Pr}}$ we start with the open $\top$ and find all relevant environments. For example, for the clause Δ_1 and the object identity l_1, we get

$$\mathit{Rel}(\top, \Delta_1, l_1) = \{[x \mapsto l_1]\}$$

and for clause Δ_2,

$$\mathit{Rel}(\top, \Delta_2, l_1) = \{[x \mapsto l_1, y \mapsto n] \mid n < 2\}$$

In particular, if we take $\rho = [x \mapsto l_1]$ then

$$\mathit{Mod}(\top, \Phi_1, \rho) = \bigvee\{fl_2 \mapsto n | n \geq 2\}$$

where $\Phi_1 \equiv \pi_1(!x)\dot{\leq}f(\pi_2(!x))$. If we take $\rho = [x \mapsto l_1, y \mapsto n]$ and $\Phi_2 \equiv (y\dot{=}f(\pi_2(!x)), y\dot{<}\pi_1(!x))$, then

$$\mathit{Mod}(\top, \Phi_2, \rho) = (fl_1 \mapsto n) \wedge (fl_2 \mapsto n)$$

By calculating the relevant environments for the other object identities, we get:

$$\begin{aligned}
\mathit{Apply}_{\mathbf{Pr}}(\top) \quad = \quad & \Big\{ \bigvee\{fl_1 \mapsto 2 \wedge fl_2 \mapsto n \,|\, n \geq 2\} \\
& \bigvee\{fl_1 \mapsto n \wedge fl_2 \mapsto n \,|\, n < 2\}, \\
& \bigvee\{fl_2 \mapsto 4 \wedge fl_3 \mapsto n \,|\, n \geq 4\} \\
& \bigvee\{fl_2 \mapsto n \wedge fl_3 \mapsto n \,|\, n < 4\}, \\
& \bigvee\{fl_3 \mapsto 4 \wedge fl_4 \mapsto n \,|\, n \geq 4\} \\
& \bigvee\{fl_3 \mapsto n \wedge fl_4 \mapsto n \,|\, n < 4\}, \\
& \bigvee\{fl_4 \mapsto 6 \wedge fl_2 \mapsto n \,|\, n \geq 6\} \\
& \bigvee\{fl_4 \mapsto n \wedge fl_2 \mapsto n \,|\, n < 6\} \Big\}
\end{aligned}$$

Repeating this process, and extracting $App^{\infty}_{\mathbf{Pr}}$, we get:

$$\begin{aligned}
App^{\infty}_{\mathbf{Pr}} \quad = \quad & \Big\{ \; (fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4), \\
& \bigvee\{fl_1 \mapsto 2 \wedge fl_2 \mapsto n \wedge fl_3 \mapsto n \wedge fl_4 \mapsto n \mid 2 \geq n < 4\} \\
& \bigvee\{fl_1 \mapsto n \wedge fl_2 \mapsto n \wedge fl_3 \mapsto n \wedge fl_4 \mapsto n \mid n < 2\} \Big\}
\end{aligned}$$

■

The following proposition shows that such proof trees are meaningful with respect to our declarative semantics, and that any solution to a program is the result of such a proof tree. In effect it represents a *soundness* and *completeness* result for our proof-tree semantics.

Proposition 14.5: For any program $\mathbf{Pr}$ and instance $\mathcal{I}$, $\mathit{Assert}_{\mathcal{I}}(\mathbf{Pr}) = \bigvee \mathit{App}^{\infty}_{\mathbf{Pr}}$. ■

Corollary 14.6: For any program $\mathbf{Pr}$, instance $\mathcal{I}$, and any $\xi \in \mathrm{FEnv}^{\mathcal{F}}(\mathcal{I})$ such that $[\![\Delta]\!]_{\mathcal{I}}\xi = \mathbf{T}$ for each $\Delta \in \mathbf{Pr}$, there is a $u \in \mathit{App}^{\infty}_{\mathbf{Pr}}$ such that $\xi \models u$. ■

14.5 Constructive solutions

The semantics described in sections 14.3 and 14.4 provide us with all the solutions to a recursive function definition program. However we saw in section 13 that certain solutions are more desirable than others, and that these are the solutions found in a "constructive" manner. We would like to try and formalize this notion, and find a way to isolate such constructive solutions.

Recalling the solutions of example 14.4, our "proof-tree" semantics provided us with three different paths to solutions, corresponding to the three opens

$$\begin{aligned} u_1 &\equiv (fl_1 \mapsto 2 \wedge fl_2 \mapsto 4 \wedge fl_3 \mapsto 4 \wedge fl_4 \mapsto 4), \\ u_2 &\equiv \bigvee\{fl_1 \mapsto 2 \wedge fl_2 \mapsto n \wedge fl_3 \mapsto n \wedge fl_4 \mapsto n \mid 2 \geq n < 4\}, \text{ and} \\ u_3 &\equiv \bigvee\{fl_1 \mapsto n \wedge fl_2 \mapsto n \wedge fl_3 \mapsto n \wedge fl_4 \mapsto n \mid n < 2\} \end{aligned}$$

Here u_1 is intuitively a constructive solution, while u_2 and u_3 are not. Assume that the base type *int* denotes the natural numbers. It is notable that there is exactly one $\mathcal{F}$-environment satisfying the open u_1, while there are two satisfying u_2 and infinitely many satisfying u_3. This immediately suggests that the constructive solutions are those corresponding to exactly one $\mathcal{F}$-environment. However, we shall see that there are problems with such a characterization.

We will show that the number of $\mathcal{F}$-environments satisfying the opens u_2 and u_3 is dependent on the domains we choose for the underlying base types, whereas u_1 is satisfied by exactly one $\mathcal{F}$-environment *regardless* of the domain of the base types.

For $i = 1, 2, 3$ let us define $F_i(\mathbf{D}^{int}) \equiv \{\xi \in \mathrm{FEnv}^{\mathcal{F}}(\mathcal{I}) | \xi \models u_i\}$, where $\mathbf{D}^{int}$ is the denotation of the base type *int*. If we take $\mathbf{D}^{int}$ to be the natural numbers excluding the number 3, then F_1 and F_2 both have cardinality 1 while F_3 is infinite. If we take D^{int} to be the real numbers then F_2 and F_3 are both uncountably infinite, while F_1 has cardinality 1. The notable property of F_1, and therefore of u_1, is not that it has cardinality 1 for some particular underlying domains, but that *it is invariant* under changes of the underlying domains and the interpretations of constant symbols.

This observation is reminiscent of work done in *lambda-definability*, particularly in [30], where values which correspond to lambda terms are characterized as those which are invariant under relations on the underlying base domains. We will try to adapt these kind of ideas to our model here. This also raises the question of whether we can find a semantic characterization those functions that may be defined using recursive function definition programs of this form.

Interpretations of constants

In order to represent the notion of a particular solution being independent of the base domains and interpretations of constant symbols, we first need to parameterize our semantics on these interpretations.

Recall that we have a set of constant symbols $\mathbf{K}$, and for each auxiliary predicate symbol p^r we will assume we have a corresponding relation $p^{\mathbf{K}} \subseteq \mathbf{K}^r$.

Definition 14.7: An **interpretation** consists of a set $\mathbf{D}^{\underline{b}}$, a function $\gamma : \mathbf{K} \to \mathbf{D}^{\underline{b}}$, and for each predicate symbol p^r a relation $\overline{p} \subseteq (\mathbf{D}^{\underline{b}})^r$, such that for any $c_1, \ldots, c_r \in \mathbf{K}$, $(\gamma c_1, \ldots, \gamma c_r) \in \overline{p}$ if and only if $(c_1, \ldots, c_r) \in p^{\mathbf{K}}$. We will often write γ for an interpretation with function γ and assume the other components of the interpretation are clear from context. ■

Of particular interest is the interpretation **Id** with domain $\mathbf{D}_{\mathbf{Id}} \equiv \mathbf{K}$, $\gamma_{\mathbf{Id}}(c) \equiv c$ for $c \in \mathbf{K}$, and $\overline{p} \equiv p^{\mathbf{K}}$.

Definition 14.8: Given any two interpretations, $(\mathbf{D}_1, \gamma_1, \overline{p}_1)$ and $(\mathbf{D}_2, \gamma_2, \overline{p}_2)$ a **interpretation morphism**, $\gamma_1 \xrightarrow{g} \gamma_2$, is a function $g : \mathbf{D}_1 \to \mathbf{D}_2$, such that, for any $c \in \mathbf{K}$, $g(\gamma_1(c)) = \gamma_2(c)$[4]. ■

Semantics of $WOL^{\mathcal{SF}}$for interpretations

In this subsection we will present a variant on our model from section 4 which is parameterized on the domain of base types, and versions of our semantic operators for $WOL^{\mathcal{SF}}$ which are parameterized on interpretations.

Definition 14.9: For a base domain $\mathbf{D}^{\underline{b}}$, and a family of sets of object identities, $\sigma^C, C \in \mathcal{C}$, define $\mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}})$ by:

$$\mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}}) \equiv \bigcup\{\sigma^C | C \in \mathcal{C}\} \cup \mathbf{D}^{\underline{b}} \cup$$

[4] Interpretations and interpretation morphisms form a category with **Id** as an initial object.

$$(\mathcal{A} \overset{\sim}{\rightarrow} \mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}})) \cup (\mathcal{A} \times \mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}})) \cup \mathcal{P}_{fin}(\mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}}))$$

We define the denotation of a type τ, $\llbracket \tau \rrbracket^{\mathbf{D}^{\underline{b}}}_{\sigma^{\mathcal{C}}}$ by

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

An instance $\mathcal{I}$ consists of a family of sets of object identities, $\sigma^{\mathcal{C}}$, and for each $C \in \mathcal{C}$ a function $\mathcal{V}^C : \sigma^C \rightarrow \llbracket \tau^C \rrbracket^{\mathbf{K}}_{\sigma^{\mathcal{C}}}$.

An **environment** for an instance $\mathcal{I}$ and an interpretation γ is a partial function with finite domain $\rho : Var \overset{\sim}{\rightarrow} \mathbf{D}(\mathbf{D}^{\underline{b}}, \sigma^{\mathcal{C}})$, where $\mathbf{D}^{\underline{b}}$ is the domain of the interpretation γ.

An $\mathcal{F}$-environment, ξ, for an instance $\mathcal{I}$ and an interpretation γ, consists of a family of functions $\xi^f : \sigma^{C_f} \rightarrow \llbracket \tau^f \rrbracket^{\mathbf{D}^{\underline{b}}}_{\sigma^{\mathcal{C}}}$ for $f \in \mathcal{F}$.

We define the operators $\llbracket \cdot \rrbracket^{\gamma}_{\mathcal{I}}$ on terms and atoms as the obvious extensions of the operators already defined. ■

The definition the frame $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I})$ (definition 14.3 can be generalized to $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I}, \gamma)$ for an interpretation γ, and the topology and entailment relations (definitions 14.4 and 14.5) can also be parameterized on interpretations, as can the various operators used in the declarative and proof-tree semantics of sections 14.3 and 14.4.

If g is an interpretation morphism from γ_1 to γ_2 then we extend g to map $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I}, \gamma_1)$ to $\mathbf{Fr}^{\mathcal{F}}(\mathcal{I}, \gamma_2)$ by

1. $g(fo \mapsto v) \equiv (fo \mapsto g(v))$ for $(fo \mapsto v) \in \mathbf{SB}^{\mathcal{F}}(\mathcal{I}, \gamma_1)$, and
2. $g(u \wedge v) \equiv g(u) \wedge g(v)$, and
3. $g(\bigvee S) \equiv \bigvee \{g(u) | u \in S\}$.

Application Sequences

We define *application sequences* to represent the particular solutions to a recursive function definition: that is, the maximal paths in the proof tree.

Definition 14.10: Suppose $\mathbf{Pr}$ is a function definition program. A **application sequence** for $\mathbf{Pr}$ and an instance $\mathcal{I}$ is a sequence $(\Delta_1, o_1), \ldots, (\Delta_n, o_n)$ where $\Delta_i \equiv (f_i(x) \doteq e_1 \Longleftarrow \Phi_i) \in \mathbf{Pr}$ and $o_i \in \sigma^{C f_i}$.

An application sequence $(\Delta_1, o_1), \ldots, (\Delta_n, o_n)$ is said to be **valid** for an interpretation γ if there is a sequence of opens, $u_0, \ldots, u_n \in \mathbf{Fr}^{\mathcal{F}}(\mathcal{I}, \gamma)$ such that $u_0 = \top$, and for $i = 1, \ldots, n$, $Rel^{\gamma}(u_{i-1}, \Delta_i, o_i) \neq \emptyset$ and $u_i \in Apply^{\gamma}_{\mathbf{Pr}}(u_{i-1}, \Delta_i, o_i)$.

An application sequence S is said to be **maximal** for an interpretation γ iff S is valid for γ and there is no (Δ, o) such that $S, (\Delta, o)$ is valid for γ. ■

Lemma 14.7: If $(\Delta_1, o_1), \ldots, (\Delta_n, o_n)$ is an application sequence for $\mathbf{Pr}$ and $\mathcal{I}$ which is valid for some interpretation γ, then there exists a sequence $(\Delta_{n+1}, o_{n+1}), \ldots, (\Delta_m, o_m)$ such that $(\Delta_1, o_1), \ldots, (\Delta_m, o_m)$ is an application sequence for $\mathbf{Pr}$ and $\mathcal{I}$ which is *maximal* for γ. ■

It follows that, for any program $\mathbf{Pr}$, instance $\mathcal{I}$ and interpretation γ, there are only finitely many application sequences for $\mathbf{Pr}$ and $\mathcal{I}$ that are valid for γ. We write $AppSeq(\mathbf{Pr}, \mathcal{I}, \gamma)$ for the set of application sequences which are maximal for γ.

If $(\Delta_1, o_1), \ldots, (\Delta_n, o_n)$ is a valid application sequence for γ then we define

$$Apply^{\gamma}_{\mathbf{Pr}}((\Delta_1, o_1), \ldots, (\Delta_n, o_n)) \equiv u_n$$

where $u_0, \ldots, u_n$ is the sequence of opens such that $u_0 \equiv \top$ and $u_i = Apply^{\gamma}_{\mathbf{Pr}}(u_{i-1}, \Delta_i, o_i)$ for $i = 1, \ldots, n$.

Proposition 14.8: For any program $\mathbf{Pr}$, instance $\mathcal{I}$ and interpretation γ,

$$App^{\infty}_{\mathbf{Pr}}(\gamma) = \bigvee \{Apply^{\gamma}_{\mathbf{Pr}}(S) | S \in AppSeq(\mathbf{Pr}, \mathcal{I}, \gamma)\}$$

where $App^{\infty}_{\mathbf{Pr}}(\gamma)$ is the operator defined in section 14.4 parameterized in interpretations. ■

Constructive application sequences

Finally we define *constructive application sequences* to represent the constructive solutions to a a recursive function definition.

Suppose S is an application sequence for program $\mathbf{Pr}$ and instance $\mathcal{I}$. Then S is said to be **constructive** iff, for every interpretation γ, S is a maximal valid application sequence for γ, and for any two interpretations γ_1 and γ_2 and morphism g from γ_1 to γ_2, $g(Apply^{\gamma_1}_{\mathbf{Pr}}(S)) = Apply^{\gamma_2}_{\mathbf{Pr}}(S)$.

It remains to show that constructive application sequences do in fact capture our intuition of the constructive solutions to a recursive function definition, and that they coincide with the solutions computed by the algorithm suggested in section 13.

Part IV

The Next Steps

15 Conclusions and Further Work

The central theme of the work proposed in this document is to study the transformation and querying of databases involving recursive or self-referential data-structures. In section 1 we suggested that the most fundamental difference between databases and other areas of computation is that all values in a database arise from some *known finite extents*, and that using knowledge of these extents could yield major gains in the expressive power of systems for transforming and querying databases.

To this end we proposed a data-model in section 4 in which the type system was extended with the notion of *classes* and the self-referential data-structures are represented by means of *object identities*. Such models have been proposed previously, for example in [2], and have been used as a basis for studying the power of query languages which allow for dynamic creation of object identities. In contrast to this work, we have been concerned with the querying an manipulation of known static extents of objects.

In the remainder of this section we will recall each of the three major parts of this proposal, dealing with database transformations, the observable properties of databases involving self-referential data structures, and evaluating recursive function definitions over finite domains of self-referential data-structures. For each part we will consider various possible additions and extensions to the work which I believe to be important. This will include work that can or should be included in a completed version of this thesis, and also areas which unfortunately it may not be feasible to include in this thesis, but which may provide nevertheless important future areas for research. There will undoubtedly be other directions for extending this work, perhaps ones which could be included in this thesis, which have not occured to me, and any suggestions are welcome.

15.1 Database transformations

In part I we looked at the problem of *database transformations*. We argued that database transformations arise from a variety sources, such as schema evolutions, integrating fragments of heterogeneous databases, data entry utilities, multiple user views and so on. We also showed that there is an important interaction between database transformations and constraints, and therefore argued that it would be highly beneficial to have a single formalism in which we could express and reason about both transformations and constraints. Our collaborations with the Human Genome Center for Chromosome 22, at the University of Pennsylvania, have shown that such transformation tasks are common, and that there is a major need for tools to help automate such transformations which is not met by established software or techniques. We propose a declarative language, *WOL*, for specifying database transformations and constraints, and a system for converting non-recursive specifications of transformations in *WOL* into an underlying database programming language for implementation.

Currently implementations of *WOL* are limited to a restricted form of the language which deals with nested relational data-structures and flat relational target databases. This implementation generates code for the query language CPL which can be used to access a variety of biological data-sources. Trials of the system are underway at Human Genome Center at the University of Pennsylvania, for transforming data from GDB to the local laboratory notebook database, Chr22DB, and the formalism has been used in implementing transformations from GenBank (ASN1) data to local prolog databases.

There are many ways in which this work can be extended, and further work which needs to be addressed in the short term. Short term issues include the following:

Implementation for general recursive types: Section 6 described a method for computing the normal form of a transformation program involving general recursive types. This work represents an extension of the current implementations, and has not yet been implemented.

Further trials: There are many applications of transformations arising from the Human Genome Project databases and other biological databases, and trials to test the utility both of the current system and any future implementations for these transformation tasks are necessary. In addition I believe that the need for such transformations arises frequently in other application areas, and it would be desirable to test these implementations in other application areas.

For the first of these, there are a number issues that need to be resolved. Firstly, assuming we continue to use CPL as a target language for implementing transformations, we need to decide how best to extend it in order to represent some form of references or self-referential data-structures. Secondly, we need to decide to what extent we wish to reuse or extend existing implementation code, and to what extent we are willing to write the new implementation from scratch. In section 7 we indicated that there could be substantial improvements to the efficiency of the normalization process gained by adopting a new internal representations for clauses.

There are a number of potential sources of transformation trials in the Human Genome Center at the University of Pennsylvania, for example in implementing a planned reorganization of the local laboratory notebook database, Chr22DB, or in creating data-entry applications. In some existing work using the *WOL* formalism to specify transformations between the ASN1 GenBank database and a local Prolog database, variants in the ASN1 database proved problematic. I would like to rework these transformations since I believe they might be handled more effectively by composing two *WOL* transformations and using intermediate data-structures for variants.

Possible longer term directions for research directions and extensions to this work include:

Interactive schema manipulation tools: The interfaces to the transformation tools described thus far are largely syntactic. They require the user to input transformation specifications in a high level language (*WOL*). It would be desirable to have more user-friendly, graphically oriented tools for carrying out schema manipulations and generating *WOL* transformation specifications.

Automated derivation of constraints: In section 6 we argued that there is an important interplay between database transformations and constraints. It would be desirable to automatically derive certain constraints from a transformation specification, for example to derive a set of constraints on a source database which are necessary for the transformation to be valid.

Implementing user views: The tools considered so far are oriented towards doing bulk one-time transformations on data. User views are also, in a sense, transformations, though they are generally implemented in a dynamic manner, transforming data as it is needed whenever the view is accessed. We would like to build tools for specifying and implementing user views using *WOL*.

Translators for other languages: In certain situations it may be desirable to convert transformations specified in *WOL* into other database programming languages, rather than use CPL as an intermediary language. It may therefore be desirable to implement translators from *WOL* normal forms into a variety of DBPLs.

Weakly Context-Sensitive Grammars as data sources: There is a great deal of data which is stored not in databases but in formatted text files. We would like to extend *WOL* so that we can use context-free or weakly context sensitive grammars to parse such files and then transform the data into some more convenient format for querying and manipulation.

The last of these items is particularly interesting. We were inspired by the work of Abiteboul, Cluet and Milo ([4]) in which they used context free grammars (CFGs) to specify the structure of a text file, and then queried against these CFGs. In this work they pushed queries down the parse tree of a text file in order to evaluate them. We believe it may often be more useful to transform data from a text file into some database format, where general queries and database technology can be applied. Also we believe we can extend *WOL* to transform text files specified by mildly context sensitive grammars (MCSGs), of which indexed grammars are an example. MCSGs are more general than CFGs and may be useful for analysing data which can not be recognized using a CF, for example certain features of biological sequence data. These issues are currently being investigated by Wenfei Fan at the University of Pennsylvania.

15.2 Observable properties of recursive data-structures

In part II we undertook a more detailed examination of our data-model, and, in particular, examined the observable properties of the model in the presence of various different kinds of operators on object identities. It is essential to understand these issues properly if we are to properly understand and reason about the expressive power of the model and its query languages.

We argued that the notion of object identity was an abstract one, and that it was natural to consider only values not involving object identities to be directly observable and to restrict the primitives available for comparing object identities. In section 11 we showed that, if we allow an equality predicate on object identities in a reasonably expressive query language, then observational indistinguishabilty of instances coincides with isomorphism.

In section 9 we proposed an alternative data-model based on *regular trees* (a "value based" based model in the terminology of [2]). We showed that this model exactly captures the observable properties of the model of section 4 under the assumption that no comparison operators are available on identities. In section 10 we showed that observational equivalence of values in an instance, under these assumptions, corresponds to a bisimulation relation on values. We also showed that such a bisimulation relation can be computed in a query language which allows for recursion over the entire extents of a database, but can not be computed in time dependent only on the size of the values being compared. This indicated that the ability to recurse over the finite extents of a database could potentially give a major gain in expressive power of queries, an idea that was explored further in part III. It also showed that computing bisimulation was not a practical means of comparing object identities.

In section 12 we argued that acyclic systems of keys give us a practical mechanism for comparing and referencing values without resorting to the intuitively dubious practice of providing a direct equality predicate on object identities. We showed that a using system of keys allows for a range of observational equivalence relations lying between isomorphism and bisimulation, and that observational equivalence under an acyclic system of keys can be computed in time dependent on the size and type of the values being compared, and not the size of the entire database.

This part of the thesis is, I believe, complete as it stands, and there are no significant additions that I wish to make to it at present.

15.3 Recursive Functions with Finite Domains

In part III we observed that our data-model allows for recursive type definitions through classes, and admits arbitrarily nested or cyclic data-structures as instances of such types. We argued that it is natural to use *recursive function definitions* in order to express queries on such data-structures. In section 13 we showed that, intuitively, such recursive function definitions have meaning because of the finite nature of database instances, but that a conventional *least-fixed-point* approach to providing a semantics for such function definitions would in general give undefined results. We also showed that a recursive function definition would, in general, admit many solutions in addition to the intuitively meaningful solutions, and argued that the meaningful solutions are those that could be formed *constructively.*

In section 14 we provided two semantics for recursive function definitions, both in terms of a topological system of partial solutions. The first semantics is declarative in nature, while the second is more operational, and corresponds to the proof trees suggested in section 13. We showed that these two semantics coincide, indicating that the solutions to such recursive function definitions, when interpreted declaratively, coincide exactly with the valid paths in the proof trees for those definitions. We also gave a denotational characterization of the "constructive" solutions to such function definitions.

Part III is, I believe, the furthest part of this proposal from being completed. The most urgent requirement is for a good operational characterization of the "constructive" solutions. Such a characterization would have to be used to prove that the semantic characterization of constructive solutions of section 14.5 is correct, and that the algorithm suggested in section 13 does indeed return all the constructive solutions for recursive function definition.

In section 13 we also conjectured that the constructive solutions to any recursive function definitions could be computed by means of structural recursion over the finite extents of a database, rather than by general, unbounded depth recursion. In fact I have some results showing that this is the case. However I believe that the validity of these results is dependent on first finding a satisfactory operational characterization of constructive solutions, and have therefore omitted them for the time being.

The minimum work that I feel is necessary in order to complete part III is therefore the following:

- Construct an operational characterization of "constructive" solutions to recursive function definitions.

- Prove that constructive solutions may be evaluated using only structural recursion over the finite domains of the functions.

- Generalize these results for set valued functions: our *function definition clauses* defined in section 13.2 require that the complete value of a function on some element be given in the head of a clause. In order to deal with set valued functions, we would also like to allow for set-inclusion atoms in the head of a function definition clause.

I believe that the last of these items can be addressed by enriching the frame defined in section 14.2 using a Hoare (lower power domain) partial ordering on sets. Such an extension is not in itself difficult. However care is required to check that the results of section 14 still go through in such an extended semantics. Also, there may be interesting correspondences between such a semantics and the inflationary semantics for Datalog with recursion.

In the longer term, work that could be done to extend part III would include trial implementations based on these ideas, and an examination of the mechanisms necessary to convert function definitions given in a conventional functional programming style into the Horn clauses described in section 13.1.

References

[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, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proceedings of 19th International Conference on Very Large Databases*, pages 73–84, Dublin, Ireland, 1993.

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

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

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

[8] W.C. Barker, D.G. George, L.T. Hunt, and J.S. Garavelli. The PIR protein sequence database. *Nucleic Acids Research*, 19:2231–2236, 1991.

[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] Luca Cardelli. Types for data-oriented languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *LNCS 303: Advances in Database Technology — International Conference on Extending Database Technology, Venice, Italy, March 1988*. Springer-Verlag, 1988.

[14] Michael J. Cinkosky, Jim Fickett, Debra Nelson, and Thomas G. Marr. The restructuring of GenBank, October 1987.

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

[16] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[17] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. In *Proc. Third International Conference on Information and Knowledge Management (CIKM)*, pages 423–432, December 1993.

[18] Department of Energy. *DOE Informatics Summit Meeting Report*, April 1993. Available via gopher at `gopher.gdb.org`.

[19] K. Hart, D. B. Searls, and G. C. Overton. SORTEZ: A relational translator for NCBI's ASN.1 database. *Computer Applications in the Biosciences*, ???(???):???, ??? To appear. See also UPenn Technical Report CBIL-9203.

[20] 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.

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

[22] Anthony S. Kosky. Modeling and merging database schemas. Technical Report MS-CIS-91-65/L&C 39, Department of Computer and Information Science, University of Pennsylvania, 1991.

[23] 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.

[24] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19, 1994.

[25] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[26] A. Motro and P. Buneman. Constructing superviews. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1981.

[27] G. Christian Overton, Jeffery Aaronson, Jurgen Haas, and Julie Adams. QGB: A system for querying sequence database fields and features. *Computational Biology*, 1994. To appear.

[28] P.L. Pearson. The genome data base (GDB), a human genome mapping repository. *Nucleic Acids Research*, 19:2237–2239, 1991.

[29] William R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Proc. Natl. Acad. Sci. U.S.A.*, 85:2444–2448, 1990.

[30] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In J.P̃. Seldin and J.R̃. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[31] J. Reynolds. Three approaches to type structure. In *Advanced Seminar on the Role of Semantics in Software Development*, Berlin, 1985.

[32] John F. Roddick. Schema evolution in database systems — An annotated bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.

[33] A. Sheth, J. Larson, J. Cornellio, and S. Navethe. A tool for integrating conceptual schemas and user views. In *Proceedings of 4th International Conference on Data Engineering*, pages 176–183, 1988.

[34] J. Smith, P. Bernstein, U. Dayal, N. Goodman, T. Landers, K. Lin, and E. Wong. Multibase — Integrating heterogeneous distributed database systems. In *Proceedings of AFIPS*, pages 487–499, 1981.

[35] Ernest Szeto and Victor M. Markowitz. Erdraw 4.0: A graphical editor for extended entity-relationship schemas. reference manual. Technical Report LBL–PUB–3084, Lawrence Berkeley Laboritory, Berkeley, California, 1993.

[36] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems I.* Computer Science Press, Rockville, MD 20850, 1989.

[37] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[38] Steven Vickers. *Topology via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[39] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[40] 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.

[41] 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.