Implementing Attribute Metadata Operators to Support Semistructured Data

Fan Guo

School of Computer Science, McGill University
Montréal, Québec, Canada

January 2005

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for
the degree of Master of Science

T. H. Merrett, Advisor

Contents

List of Figures

List of Tables

Abstract

This thesis documents the design and implementation of nine new features for supporting semistructured data in jRelix, including the augmentation of attribute metadata, the extension of the wildcard, and the expansion of recursive virtual nested relations.

The strategy for implementation of semistructured data in Aldat system is to augment the data type and operations in the programming language, jRelix. The functionalities of attribute metadata operators (**eval**, **quote**, **transpose**) have been enhanced and new operators (**relation**, **typeof**, **self**) have been added. The capacity of the wildcard has been magnified; therefore, it can represent top-level relations or nested relations in domain algebra and relational algebra depending on the context. The expansion of recursive virtual nested relations has been implemented to support recursive nesting structure. Applications of these operators also have been presented including attribute path and schema discovery, data reorganization, and queries involving transitive closure on graphical structured data.

Résumé

Ce mémoire documente l'élaboration et l'implémentation de neuves nouvelles caractéristiques supportant les données semi-structurées en jRelix, incluant l'augmentation des métadonnées attribut, l'extension du joker("wildcard") et l'extension des relations imbriquées virtuelles récursives.

La stratégie pour l'implémentation de données semi-structurées dans le système Aldat consiste en l'augmentation du type de données et des opérations dans le langage de programmation jRelix. Les fonctions des opérateurs de métadonnées attribut (**eval**, **quote**, **transpose**) ont été améliorées et de nouveaux opérateurs (**relation**, **typeof**, **self**) ont été ajoutés. La capacité du joker a été amplifiée et il peut dorénavant, selon le contexte, représenter les relations du niveau supérieur ou les relations imbriquées dans le domaine à la fois de l'algèbre et de l'algèbre relationnelle. L'extension des relations imbriquées virtuelles récursives a été implémentatée afin de supporter la structure imbriquées récursives. Les utilisations de ces opérateurs décrites dans ce mémoire sont : le chemin d'accès attribut, la découverte de la schéma, la réorganisation des données et les questions impliquant les fermetures transitives des données graphiques structurées.

Acknowledgments

The most special thanks are due to my supervisor Professor Tim Merrett for his invaluable advice, patient guidance, and constant encouragement throughout the development of this study. I have benefited enormously not only from his valuable insights, mentoring, and motivating, but also from his expertise as well as erudition. I feel very fortunate to conduct this thesis under his supervision. I would particularly like to thank him for his generous financial support.

I am grateful to all of my colleagues in the Aldat lab. Special thanks goes to Zhongyan Wang who provided consultation and great help in my understanding of the Aldat system and the lab-working environment. Yu Gu kindly supplied me with her related code so that I had an environment to integrate and test my program.

I wish to thank the School of Computer Science for the graduate courses and the research environment. I acknowledge all the secretaries and system staff for their administrative help and technical assistance.

Many thanks go to Brenda Anderson who proofread this thesis and Christophe Chénier who translated the abstract into French.

Last but not least, I would like to express my appreciation to my husband, Hong Xu, for his love, support, and encouragement during the time of my study.

Chapter 1

Introduction

This thesis documents the enhancement of attribute metadata and the extension of the wildcard, ".", in regular expression and domain algebra in jRelix for supporting semistructured data. We augment the functionalities of the existing attribute metadata and develop three new operators. In this chapter, we will first present the motivation of our work in Section 1.1 and then introduce the outline of the thesis in Section 1.2.

1.1 Motivation

Many applications today contain semistructured data. Well known examples are those data that are published on the World-Wide Web: journal papers, technical documents, tutorials, news articles, etc. Other examples are scientific research databases, or commercial databases. Most of these different kinds of data have irregular data structures, partially defined schemas, or the structure of the data changes over time.

The emerging of semistructured data and its popular applications posed new requirements for the database research area. New database systems for managing semistructured data have been developed and the relational database system has been combined with systems which manage semistructured data. Our work is to integrate the management of semistructured data into the existing jRelix database programming language.

The relational data model introduced by E.F. Codd [Cod70] has been successfully applied to a considerable number of applications. It has become the core technique of most database management systems and is particularly suitable for business database management where the structure of the data is relatively simple and can be represented in table form. The relational database system provides a simple and flexible way for application development and maintenance.

The database query language is one type of the languages to manipulate the structure and data in a relational database. The purpose of a query language is to provide both programmers and non-programmers with a tool to retrieve information from databases in an easy way. Users focus only on the content of the information to be retrieved. The implementation of a database management system with such language needs to translate the user queries into logic query plans containing the operators and functions that can be executed in the data engine. System query optimization is thus a critical issue under this paradigm. SQL (Structural Query Language) is a standard database query language for the relational database system and has been used in many commercial systems.

Relational query languages have limited expressional power in dealing with complex data. As a result, they do not suit non-business applications involving numerical computation or complicated manipulation of data [AB87]. Extensions have been made to meet the requirements of the continuing emergence of new applications. We describe them in the next three paragraphs. They include the extension of data types, the extension of structure from flat relations to nested relations, and the augmentation of the query language.

Originally, only generally used data types for numbers and strings are available in a database system. Along with new applications, a database system needs to support new data types for text, images, abstract data types [SRG83, OFS84, OH86, Sto86, Zhe02], and types that have more than one value (UNIVERSAL [Mer01, Roz02] and UNION [ACC$^+$97, Mer03, Gu05]).

To make nested relations is to allow the attributes of a relation to be able to hold relational data instead of only simple (or scalar) data [Mak77, FG85, TF86, JS82].

In query languages such as SQL, operations have been extended from basic relational algebra to more powerful operations such as aggregates, ordering, grouping, updates, and built-in functions; simple query form "Select-From-Where" has been extended to nested structures so that nested relations can be accessed.

Another direction for developing database management system languages is the design and implementation of database programming languages. The reason for database programming language is the need for an integrated programming environment for database systems to uniform the specific database systems developed for particular applications. jRelix , developed at the Aldat Lab in the School of Computer Science at McGill University, is a high-level database programming language that subsumes the database query language. It is designed and implemented based on the relational algebra [Cod70] and domain algebra [Mer76, Mer84], and provides both the database query language and mechanisms to application solutions in many fields, such as expert system, numerical computation, and data mining.

The semistructured data management system has been an important research and development field in the past decades. In addition to application oriented management systems for individual applications, such as AceDB and XML, research has focused on the development of general purpose semistructured database systems. These systems include Lore [MAG$^+$97] and its query language Lorel [AQM$^+$97], and Lore's contemporaneous system UnQL [BDS95, BDHS96, BFS00]. Both adopt a direct graph data model, and were designed and implemented particularly for semistructured data. The query languages are SQL-style combined with simple path expressions and general path expressions. UnQL also provides mechanisms for restructuring a database graph while traversing it. The architecture of these systems is more or less the same as that of a traditional relational database system except that the semistructured query needs first to be transformed into a SQL-like query.

Another strategy for managing semistructured data is to bind the data model and the query language to an existing relational database system. Many database systems with SQL-based query language, such as Oracle, adopt this tactic. Under this paradigm, the database system has all the features for processing the relational data model. Furthermore, new data types are added to hold semistructured data. A second query language accomplishes manipulation of the new types of data. Data is moved between two languages to fulfill the data exchange inside the system. Users must be bilingual who can use both query languages.

At the Aldat database system, the management of semistructured data has been integrated into jRelix and uniformly managed with other relations [Mer03]. The existing jRelix already has extensive ability in manipulating semistructured data. In order to provide a full support for semistructured data, a new data type, TYPE, and new operators to manipulate attributes are introduced. The purpose of this thesis is to augment attribute metadata operators into jRelix to support queries associated with the structure of relations.

Note that the semistructured data in jRelix is a general purpose data form. Its syntax is a "mark-up" representation called xML that has the characteristics of all the markup notations originated from GML [Mer03]. Thus in xML, x represents G, HT, SG, and X in GML, HTML, SGML, and XML, respectively.

1.2 Outline

This chapter introduces the motivation of this thesis. Chapter 2 reviews the literature on the relational data model, nested relations, database query languages and database programming languages, semistructured data and its management systems, and the strategy to integrate semistructured data management into jRelix. Chapter 3 introduces the use of the jRelix system and presents the power of jRelix as a database programming language with examples and applications. Chapter 4 is the user manual of new functionalities of attribute metadata operations. Chapter 5 exhibits the solutions to some classical queries posed on semistructured data using new attribute metadata. Chapter 6 describes the implementation of the attribute metadata. Chapter 7 concludes the thesis with a summary and proposals for future work.

Chapter 2

Background and Related Work

Semistructured data is becoming more and more popular especially with the huge amount of data published on the World Wide Web. Different from data that was processed in the traditional relational database, semistructured data does not always have predefined and fixed data structures or schema. The traditional database system alone cannot manage and retrieve semistructured data. In this chapter, the general strategy to manage semistructured data in the Aldat system will be presented, preceded by the literature review of the relational data model and the nested relation, the features of semistructured data, and the database query language and programming language.

2.1 Relational Model

E.F.Codd signaled the start of a new era of the database management system by proposing the relational model in "A Relational Model for Large Shared Data Banks" [Cod70] published in 1970. Before that time, database products were usually production oriented, complex, and difficult to maintain and enhance. By following the abstract data model, database products began to become more flexible and easier to maintain [CCS94]. The relational model has influenced the database products in different fields of applications thereafter. From the early prototype system Ingres [HSW75] and System R [ABC$^+$76] to nowadays most widely used products in large multi-user environments including Oracle, Sybase, Ingres and DB2, the relational data model acts as their foundation and principle.

The relational data model is described as independent of the physical representation of data. In this model, data is thought of as two-dimensional tables, known as **relation**. Columns of the table are labeled and the names of the column are called attributes. The values of the attributes are called domains. A row in the table is called a tuple. The model has the following properties:
1) All tuples are different from each other
2) The order of tuples is not important
3) Each attribute has a different name, therefore, the order of attributes does not matter
4) The value in each tuple under an attribute is "simple"

A relation that satisfies the above properties also is known as a "flat relation". "Simple" in the fourth property basically means that the value of an attribute should be of basic types such as STRING, BOOLEAN, INTEGER, etc. This excludes any possibility that the value of an attribute is a relation.

2.2 Nested Relation

By removing the restrictions in flat relation, which is "the value of an attribute in a tuple should be atomic", a relation may be designed to be a nested relation which might have relations as values of an attribute. The possibility for a relation to have non-atomic value was brought up along with the relational data model by Codd [Cod70] and the idea of nested relation was first proposed formally [Mak77] to satisfy the need of non-business database applications, such as hierarchical databases. It is more natural to model complex data like this in a nested relation. Fisher and Gucht [FG85] discussed the one-level nested relation which later was generalized by Thomas and Fischer [TF86] to an arbitrary but fixed depth. Jaeschke and Schek[JS82] added two operators to Makinouchi's [Mak77] model: **nest** and **unnest**, to do transformations between flat relations and nested relations.

2.3 Aldat and jRelix

Aldat, which stands for the **al**gebraic approach to **dat**a, is a project under development at McGill. Aldat aims to integrate general purpose programming language into database query language so that the language has the power to deal with different applications. The foundation of the approach is the generalization of relational algebra that provides operations over the relational model database, and the created domain algebra that empowers the query language with arithmetic, logic, string processing, and much more.

jRelix, standing for a **re**lational database programming **l**anguage in Un**ix**, written in **J**ava, is the query and programming language in Aldat. Relational algebra and Domain algebra for flat relation and nested relation are implemented [Hao98, Yua98, Kan01, Cha02, Sun00]. Computation [Bak98] and ADT (Abstract Data Type) [Zhe02] provide mechanisms for procedural abstraction and data abstraction, respectively. Distributed data processing in jRelix [Wan02] empowers jRelix with the ability to process data across the Internet. Attribute metadata [Roz02] was implemented to support operations over attributes.

For more details on jRelix, please refer to Chapter 3.

2.4 Semistructured Data

2.4.1 Semistructured Data and its Features

While the traditional database system and query language play important roles in different fields, new applications have emerged and posed new requests to database systems. Semistructured data which is available in many applications is one of them. Documents on the Web and scientific research data are two representatives. Semistructured data does not conform to the traditional relational data model. Compared with data under the relational data model, which has predefined fixed structures, semistructured data is not well structured. The data schema, which describes data types and attributes, is implicitly specified together with the data. Therefore, semistructured data is self-describing.

Semistructured data has the following features [Abi97]:

1) the structure is irregular: some of the data has missing attributes; some attributes have different types of data; some attributes may have more than one value; in data integration from an heterogeneous source, semantically related data may be represented in different ways.
2) the data structure is implicit
3) the data structure is incomplete
4) data type is not strictly defined
5) the structure of data may be much larger than the data itself
6) the schema is dynamic and develops very fast
7) schema may be changed and queried as data

2.4.2 Semistructured Data Examples

Semistructured data has many applications and for each application there is a particular database system which includes the data model and its query language. In the following subsections, some examples of semistructured data and how these data are accessed are investigated.

BibTex

BibTex [Jac96] is a program and file format designed for LaTex to create bibliographies. A BibTex file contains many entries, each of which consists of standard entries with field names and one or more values describing one aspect of the entry. The standard entries have required fields and optional fields and vary according to the different types of entries (a BibTex file may have as many as 14 types entries). Optional fields can be omitted without causing any format problem, but the required fields are necessary and should not be omitted. Figure 2.1 and Figure 2.2 show schemas for two different categories of BibTex file, one is for book type, the other is for conference paper type. Fields in square brackets "[" and "]" are optional. The field *citation_key* is the label used in the citation in a LaTex document. Running the BibTex program with the BibTex database, all entries for cited documents will be extracted from the database and reformatted into the specified style. The newly-formed data is stored in a bibliographic file with each entity having a label which corresponds to the one in the citation. A bibliographic list can be printed out using corresponding commands in LaTex file with the specified style. As long as users create a valid bibliographic database and cite correctly in the text, the access of each entry of the database is accomplished automatically by the BibTex program and Latex. Users do not have to worry about it.

Figure 2.3 exhibits example entries for book and conference. Note that both examples omit some of the optional entries. Therefore, in a BibTex database, every entry may have a different data structure.

```
@book{citation_key,
      author/editor, title, publisher, year
      [,volume, series, address, edition, month, note, key ]
}
```

Figure 2.1: BibTex Schema for “book”

```
@conference{citation_key,
      author, title, booktitle, year
      [, editor, pages, organization, publisher, address, month, note, key]
}
```

Figure 2.2: BibTex Schema for “conference”

```
@conference{Merrett88,
    author={T. H. Merrett},
    title={Experience with the Domain Algebra},
    booktitle={Proceedings of the Third International Conference
        on Data and Knowledge Bases: Improving Usability
        and Responsiveness},
    year={1988},
    month={June},
    editor={Catriel Beeri, Joachim W. Schmidt, Umeshwar Dayal},
    pages={335-346}
}

@book{Merrett84,
    author={T.H. Merrett},
    title={Relational Information Systems},
    publisher={Reston Publishing Co.},
    year={1984},
    address={Reston, VA}
}
```

Figure 2.3: Two BibTex File Entries

HTML

HTML (**H**yper **T**ext **M**arkup **L**anguage) [RH98] is a language used to create documents that can be displayed as a web page using a web browser. HTML uses tags to structure text. The tags specify the display format of the page and most of them appear in pairs. A HTML file consists of elements that are enclosed in start tags and end tags. In between the tags are content of the elements (e.g. Semistructured data < /b> or <title> Information System < /title>). The tags are standard but not all of them may appear in every page. A hyperlink is used to link the current file to another file (another HTML file, an external image, a source file, etc.). Anchor tags ($< a >$ and $< /a >$) specify the destination URL address and the corresponding keyword in the current document (e.g. <a href=“http://www.cs.mcgill.ca” > McGill University $< /a >$). Navigations from the current page to the destination file are performed by clicking the highlighted keyword in the current page. Table 2.1 presents some of the frequently used tags. Figure 2.4 gives an example of an HTML file, which includes a small paragraph, a hyperlink and a table.

start and end tags	function
<html> ... </html>	define an HTML document
<title> ... </title>	define an title of document
<body>... </body>	define an HTML document body
...	the largest header
...	the smallest header
<pre> ... </pre>	pre-formatted text
 ... 	bold font
 	enforce a line break
...	hyperlink to "URL"

Table 2.1: HTML tags

Many query languages were proposed for web searching, such as WWWW [McB94], Lycos [ML94], W3QL [KS95] , WebSQL [MMM97], and WebLog [LSS96]. They either query information from inside an HTML file or retrieve HTML documents from the web.

WebSQL, on the other hand, is a SQL-like query language for the Web. HTML is the basic query unit in the language and the information available in HTML (i.e., URL, title, text, type, length, and the last modification date) are query attributes. WebQL uses the virtual relation “`Document[url, title, text, type, length, modif]`” to query the HTML document content (the value of the attributes).

Queries related to hyperlink structures of the Web are performed based on another virtual relation “`Anchor[base, href, label]`” together with different links that point to documents inside the current file, documents in the same site, and documents in a

```
<html>
<title> Students Records </title>
<body>
<p> The following table records the information of students who take
the course <a href=" http://www.cs.mcgill.ca/~cs612/homepage.html">
Database System</a> .
</p>

<table border="1">
<caption>Students Records</caption>
<tr>
  <th>id</th>   <th>name</th>   <th>email</th>
</tr>
<tr>
  <td>1325</td>  <td>Joe</td>  <td>Joe@cs.mcgill.ca</td>
</tr>
<tr>
  <td>1386</td>  <td>Ted </td>  <td>Ted@cs.mcgill.ca</td>
</tr>
<tr>
  <td> ...</td>  <td> ...</td>  <td> ...</td>
</tr>
</table>

</body>
</html>
```

Figure 2.4: An HTML File

remote site. In WebSQL, internal structure of an HTML is not considered, therefore, even if the structure is known, users cannot benefit from it.

WebLog is a declarative language for querying and restructuring HTML documents [LSS96]. WebLog uses three types of predicates based on the internal structure of an HTML file for retrieving information. The built-in predicates are used to express general relationships (e.g. substring(string, string)); the programming predicates are used to express predicates which only exist in the context of a program (e.g. answer(Y)); predicate definitions are used to compose mappings from attributes to values. Attributes include tags and two special strings, "occurs" and "hlink". Values are strings and hyperlinks in HTML files. Tags and string "occurs" are mapped to string values (e.g. title $\rightarrow$ 'Students Records') and string "hlink" is mapped to a hyperlink (e.g. hlink $\twoheadrightarrow$ L, here L is a variable). Information is retrieved dynamically by navigating through hyperlinks. In WebLog, an arbitrary navigation is performed via the *traverse*(L) function together with a recursive query structure which terminates when the specified condition is satisfied and usually the condition is a string or text that is expected. The *traverse* function asserts the fact that the page specified by hyperlink L is traversed, and the recursive structure allows the traverse to continue until a condition is satisfied.

The query mechanism that the WebLog provides is to retrieve information inside an

HTML document and other documents that are referenced to it via hyperlinks.

XML

XML (**E**xtensible **M**arkup **L**anguage) is another markup file type [BPSM+04] for data exchange on the Web. An XML file, like HTML, also consists of elements that are enclosed by a pair of tags, but the tags of XML are user defined and vary from case to case. The tags specify the structure of the file. They act as the attribute name in a relation. All tags in XML must appear in pairs, the start tag (e.g. <name>) and the end tag(e.g. < /name>), and the tags in each pair must match. An element is quoted by a start tag and an end tag. The content between a pair of tags can be text or other elements, or both. As a result, the structure of XML can be nested and have arbitrary depth. Figure 2.5 shows an XML file of the Books.

```
<Books>
        <title>Database System</title>
        <authors>
            <author> Larry </author>
        </authors>
        <year>
            1990
        </year>
        <publisher>Course Technology </publisher>
        <title>Distributed Systems</title>
        <authors>
            <author>  Paul  </author>
            <author>  Bill  </author>
        </authors>
        <year>
            1996
        </year>
        <publisher>Pearson Education </publisher>
</Books>
```

Figure 2.5: An XML file

Many query languages have been proposed for XML, such as XML_QL [DFF+98], Quilt [RCF00], XQuery[BCF+04], and XPath [CD99] to name a few. These languages were all developed particularly for XML. XQuery shares the same data model, functions, and syntax as Xpath 2.0, but provides additional functionalities. They are both used for extracting data from XML documents but have no facilities to update an XML database.

AceDB

AceDB (A C. elegans Database) [TMD92] is a database originally designed for a specific biology organism (as the name indicates) by Jean Thierry-Mieg and Richard Durbin. It has been used for many scientific research fields due to its ability to deal with incomplete data.

Each AceDB database has a predefined schema, which can be thought of as an edge-labeled tree. An instance of a database is composed of data and its schema which conforms to the predefined schema or its subset. Figure 2.6 shows a schema of class "*?Paper*". *Title*, *Author*, *Page*, *Language*, and *Complete_Text* are names of attributes/fields. They are followed by types and restrictions of each attribute. An attribute's tag restricted with UNIQUE indicate that the attribute can only have one value for each instance. Otherwise multi-value is allowed. For example, an instance of *Paper* can have only one title but may have more than one author. AceDB provides a way to specify a regular schema, but it also allows attributes missing inside each class. Figure 2.7 shows a data value of class "*?Paper*".

```
?Paper     Title   UNIQUE Text Int
           Author  Text
           Year    UNIQUE Int
           Page Int UNIQUE Int
           Language UNIQUE  French
                            English
           Complete_Text ?LongText
```

Figure 2.6: An AceDB Schema for "?Paper"

```
dbPaper   Title   "Information Management System"
          Author  "Thomas",
                  "Eric"
          Language English
```

Figure 2.7: An Instance of "?Paper"

AceDB has its own query language, AQL (**A**cedb **Q**uery **L**anguage). AQL retrieves data in more or less the same way as SQL and OQL do, but it does not support database modification. Data cannot be added to database and data in database cannot be changed using AQL. The creation of an AceDB database as well as data updating is managed by system administrators.

2.4.3 Semistructured Systems and Query Languages

With the emerging of various applications of semistructured data and their special purpose query language, researchers are working on general querying language for semistructured data. Data models that represent data with semistructured features and languages for manipulating data upon these models have been proposed.

Lore and Lorel

Lore [MAG+97], developed in Stanford University, is a prototype database management system especially for semistructured data. It provides a friendly user interface for browsing the query results and a data guide for examining the structure of the database. User queries are processed in the "query processor" before the operations for accessing the database, such as obtaining an object from the database or comparing two objects, are performed. In addition to the process which most database query language system adopts, such as parsing a query and executing the operations specified in the query, Lore needs to translate the parser result into an OQL-like query before transforming it into a logical query plan. The logic plan in Lore is a set of relational algebra operators taking a number of arguments, some of which are specified in the Lorel query.

The data model of Lore is OEM (Object Exchange Model), originally designed for the Tsimmis project [PGMW95] in Stanford University. Data under OEM are objects and are organized as a directed graph. Nodes of the graph are objects of different types and can be distinguished by unique object identifiers. The edges are labeled with the name of the object to which it leads. Leaf nodes contain values of primitive types of data and other nodes are objects which point to objects by object identifier. The structure information is contained in those labeled edges and changes dynamically.

Lorel [AQM+97], an extension of OQL, is the query language of Lore. It provides query as well as update functionality.

The main body of a Lorel query is a "select-from-where" form, where a simple path expression or a general path expression can appear in each part of the three. The "where" clause can be omitted depending on the context. The path expression is the building block of Lorel. Simple path expression in Lorel allows the user to form a query to retrieve a set of objects along the labeled edges starting from a named object in the

data model. General path expression consists of symbols representing a pattern of a path, a label name, or an object name in a path expression. It gives the flexibility in forming a query when users do not know the structure of the database precisely. Lorel query language also supports path variables and a path function, *path-of*(). This is useful for queries such as "to get a set of paths that lead to a particular label" or "get all paths that lead to an object with a particular value".
For example, the following query:

```
Select distinct path-of(P)
From Company.# @ P.manager
```

obtains the set of paths ending with *manager* in a database named *Company* .

Lorel aims to provide a query language which is simple to use. It neither supports restructuring a tree to arbitrary depth nor recursive general path expression.

UnQL

Independent of OEM, [BDS95, BDHS96] proposed another data model similar to it. The data model can be thought of as a tree. The leaves in the tree represent atomic values and the non-leaf nodes represent objects. The data model is "value based". Objects do not have identifiers and non-atomic values are represented as sets of label/value pairs.

UnQL (Unstructured Query Language), implemented in Standard ML [AM87], is a semistructure query language for such data models [BDS95, BDHS96, BFS00]. It has the ability to query a relational database which is in graph representation. It also can query data in a graph whose depth is unknown.

UnQL has the following characteristics. First, UnQL adopts patterns matching together with "select-where" form to define queries. User queries use a tree pattern to form the "select" and "where" statement and a generator in the "where" statement will organize all trees in a database that match the specified pattern. Second, UnQL expresses the join operation similar to that in relational algebra with multiple patterns and variable equality. Third, UnQL use structural recursion [BBW92, BLS+94, BNTW95] into which user queries can be translated. Fourth, a wildcard and a repeated wildcard are used in queries to support traversing a tree of arbitrary depth for a given input data

tree. For instance, the query
$select\ \{t\}$
$where\ \_* \Rightarrow \backslash l \Rightarrow \_ \leftarrow Company,\ isstring(l)$
is expected to return all string edges in a database named *Company*. However, UnQL does not support recursive queries because the transitive closure is not expressible in UnQL.

2.4.4 Oracle XML DB

Oracle [GSS04] supports XML management by extending the existing relational database with XML DB, which offers features of importing, storing, querying, and generating XML data.

Oracle provides a new data type, **XMLType**, for databases to contain attributes or relations holding XML documents. The data is stored in CLOB (a data type in Oracle that can hold up to 128 TBs characters in Oracle Database 10g) data type. The data type can be used as any other data type in Oracle. A set of methods for manipulating XMLType data is provided to extract information from XML documents, update XML documents, check if an XML document contains a specified node, examine the relationship of an XMLType and a known XML schema, or execute an XSL [Cla99] transformation. Most of these methods (for querying, updating) apply XPath [CD99] expressions to the XML document and do the corresponding operation. Users who query XML data in Oracle need to have both knowledge of SQL and XPath query language.

2.4.5 Schema

Semistructured data has no predefined regular schema. Users are not able to form a query with specific structure as they can do if the structure of the database is known. For a system that supports a query language which needs to transform a user query into a logical query plan, database schema facilitates the optimization of the logical plan. Hence, schema discovery and presentation becomes one of the important topics in semistructured data management research. Systems that manage semistructured data have different methods of dealing with the schema issue.

In the OEM model or directed graph, schema exists in the labels of the data graph. Lore provides the data guide [GW97, NUWC97] to allow users to browse the dynamically changing database structure and it also provides the system with this structure in query plan optimization. The data guide is a dynamically generated OEM object that summarizes the structure of database. It contains all those paths but only those paths that appear in a database without repetition.

[NAM97] proposed an algorithm for inferring an approximate structure of a large set of semistructured data. Under a graph data model such as OEM, the data is automatically analyzed and classified. Each type of data is assigned a name which represents the characteristic of the type. Attributes are extracted from the data of the type. The preciseness of the classification depends on the threshold value which the user specified. Therefore, the structure result derived from a given semistructured data varies according to its different input threshold.

Oracle XML DB adopts XML Schema [FW04] to validate that an XML instant conforms to an XML schema. An XML Schema specifies the structure, content, and semantics of a set of XML documents and can be composed in any text editor. It must be registered with a unique key to the XML DB before being referenced. Using this key, an XML schema can be recognized and required operations can be performed on it.

2.5 Database Query Language and DBPL

The relational data model has been very successful in many traditional business applications since its introduction three decades ago. Commercial relational implementations, however, show their limitations when applied to applications such as Computer-Aided Design (CAD) or Computer-Aided Manufacturing (CAM) system, Very Large-Scale Integration (VLSI) chip design system, or Geographical Information System (GIS) where new requests including complex data and numerical computations, for database management systems are posed. To meet the new requests, a large amount of work has been done and approaches were adopted for different types of database systems. There are two directions in this field. One is based on the database system using query language and the other is based on the database programming language (DBPL).

2.5.1 Query Language

The common language for the relational database management system is SQL (Structural Query Language). Early research database systems such as System R [ABC$^+$76] and INGRES [HSW75] and many of today's commercial database such as Oracle, DB2, Sybase, and Informix use SQL to access and manipulate data.

The query language provides syntax that focuses on the content to be retrieved. How the information is retrieved is fulfilled in the translation from a user query to a set of operations to be executed against the database. A query needs to be translated into relational algebra for SQL, structural recursion for UnQL, and for Lorel firstly translated into a OQL [1] type language and then further formulated into relational algebra. As the logical query plan is formed automatically by the system program, query optimization is required for an efficient processing. Therefore, the optimization result totally depends on the functionality of the optimization.

A database management system that uses query language usually adopts add on strategy to support new functionalities. For instance, in Oracle, OLAP Data Manipulation Language (OLAP DML) is combined with the original system to support OLAP, XML DB and a XPath-like language for XML, and data exchange is accomplished via an interface between the two languages; users need to be bilingual to operate both languages. As users switch from one language to the other to do information retrieving, the data involved in the process is transported between languages.

2.5.2 Database Programming Language

A language that has power in dealing with both traditional database management together with new requests such as flexible user defined data types, event manager, numerical computation, has been a trend in database research. The purpose of this research is to let the language have both the features of a database query language and that of a programming language, for the programming language usually needs new structures to deal with complex databases and a database query language usually lacks the power to deal with numerical computations.

[1] Object Query Language, a query language similar to SQL for query object-oriented database

One approach is to embed database query language into a programming language. The embedded language needs to be translated into the host language which has facilities to invoke subroutines that interact with the corresponding database.

Another approach for database programming language is to add the data model and data manipulating mechanisms to an existing programming language. Pascal/R [Sch77] combines the relational model with Pascal programming language. PS-algol [ABC+83], derived from S-algol [Mor79, CM82] is one of the programming languages achieving uniform persistence.

jRelix is a high-level general-purpose database programming language based on the relational algebra which abstracts over looping. It subsumes the functionality of a typical database query language [MBC+02, Mer03], therefore, it has all the functions that a query language does and can do much more. Database Programming Language gives the programmer the flexibility to compose queries in the relational algebra and domain algebra level (in jRelix). Optimization is left to the user who will compose an efficient query based on logic and personal experiences.

For a database programming language, new functionalities are accomplished via none or minor extension to the original language. In jRelix, applications such as an expert system can be accomplished without new extensions. Domain algebra in jRelix solved the numerical computation in many applications. Examples will be shown in Chapter 3. jRelix also has facilities to support procedure abstraction and data type abstraction. [Zhe02] presented a data model and its related operation of map overlay in the field of GIS.

2.6 Querying Semistructured Data in a Relational DBPL

The common feature for semistructured data management systems previously discussed is that they are all systems specifically for semistructured data. Using graph data models, the query languages were developed from scratch. A query language specifically designed for semistructured data is not very different from standard database query language[ABS00], therefore it is economic and elegant to enhance the database query

Books

(title	authors (author)	year	publisher)
Database System	Larry	1990	Course Technology
Distributed Systems	Paul Bill	1996	Pearson Education

Table 2.2: The Display form of Relation *Books*

and programming language with functionalities which support semistructured data. On the other hand, a query language that only supports semistructured data will loose the benefits of structured data which provides the schema. Hence, a semistructured language needs to be integrated with that for querying standard structured data. A query language for semistructured data has the same functionality as that for relational data in sense that both need the ability to extract part of the database and the ability of updating the database. Also, as semistructured data may have embedded structure or recursive structure, functionality for traversing these structures is necessary. This is also true for the relational database extended with nested relations.

jRelix, a relational database programming language, has been developed not only for querying relational databases, but also for multi-applications. In addition to supporting applications for expert system, numeric computation, and GIS, it also can deal with queries posed in semistructured data. Take the data in Figure 2.5 in Section 2.4.2 as an example. The XML file can be represented as a relation as shown in Figure 2.8. Relation *Books* in its display form can be seen in Table 2.2. The queries can then be made on the relation.

Query: Find all book authors.

Figure 2.9 shows the query code and the result.

Query: Find the set of book authors, retaining the distinction between books.

Figure 2.10 shows the query code and the result.

There are some other queries such as:

Query: Finding books including Bill as an author, and split into separate entries for each author.

and

Query: Group books under their year of publication.

```
>domain title, author, publisher strg;
>domain year intg;
>domain authors(author);
>relation Books(title,authors,year, publisher) <-{
      ("Database System",    {("Larry")},        1990, "Course Technology"),
      ("Distributed Systems",{("Paul"),("Bill")},1996, "Pearson Education")};
>pr Books;
+----------------------+----------------------+-------------+----------------------+
| title                | authors              | year        | publisher            |
+----------------------+----------------------+-------------+----------------------+
| Database System      | 1                    | 1990        | Course Technology    |
| Distributed Systems  | 2                    | 1996        | Pearson Education    |
+----------------------+----------------------+-------------+----------------------+
relation Books has 2 tuples
>pr authors;
+----------------------+----------------------+
| .id                  | author               |
+----------------------+----------------------+
| 1                    | Larry                |
| 2                    | Bill                 |
| 2                    | Paul                 |
+----------------------+----------------------+
relation .authors has 3 tuples
>
```

Figure 2.8: Relation *Books*

```
>allAuthors<-Books/authors;
>pr allAuthors;
+----------------------+
| author               |
+----------------------+
| Bill                 |
| Larry                |
| Paul                 |
+----------------------+
relation allAuthors has 3 tuples
>
```

Figure 2.9: Query for All Authors

```
>let bookauthors be [authors] in Books;
>BookAuthors<-Books/bookauthors;
>pr BookAuthors;
+---------------------+
| authors             |
+---------------------+
| 1                   |
| 2                   |
+---------------------+
relation BookAuthors has 2 tuples
>pr authors;
+---------------------+---------------------+
| .id                 | author              |
+---------------------+---------------------+
| 1                   | Larry               |
| 2                   | Bill                |
| 2                   | Paul                |
+---------------------+---------------------+
relation .authors has 3 tuples
>
```

Figure 2.10: Query for Book Authors

BooksbyYear

(1990				1996			)
(title	author	publisher	)	(title	author	publisher	)
Database System	Larry	Course Technology		Distributed Systems Paul	Bill	Pearson Education	

Table 2.3: Reorganization of Data

These two queries cannot be solved. In the first query, it's easy to find books including Bill as an author (a T-Select will do). But in order to separate the entries for each author, new facility for grouping attributes into a nested relation is needed.

The second query actually requires a reorganization of the data and attributes: the data of the attribute *year* will become attributes and the data for the new attributes are data from the same tuple in the original relation. The expected result is shown in Table 2.3. In order to turn the value of an attribute into attributes and assign data to them, a mechanism is necessary to manipulate the value of attribute.

For those examples shown in Section 2.4.3, jRelix already has the facility of general path expression to retrieve arbitrary depth of a relation. To get all the string attributes of a relation, **transpose** and **typeof** operators, which will be developed in this work to do the task, are needed. The other way to do it is to use **grep** operator. This is discussed in [Gu05].

In addition, based upon the discussion above, schema discovery is another important issue. In order to get the schema or paths of a database, facilities to trace the parent relation of an attribute and to get the attributes of a relation are needed.

2.6.1 Attribute Metadata Operators

We have discussed that it is necessary to have a language support semistructured data as well as relational data, and we also have seen that a language such as jRelix has the ability to manipulate semistructured data with little effort through the operators for attributes, i.e., attributes metadata.

Generally speaking, metadata is data about data. The "table of contents" of a book and the call numbers of books in libraries are two examples. In the database system, a special group of metadata is data that describes the relations, including the attributes a relation has, the types of attributes, the relationship among nested relations, and so on. It is the specific data for attributes. We call metadata for this purpose "Attribute Metadata" to distinguish it from metadata in other fields.

Attribute metadata is not only demanded for semistructured data, but also in applications of data mining and GIS. Before the writing of this thesis, the jRelix system already had support operators for attribute metadata. These operators include **quote** and **eval** (please refer to Section 3.5.2 and Section 3.5.2 for detail). Due to the new demands from semistructured data, the enhancement of the functionality of these operators is necessary.

To be able to manage semistructured data, a system should have the ability to process schema information such as attributes as data. Therefore, the attribute of the type ATTRIBUTE should be allowed and operators applied to attributes should be supported.

To trace the parent relation of an attribute and get the attributes of a relation which are needed for schema discovery, the **self** operator will be introduced to get the name of the parent relation. The scalar attributes of a relation can be obtained via operator **transpose**.

2.6.2 Related Work in jRelix

Other work for supporting semistructured data in jRelix includes:

1) Recursive nesting declaration and general path expression in relational expression [Yu04].

2) New data type UNION is allowed to let an attribute have different types of data, and the **grep** operator extracts information without specifying the exact structure of the database in the query [Gu05].

3) The **grep** operator extracts information from a text file and the integration of text data source into jRelix [Xie04].

Chapter 3

Overview of jRelix

This chapter gives an overview of the jRelix system and exhibits its application in different fields by giving related examples. The chapter also presents in detail most of the operations of the system upon which our work is based. The content includes relational and domain algebras, and the programming feature of the system. Section 3.1 describes the declaration of domains and relations and the relation initialization. Section 3.2 presents the assignment operation. Section 3.3 is an introduction to views. Section 3.4 depicts the relational algebra followed by an application: Inference Engine. Section 3.5 delineates the domain algebra followed by two applications: Matrix Computation and Finding Area. Section 3.6 summarizes the update operation. Section 3.7 examines the programming language construct in jRelix, specifically computation, abstract data type, and event handler. Section 3.8 describes the distributed data processing.

In this document, the **bold** font is used for keywords. The *italic* font enclosed in angle brackets (i.e.<*italic*>) is used for nonterminals. (`typewriter`)? or (`typewriter`)* is used for meta syntax. The *italic* font alone is used for the name of a relation or an attribute in text and queries.

3.1 Domain and Relation Declaration

3.1.1 Domain Declaration

Domains in jRelix are declared with the keyword **domain**, as follows:

domain <*IDList*> <*Type*>;

where *IDList* is a list of attributes to be declared with the type specified by *Type*.

If a domain itself is a relation, it will construct a nested relation in the relation containing this domain. The declaration will go with the following syntax:

domain <*IDList*> "(" <*domList*> ")";

where *domList* is a list of declared domains which the nested domains specified in *IDList* contain. Figure 3.1 shows examples of domain declaration and the commands for display of the information of the domains.

The syntax for domain declaration in semistructured data (xML in jRelix) is different, as the semistructured data is self-describing, hence the declaration of a new domain occurs when it first appears in the semistructured input of the relation initialization [Yu04]. The syntax is as follows:

< *domainName* **type** = *data_type* > ...</*domainName*>

where *domainName* in both angle brackets is the name of the new domain being declared. The start tag is surrounded by angle brackets, and the end tag has the angle brackets followed by a slash "/". The *domainName* in both tags should be the same. *data_type* is the data type of the domain. It is optional, and the default value is STRING. Examples will be shown together with examples of relation initialization in the next section.

3.1.2 Relation Declaration and Initialization

Relations are declared and initialized in the jRelix system in three ways. In each, a relation is created according to the specified name and stored first in tuple-by-tuple and then in attribute-by-attribute form if it is a flat relation. Nested relations are stored with several flat relations linked together by surrogates. The remaining part of this subsection will introduce the three methods of declaration and initialization and the system forms of flat relations and nested relations.

```
>domain family, name, website strg;
>domain Lscale bool;
>domain company(name,website);
>sd;
------------------------------ Domain Entry ----------------------------
Name        Type            NumRef      IsState   Dom_List
------------------------------------------------------------------------
Lscale      boolean         0           false
company     idlist          0           false     .id, name, website,
website     string          1           false
family      string          0           false
name        string          1           false
------------------------------------------------------------------------
>sd family;
------------------------------ Domain Entry ----------------------------
Name        Type            NumRef      IsState   Dom_List
------------------------------------------------------------------------
family      string          0           false
------------------------------------------------------------------------
>
```

Figure 3.1: Examples of Domain Declaration

In the first way in jRelix, relations are declared by using the keyword **relation** with the following syntax:

relation <*IDList*> `"("` <*IDList*> `")"` `(Initialization)?;`

where the first *IDList* denotes the list of relations being declared and the second *IDList* denotes the attributes on which these relations are defined. The `Initialization` consists of the content of these relations and is optional.

The initialization of relations uses the curly bracket syntax, under which a relation starts and ends with a pair of curly brackets "{" and"}" , while tuples start and end with round brackets "(" and ")". Domains are separated with commas. Figure 3.2 gives an example of relation declaration and initialization along with the command to display a relation.

```
>relation products (family, Lscale)<-{("CPLD",true),("FPGA",true),("SPLD", false)};
>pr relation;
+----------------------+--------+
| family               | Lscale |
+----------------------+--------+
| CPLD                 | true   |
| FPGA                 | true   |
| SPLD                 | false  |
+----------------------+--------+
relation products has 3 tuples
>
```

Figure 3.2: An Example: Relation Declaration and Initialization

The second way jRelix puts data into a relation is by using the name of an existing relation. This method is actually one of the assignment commands of jRelix, as we will see later in Section 3.2.

The third way is for semi-structure data input:

relation `<`*IDList*`>` `Initialization;`

where *IDList* specifies the name of the relations being declared and the `Initialization` is the angle bracket syntax in which the relation starts with the first brackets "<" and ends with the last angle brackets ">". Since an input of the semistructured data contains the information on the names of attributes in the relation, there is no specification for attributes in the declaration of a relation. Figure 3.3 shows a simple example. More details can be found in [Yu04].

```
>relation Ordered <- <Ordered>
                          <company type = string> "Canon"</company>
                          <quantity type = intg>100</quantity>
                          <company> "Dell"</company>
                          <quantity>50</quantity>
                      </Ordered>;
>pr Ordered;
+----------------------+-------------+
| company              | quantity    |
+----------------------+-------------+
|  "Canon"             | 100         |
|  "Dell"              | 50          |
+----------------------+-------------+
relation Ordered has 2 tuples
>
```

Figure 3.3: Initialize a Relation in Semistructured Format

The schema and the content of a relation also can be saved in a file, so the declaration and initialization of a relation can use the following syntax:

relation `<`*rel_name*`> <- <"`*file_path/file_name*`">;`

where *rel_name* is the name of the relation to be declared and "*file_path/file_name*" is the file path and file name where the content of declaration is.

Relations declared and initialized in Figure 3.2 and Figure 3.3 are both flat relations. The form in which they are stored in the system is tuple-by-tuple and then attribute-by-attribute. A nested relation stores in the form of several flat relations, with one top level flat relation and several other flat relations holding the data of nested relations. The linkages between the top level relation and sub-relations are surrogates, which are used to replace values of the nested attributes in the relation. The actual values of the

Contacts

(dept	employee (name	) contactinfo) (phone)
Development	Judy	7473865 9794876
	Sam	3454657 7671234
Technical Support	Tom	7450943

Table 3.1: The Display form of Nested Relation *Contacts*

nested attribute are stored in a separate flat relation, a dot relation, with an additional attribute *.id*, which contains the surrogates linking to the upper level relation. The name of the dot relation is named after the nested attribute and prefixed with a period. For instance, Table 3.1 shows a nested relation *Contacts*. Its declaration and initialization is performed in Figure 3.4, together with the three flat relations that constitute the relation *Contacts*. Figure 3.5 depicts the relationship among these flat relations.

Recursive Relation Declarations and Initializations

In recursive nesting, the name of a relation can be an attribute of the relation itself. The declaration syntax is the same as that of nested relations, except that the name of the nested attribute can also appear in the attributes list. For example, in Figure 3.6, relation *task* is defined on nested relation *subroutine*, which is recursively defined on itself. The null value, "*dc*" [Yan03], is used to terminate the recursion in the initialization of the recursive relation.

3.2 Assignments

jRelix provides two types of assignment operators: the assignment and the incremental assignment. The syntax for assignment operations is:

```
<Identifier> (<- | <+) <Expression>;
<Identifier> "[" <IDList> (<- | <+) <ExpressionList> "]" <Expression>;
```

whereas in the second form, the attributes in the *ExpressionList* are renamed to the attributes specified in the *IDList*.

```
>domain dept, name, phone strg;
>domain contactinfo(phone);
>domain employee (name, contactinfo);
>relation Contacts(dept, employee) <-
{("Technical Support", {("Sam",{("7671234"),("3454657")}),
                        ("Judy",{("7473865"),("9794876")})}),
 ("Development",       {("Tom",{("7450943")})})};
>pr Contacts;
+---------------------+---------------------+
| dept                | employee            |
+---------------------+---------------------+
| Development         | 4                   |
| Technical Support   | 1                   |
+---------------------+---------------------+
relation Contacts has 2 tuples
>pr .employee;
+---------------------+---------------------+---------------------+
| .id                 | name                | contactinfo         |
+---------------------+---------------------+---------------------+
| 1                   | Judy                | 3                   |
| 1                   | Sam                 | 2                   |
| 4                   | Tom                 | 5                   |
+---------------------+---------------------+---------------------+
relation .employee has 3 tuples
>pr .contactinfo;
+---------------------+---------------------+
| .id                 | phone               |
+---------------------+---------------------+
| 2                   | 3454657             |
| 2                   | 7671234             |
| 3                   | 7473865             |
| 3                   | 9794876             |
| 5                   | 7450943             |
+---------------------+---------------------+
relation .contactinfo has 5 tuples
>
```

Figure 3.4: Print Form of Nested Relation *Contacts*

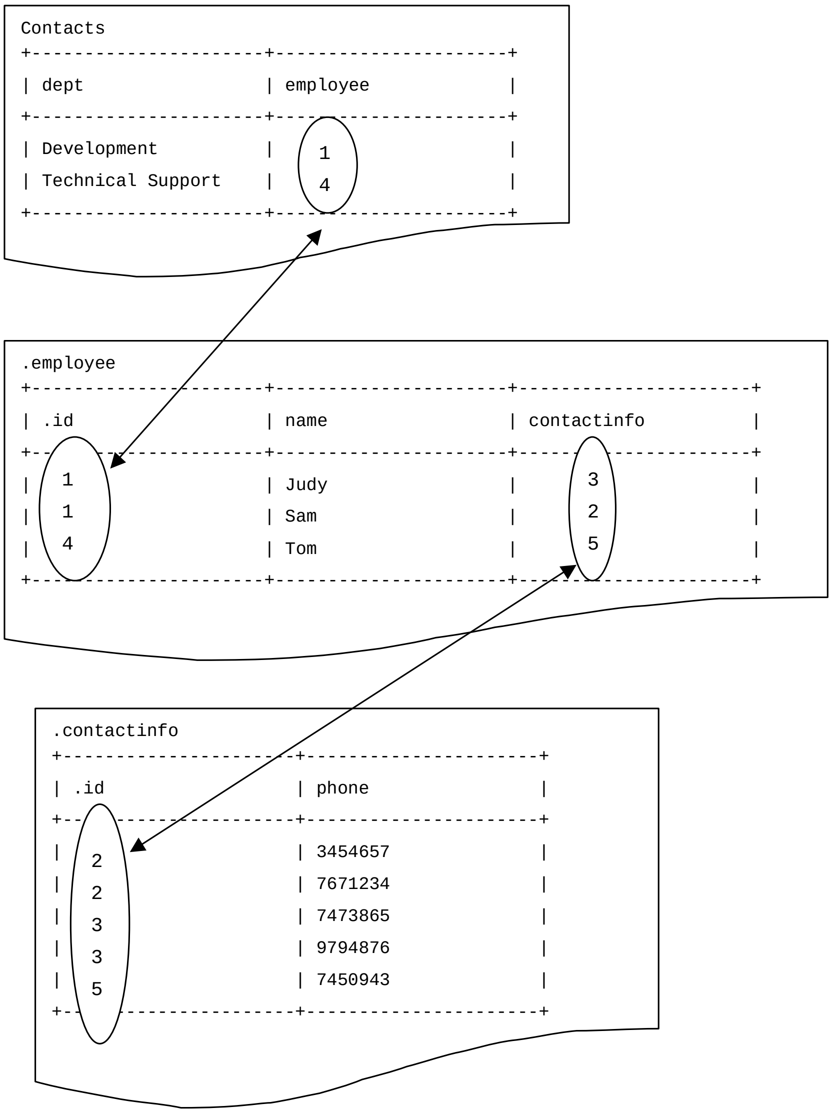

Figure 3.5: Storage of Nested Relation *Contacts*

```
>domain routine strg;
>domain subroutine(routine, subroutine);
>relation task(routine, subroutine) <-  {("A", {("A1",{("A11", dc)}),
                                                ("A2",{("A21",dc)}) }),
                                         ("B", dc),
                                         ("C", {("C1",dc)})};
>pr task;
+----------------------+----------------------+
| routine              | subroutine           |
+----------------------+----------------------+
| A                    | 9                    |
| B                    | dc                   |
| C                    | 12                   |
+----------------------+----------------------+
relation task has 3 tuples
+----------------------+----------------------+----------------------+
| .id                  | routine              | subroutine           |
+----------------------+----------------------+----------------------+
| 9                    | A1                   | 10                   |
| 9                    | A2                   | 11                   |
| 10                   | A11                  | dc                   |
| 11                   | A21                  | dc                   |
| 12                   | C1                   | dc                   |
+----------------------+----------------------+----------------------+
relation .subroutine (Compact Form) has 5 tuples
>
```

Figure 3.6: Recursive Nesting Declaration

The assignment (<–) operator totally replaces the left-hand side relation, which is specified in *Identifier*, with the right-hand side relational expression, *Expression*, no matter whether the left-hand side relation is previously defined or not. The attributes of the left-hand side relation do not need to be the same as the right hand side relation.

The incremental (<+) operator adds the additional tuples generated by the right-hand side relational expression to the left-hand side relation. The attributes of the left-hand side relation must be the same as those of the right-hand side relation.

The attributes of the left-hand side relation under either assignment or incremental assignment operators also can be explicitly specified (but previously declared) and matched to the attributes on the right. Figure 3.7 shows examples of assignment operations.

3.3 Views

The view is a mechanism to define a relation according to an expression as in an assignment operation. The difference is that in an assignment operation, the right-hand side expression is evaluated and the value is assigned to the left-hand side relation, whereas

```
>domain category, article, items strg;
>domain revenue intg;
>relation R (category, article, revenue) <-{
  ("computer","desktop",50000),("computer","notebook", 30000),
  ("TV","LCD",80000),           ("TV","plasma",40000)};
>pr R;
+----------------------+----------------------+-------------+
| category             | article              | revenue     |
+----------------------+----------------------+-------------+
| TV                   | LCD                  | 80000       |
| TV                   | plasma               | 40000       |
| computer             | desktop              | 50000       |
| computer             | notebook             | 30000       |
+----------------------+----------------------+-------------+
relation R has 4 tuples
>Report [category, items, revenue <- category, article, revenue] R;
>pr Report;
+----------------------+----------------------+-------------+
| category             | items                | revenue     |
+----------------------+----------------------+-------------+
| TV                   | LCD                  | 80000       |
| TV                   | plasma               | 40000       |
| computer             | desktop              | 50000       |
| computer             | notebook             | 30000       |
+----------------------+----------------------+-------------+
relation Report has 4 tuples
>relation newReport(category, items, revenue) <-
                   {("telephone","telephons", 6000),
                    ("telephone","Fax machines", 10000)};
>Report <+ newReport;
>pr Report;
+----------------------+----------------------+-------------+
| category             | items                | revenue     |
+----------------------+----------------------+-------------+
| TV                   | LCD                  | 80000       |
| TV                   | plasma               | 40000       |
| computer             | desktop              | 50000       |
| computer             | notebook             | 30000       |
| telephone            | Fax machines         | 10000       |
| telephone            | telephons            | 6000        |
+----------------------+----------------------+-------------+
relation Report has 6 tuples
>
```

Figure 3.7: Examples of Assignment Operations

no evaluation, hence no value assignment, is performed in a view operation. The view will be evaluated later when there is an assignment operation or other operations such as **pr** where the view is involved. The syntax for view is as follows:

<*Identifier*> **is** <*Expression*>;

where *Identifier* is the name of the view and *Expression* specifies the content of the view. Figure 3.8 shows an example of a view declaration and its evaluation.

```
>domain level intg;
>relation Accumulate (level) <- {(1)};
>pr Accumulate;
+-------------+
| level       |
+-------------+
| 1           |
+-------------+
relation Accumulate has 1 tuple
>let level' be level +1;
>let level be level';
>Accumulate is Accumulate ujoin [level] in [level'] where level <3 in Accumulate;
>pr Accumulate;
+-------------+
| level       |
+-------------+
| 1           |
| 2           |
| 3           |
+-------------+
expression has 3 tuples
>
```

Figure 3.8: An Example of view

3.4 Relational Algebra

Relational algebra is essential to manipulate relations in a database system. It considers the relation as an atomic unit. All operations are performed on relations and the source relations are not affected by the operations. Also, the production of operations is relations. This allows the construction of complex expressions with a number of relational operators. The operations are categorized into unary operations and binary operations, and will be introduced in the following subsections.

3.4.1 Unary Operators

Unary Operators in jRelix include: **projection**, **selection**, **T-selection**, **QT-selection**, **eval**, and **quote**.

Projection

Projection extracts a subset of a source relation according to a set of a specified domain, which is called a projector. Duplicate tuples are removed from the result relation. If a projector is null, then the result relation contains only one attribute of type BOOLEAN with the name "*.bool*". The value would be *true* if the source relation contains at least one tuple and *false* if the source relation is empty. The syntax for projection is as follows:

"[" <*IDList*> "]" **in** <*RelationalExpression*>

where *IDList* is the actual attribute list of the new relation, which is, of course, the attributes (actual or virtual) of the source relation: the *RelationalExpression*. The example in Figure 3.9 gives the result of "all items and categories in Report."

```
>Merchandise <- [category, items] in Report;
>pr Merchandise;
+----------------------+----------------------+
| category             | items                |
+----------------------+----------------------+
| TV                   | LCD                  |
| TV                   | plasma               |
| computer             | desktop              |
| computer             | notebook             |
| telephone            | Fax machines         |
| telephone            | telephons            |
+----------------------+----------------------+
relation Merchandise has 6 tuples
>
```

Figure 3.9: An Example of Projection

Selection

Selection extracts a subset of the source relation according to the condition specified in the selection clause. The select condition must be evaluated to be *true* or *false* for each tuple in the source relation and only those with condition being *true* are selected to be tuples in the result relation. The attributes of the result relation are the same as those in the source relation. The syntax is as follows:

where <*SelectClause*> **in** <*Expression*>

Figure 3.10 shows an example that retrieves "all tuples containing *Computer* in relation *Report*".

```
>CompReport <- where category = "computer" in Report;
>pr CompReport;
+----------------------+----------------------+-------------+
| category             | items                | revenue     |
+----------------------+----------------------+-------------+
| computer             | desktop              | 50000       |
| computer             | notebook             | 30000       |
+----------------------+----------------------+-------------+
relation CompReport has 2 tuples
>
```

Figure 3.10: An Example of Selection Operation

T-Selection

T-selection is the combination of projection and selection. The production is a relation that is the subset of the source relation. The syntax is as follows:

"[" *<IDList>* "]" **where** *<SelectClause>* **in** *<Expression>*

As the definition of T-selection shows, the operation selects the tuples according to *SelectionClause*, while the attributes are specified in *IDList*.

If only the attribute *item* is required in the example in the previous subsection, the result for "all items of computer in *Report*" will be the one shown in Figure 3.11.

```
>Computer <- [items] where category = "computer" in Report;
>pr Computer;
+----------------------+
| items                |
+----------------------+
| desktop              |
| notebook             |
+----------------------+
relation Computer has 2 tuples
>
```

Figure 3.11: An Example of T-Selection Operation

3.4.2 Binary Operators

The binary operations on relations are extensions of the binary operations on sets [Mer84]. jRelix provides two categories of binary operators, μ-joins that contains 7 operators and σ-joins that contains 12 operators. The results of μ-joins and σ-joins are also relations. The syntax for these join operators are shown as follows:

<Expression> **JoinOperator** *<Expression>*

|

<Expression> "[" *<ExprList>*:**JoinOperator**:*<ExprList>* "]" *<Expression>*

In the first production, the two operands join on their common domains. If no common domains exist in the two relations to be joined, domains are explicitly specified as join domains by using the second production.

μ-joins

μ-joins corresponds to the set operations including union, intersection, difference, and symmetric difference of sets. Their relationship is shown in Table 3.2.

μ-join	Operator	Set Operator
natural join	**ijoin** or **natjoin**	$\cap$
union join	**ujoin**	$\cup$
left join	**ljoin**	
right join	**rjoin**	
left difference join	**djoin** or **dljoin**	$-$
right difference join	**drjoin**	
symmetric difference join	**sjoin**	$+$

Table 3.2: μ-joins and Set Operators

Generally, μ-joins can be defined in three parts: the center, the right, and the left. Given two relations $R(X,Y)$ and $S(Y,Z)$ sharing a common attribute set Y, the three parts are defined as following:

$\mathit{center} \equiv \{(x,y,z)|(x,y) \in R \wedge (y,z) \in S\}$

$\mathit{left} \equiv \{(x,y,dc)|(x,y) \in R \wedge \forall z, (y,z) \notin S\}$

$\mathit{right} \equiv \{(dc,y,z)|(y,z) \in S \wedge \forall x, (x,y) \notin R\}$

If two relations $R(W,X)$ and $S(Y,Z)$ do not share a common set, the three parts are defined as following:

$\mathit{center} \equiv \{(w,x,y,z)|(w,x) \in R \wedge (y,z) \in S \wedge x = y\}$

$\mathit{left} \equiv \{(w,x,y,dc)|(w,x) \in R \wedge x = y \wedge \forall z, (y,z) \notin S\}$

$\mathit{right} \equiv \{(dc,x,y,z)|(y,z) \in S \wedge x = y \wedge \forall x, (x,y) \notin R\}$

Note here the symbol *dc* stands for one of the null values in jRelix. The other is *dk*. Details of the null values can be found in [Mer84].

The μ-joins can then be defined as:

R **ijoin** $S = \mathit{center}$

R **ujoin** $S = \mathit{left} \cup \mathit{center} \cup \mathit{right}$

R **djoin** $S = \mathit{left}$

R **drjoin** $S = \mathit{right}$

R **lrjoin** $S = \mathit{left} \cup \mathit{center}$

R **rjoin** $S = \mathit{center} \cup \mathit{right}$

R **sjoin** $S = \mathit{left} \cup \mathit{right}$

Based on relations in Figure 3.12, one is a relation of students and their id numbers and the other is a relation of students and the courses they selected, the following query can be achieved.

```
>domain ID intg;
>domain name strg;
>relation Student (ID, name) <-{(121, "John"),(122, "Mary")};
>pr Student;
+-------------+----------------------+
| ID          | name                 |
+-------------+----------------------+
| 121         | John                 |
| 122         | Mary                 |
+-------------+----------------------+
relation Student has 2 tuples
>domain course strg;
>relation Course(ID, course)<-{(121,"DB"), (135,"Algorithm")};
>pr Course;
+-------------+----------------------+
| ID          | course               |
+-------------+----------------------+
| 121         | DB                   |
| 135         | Algorithm            |
+-------------+----------------------+
relation Course has 2 tuples
>
```

Figure 3.12: Relation *Student* and *Course*

Query: find the course a student selected and his/her id number.

Figure 3.13 exhibits the results.

```
>StudentCourse <- Student ijoin Course;
>pr StudentCourse;
+-------------+----------------------+----------------------+
| ID          | name                 | course               |
+-------------+----------------------+----------------------+
| 121         | John                 | DB                   |
+-------------+----------------------+----------------------+
relation StudentCourse has 1 tuple
>
```

Figure 3.13: **ijoin** Operation

Query: find all students who do not select any course.

Figure 3.14 depicts the results.

```
>StudentCourse <- Student djoin Course;
>pr StudentCourse;
+-------------+----------------------+
| ID          | name                 |
+-------------+----------------------+
| 122         | Mary                 |
+-------------+----------------------+
relation StudentCourse has 1 tuple
>
```

Figure 3.14: **djoin** Operation

σ-joins

The σ-joins extends the truth-valued comparison operation on sets to relations by applying them to each set of values of the join attribute for each of the other values in the two relations [Mer84]. The relationship between σ-joins and the set operations are shown in Table 3.3. The grouping facility σ-joins makes it possible in applications such as Inference Engine that the result operation should be determined by a set of tuples grouped by the join attributes. An example of this will be shown at the end of this subsection. Note that the attributes of the result relation from a μ-join operation are the union of the attributes of the two source relations (except **djoin** and **drjoin**) and that the attributes from a σ-join operation are the symmetric difference of the attributes from the two source relations.

sigma-join	Operator	Set Operator
natural composition	**icomp** or **natcomp**	$\not{\cap\!\!\!\!\circ}$
empty intersection join	**!ejoin** or **sep**	$\cap\!\!\!\!\circ$
equal join	**eqjoin**	$=$
not equal join	**!eqjoin** or **not eqjoin**	$\neq$
greater than or equal join	**gejoin** or **sup** or **div**	$\supseteq$
not greater than or equal join	! **gejoin**	$\not\supseteq$
greater than join	**gtjoin**	$\supset$
not greater than join	! **gtjoin**	$\not\supset$
less than or equal join	**lejoin** or **sub**	$\subseteq$
not less than or equal join	! **lejoin**	$\not\subseteq$
less than join	**ltjoin**	$\subset$
not less than join	! **ltjoin**	$\not\subset$

Table 3.3: Summary of σ-joins

The σ-joins can be defined by using the following notation. In relations $R(X, Y)$ and $S(Y,Z)$, R_w is the set of values of X associated by R with a given value, w of W, and S_z is the set of values of Y associated by S with a given value, z of Z. If W and X are disjoint sets of the attributes of R, and Y and Z are disjoint sets of the attributes of S, the following definitions hold. (X and Y must be at least compatible attribute sets, though they may be the same set of attributes.)

R **icomp** $S \equiv \{(w,z)|R_w \cap S_z \neq \emptyset\}$
R **sep** $S \equiv \{(w,z)|R_w \cap S_z = \emptyset\}$
R **eqjoin** $S \equiv \{(w,z)|R_w = S_z\}$
R **!eqjoin** $S \equiv \{(w,z)|R_w \neq S_z\}$
R **sup** $S \equiv \{(w,z)|R_w \supseteq S_z\}$
R **!gejoin** $S \equiv \{(w,z)|R_w \not\supseteq S_z\}$
R **gtjoin** $S \equiv \{(w,z)|R_w \supset S_z\}$
R **!gtjoin** $S \equiv \{(w,z)|R_w \not\supset S_z\}$
R **lejoin** $S \equiv \{(w,z)|R_w \subseteq S_z\}$
R **!lejoin** $S \equiv \{(w,z)|R_w \not\subseteq S_z\}$
R **ltjoin** $S \equiv \{(w,z)|R_w \subset S_z\}$
R **!ltjoin** $S \equiv \{(w,z)|R_w \not\subset S_z\}$

The following example uses the relations defined in Figure 3.12.

Query: find the course selected by student and the student ID number.

Figure 3.15 shows the result.

```
>SIDCourse <- Student icomp Course;
>pr SIDCourse;
+-------------+----------------------+
| ID          | course               |
+-------------+----------------------+
| 121         | DB                   |
+-------------+----------------------+
relation SIDCourse has 1 tuple
>
```

Figure 3.15: **icomp** Operation

Application: Inference Engine

The following is an example of an inference engine using Horn Clauses. Figure 3.16 shows the set of conditions each of which implies a conclusion. The starting point of the inference is given in relation *Known* in Figure 3.17. The recursive view gives all conclusions inferred from the Horn Clause. This one-line inference engine can be expanded into a more complex one for applications in an expert system, together with the "expert system shell"[Mer91].

```
>domain rule intg;
>domain precond, conclusion strg;
>relation Horn(rule, precond, conclusion) <-{
(1, "near the road",        "good location"), (1, "have parking lot",     "good location"),
(2, "good location",        "make profit"),   (2, "good management team", "make profit"),
(3, "good neighborhood",    "make profit"),   (3, "good location",        "make profit"),
(4, "make profit",          "easy to sell"),  (4, "good location",        "easy to sell")};
>pr Horn;
+-------------+----------------------+----------------------+
| rule        | precond              | conclusion           |
+-------------+----------------------+----------------------+
| 1           | have parking lot     | good location        |
| 1           | near the road        | good location        |
| 2           | good location        | make profit          |
| 2           | good management team | make profit          |
| 3           | good location        | make profit          |
| 3           | good neighborhood    | make profit          |
| 4           | good location        | easy to sell         |
| 4           | make profit          | easy to sell         |
+-------------+----------------------+----------------------+
relation Horn has 8 tuples
>
```

Figure 3.16: Horn Clause: Known Preconditions and Conclusions

3.5 Domain Algebra

The domain algebra provides a set of operations applied to attributes[Mer77, Mer84]. It is used by the declaration of virtual attributes and their actualization in relations. The domain algebra provides a way to do calculations that are not possible in relational algebra but necessary in a programming language.

The result of the domain algebra is a virtual domain, as the operations on attributes are not associated with any particular relation until they are actualized. In other words, a domain algebra operation can be applied to any relation where the operation can work. The syntax for virtual domain declaration is:

```
>relation Known(conclusion)<-{("near the road"), ("have parking lot"), ("good management team")};
>pr Known;
+----------------------+
| conclusion           |
+----------------------+
| good management team |
| have parking lot     |
| near the road        |
+----------------------+
relation Known has 3 tuples
>relation Conclusion (conclusion);
>Conclusion is Known ujoin ([conclusion] in (Conclusion[conclusion:sup:precond]Horn));
>pr Conclusion;
+----------------------+
| conclusion           |
+----------------------+
| easy to sell         |
| good location        |
| good management team |
| have parking lot     |
| make profit          |
| near the road        |
+----------------------+
expression has 6 tuples
>
```

Figure 3.17: Horn Clause: the Given Knowledge and the Conclusions

let <*Identifier*> **be** <*Expression*> ;

where the *identifier* denoted the name of the virtual domain being declared, and the *Expression* is the value of the virtual domain.

The domain algebra can be divided into two categories: the horizontal operations and the vertical operations. Horizontal operations operate within tuples so they also are called scalar operations and vertical operations work across the tuples and they also are known as aggregate operations.

3.5.1 Horizontal Operations

Horizontal operations of domain algebra support arithmetic, logic, and string processing on attributes that can be actualized. These operations include constant definition, renaming, arithmetic functions, conditional statements, and relational algebra. Table 3.4 presents some examples.

3.5.2 Vertical Operations

Vertical operations of domain algebra support reduction and functional mapping operation. Reduction includes operations of both simple reduction and equivalence reduction

functions	declarations	actualized value
constant definition	**let** level **be** 1;	1
renaming	**let** level **be** level″;	same as level″
arithmetic function	**let** level″ **be** level + level′;	sum of level and level′
conditional statement	**let** X **be if** level = maxlevel **then** true **else** false;	true/false
relational algebra	**let** visit″ **be** visit′ **ujoin** visit;	a relation resulting from visit′ **ujoin** visit

Table 3.4: Horizontal Operation and Examples in Domain Algebra

while functional mapping includes operations of both functional mapping and partial functional mapping.

Reduction

Simple reduction generates a single result for each tuple of the relation from all of the tuples of a single attribute in the relation. The syntax is:

let <*Identifier*> **be red** <*operator*> **of** <*expression*>

where *Identifier* denotes the name of the virtual domain being declared and *expression* denotes the operand of the operator. The operator in the syntax must be both commutative and associative. As the order of tuples in a relation doesn't matter, only the commutative and associative operators that work on the value of an attribute will produce the same value regardless of the order of the tuples. These operators include: addition (+), multiplication (∗), **max** and **min** for numeric operations, **and** and **or** for Boolean operations, and **ijoin**, **ujoin**, and **sjoin** for relational operations etc. An example of the reduction operation with addition operator appears in Figure 3.18. Relation *Report* can be found in Figure 3.7, the final result from incremental assignment.

```
>let TotRevenue be red + of revenue;
>TotRev <- [TotRevenue] in Report;
>pr TotRev;
+-------------+
| TotRevenue  |
+-------------+
| 216000      |
+-------------+
relation TotRev has 1 tuple
>
```

Figure 3.18: An Example of Reduction Operation

EmpName'
(empname) (name)
Judy Sam
Tom

Table 3.5: The Display form of Nested Relation *EmpName'*

Level Lifting and Unnest

Nested relations allow relations as values of attributes. Operations on the nested relation are nothing new except for the declaration for the nested relation as seen in Section 3.1. All that is needed is to apply operations of relational algebra to the relational attributes. Subsuming the relational algebra into domain algebra gives the nested relation for free[1]. Consider the following example. Projection operations shown in Figure 3.19 achieve "all the employees from the relation *Contacts*". Relation *EmpName'* is a nested relation (it is clearer to see it in its display form as in Table 3.5) which is not convenient to access. Without new operators, but only using reduction and anonymity, we can have a flat relation which is the result of level lifting. The code and the result is shown in Figure 3.20.

The syntactic sugar for anonymous **red ujoin of** in projection operation is introduced in [Mer03]. In the above example,

```
EmpName <- [red ujoin of empname] in Contacts;
```

is equivalent to:

```
EmpName <- Contacts / empname;
```

More usages of this short form, or path expression, can be found in [Yu04]. Now the query can be simplified to the one shown in Figure 3.21.

Equivalence Reduction

Equivalence reduction provides a grouping mechanism in reduction. These groups are equivalent as they have the same value within the group in terms of the specified at-

[1] A new operator for grouping attributes into a nested relation will be added in this thesis. This new operator in jRelix is intended to support applications such as graph queries and semistructured data. Please refer to Chapter 4 section 4.5 for details.

```
>let empname be [name] in employee;
>EmpName' <- [empname] in Contacts;
>pr EmpName';
+----------------------+
| empname              |
+----------------------+
| 14                   |
| 15                   |
+----------------------+
relation EmpName' has 2 tuples
>pr .empname;
+----------------------+----------------------+
| .id                  | name                 |
+----------------------+----------------------+
| 14                   | Tom                  |
| 15                   | Judy                 |
| 15                   | Sam                  |
+----------------------+----------------------+
relation .empname has 3 tuples
>
```

Figure 3.19: The Nested Relation: *EmpName'*

```
>let empname be [name] in employee;
>EmpName <-[red ujoin of empname] in Contacts;
>pr EmpName;
+----------------------+
| name                 |
+----------------------+
| Judy                 |
| Sam                  |
| Tom                  |
+----------------------+
relation EmpName has 3 tuples
>
```

Figure 3.20: The Unnested Relation: *EmpName*

```
>EmpName <-[name] in Contacts/empname ;
>pr EmpName;
+----------------------+
| name                 |
+----------------------+
| Judy                 |
| Sam                  |
| Tom                  |
+----------------------+
relation EmpName has 3 tuples
>
```

Figure 3.21: Using Syntactic Sugar in the Query

tribute list. The syntax for equivalence reduction is:

let <*Identifier*> **be equiv** <*operator*> **of** <*expression*> **by** <*expressionList*>

the *expressionList* following the **by** keyword specifies the sort attributes, according to which the reduction is performed. Figure 3.22 presents an example of the equivalence reduction.

```
>let crev be equiv + of revenue by category;
>CRev <-[category, items, crev] in Report;
>pr CRev;
+----------------------+----------------------+-------------+
| category             | items                | crev        |
+----------------------+----------------------+-------------+
| TV                   | LCD                  | 120000      |
| TV                   | plasma               | 120000      |
| computer             | desktop              | 80000       |
| computer             | notebook             | 80000       |
| telephone            | Fax machines         | 16000       |
| telephone            | telephons            | 16000       |
+----------------------+----------------------+-------------+
relation CRev has 6 tuples
>
```

Figure 3.22: An Example of Equivalence Reduction

Functional Mapping

Functional mapping provides ways to perform operations which include non-commutative and non-associative operators. By sorting on one or more attributes in a relation, the order of tuples is decided. Therefore, non-commutative and/or non-associative operations are possible. Also calculations for cumulative sums and other operations are supported with the ordered attributes. Here is the syntax for function mapping:

let <*Identifier*> **be fun** <*operator*> **of** <*expression*> **order** <*expressionList*>

where **fun** is the keyword for function mapping syntax and the order of tuples is provided by the *expressionList* followed by keyword **order**. Figure 3.23 presents an example of functional mapping to "get the revenue rank in relation *Report*".

Figure 3.24 shows another example with the operator "cat", which is one of the non-associative operators.

```
>let revRank be fun + of 1 order revenue;
>RevRank <- [category, items, revenue, revRank] in Report;
>pr RevRank;
+----------------------+----------------------+-------------+-------------+
| category             | items                | revenue     | revRank     |
+----------------------+----------------------+-------------+-------------+
| TV                   | LCD                  | 80000       | 6           |
| TV                   | plasma               | 40000       | 4           |
| computer             | desktop              | 50000       | 5           |
| computer             | notebook             | 30000       | 3           |
| telephone            | Fax machines         | 10000       | 2           |
| telephone            | telephons            | 6000        | 1           |
+----------------------+----------------------+-------------+-------------+
relation RevRank has 6 tuples
>
```

Figure 3.23: An Example of Functional Mapping Operation

```
>domain num intg;
>domain word strg;
>relation Words(word, num)<-{("Morning ",2 ),("One", 4),("Good ", 1),("Every ", 3)};
>pr Words;
+----------------------+-------------+
| word                 | num         |
+----------------------+-------------+
| Every                | 3           |
| Good                 | 1           |
| Morning              | 2           |
| One                  | 4           |
+----------------------+-------------+
relation Words has 4 tuples
>let sentence be fun cat of word order num;
>Sentence <-[num, sentence] in Words;
>pr Sentence;
+-------------+------------------------+
| num         | sentence               |
+-------------+------------------------+
| 1           | Good                   |
| 2           | Good Morning           |
| 3           | Good Morning Every     |
| 4           | Good Morning Every One |
+-------------+------------------------+
relation Sentence has 4 tuples
>
```

Figure 3.24: Functional Mapping Applied to a Non-associative Operator

Partial Functional Mapping

Partial functional mapping adds a grouping facility to functional mapping in the same way that equivalence reduction does to reduction. The syntax for partial functional mapping follows:

let <*Identifier*> **be fun** <*operator*> **of** <*expression*> **order** <*expressionList*> **by** <*expressionList*>

The group condition is specified in the *expressionList* followed keyword **by** in the above syntax.

Figure 3.25 shows the use of partial functional mapping to calculate the pay rank within each department.

```
>let catrevRank be par + of 1 order revenue by category;
>CatRevRank <- [category, items, revenue, catrevRank] in Report;
>pr CatRevRank;
+----------------------+----------------------+-------------+-------------+
| category             | items                | revenue     | catrevRank  |
+----------------------+----------------------+-------------+-------------+
| TV                   | LCD                  | 80000       | 2           |
| TV                   | plasma               | 40000       | 1           |
| computer             | desktop              | 50000       | 2           |
| computer             | notebook             | 30000       | 1           |
| telephone            | Fax machines         | 10000       | 2           |
| telephone            | telephons            | 6000        | 1           |
+----------------------+----------------------+-------------+-------------+
relation CatRevRank has 6 tuples
>
```

Figure 3.25: An Example of Partial Function Mapping Operation

Set Operation

In jRelix, the set operation [Roz02] transforms the value of a unary relation (i.e., a relation that has only one attribute) to a set of attributes in the place where the attribute list can appear. The unary relation should be defined to have an attribute of type ATTRIBUTE. The projector list in a projection operation and the **by** list in equivalence reduction and partial functional mapping are those sites that a set operation can take place. Following are some examples.

Figure 3.26 shows relations that are used to demonstrate the set operation. Figure 3.27 presents a set operation performed in the **by** list in equivalence reduction and partial functional mapping of queries. Figure 3.28 displays an example of set operation in the projector list in projection operation.

```
>domain Attr attr;
>domain L,W,T intg;
>relation C(L,W,T)<-{(1,1,5),(1,1,10), (2,2,5), (3,4,20)};
>pr C;
+-------------+-------------+-------------+
| L           | W           | T           |
+-------------+-------------+-------------+
| 1           | 1           | 5           |
| 1           | 1           | 10          |
| 2           | 2           | 5           |
| 3           | 4           | 20          |
+-------------+-------------+-------------+
relation C has 4 tuples
>relation aLWT(Attr)<-{(L),(W),(T)};
>pr aLWT;
+---------------------+
| Attr                |
+---------------------+
| L                   |
| T                   |
| W                   |
+---------------------+
relation aLWT has 3 tuples
>relation aT(Attr)<-{(T)};
>pr aT;
+---------------------+
| Attr                |
+---------------------+
| T                   |
+---------------------+
relation aT has 1 tuple
>
```

Figure 3.26: Relations for Set Operation

Quote

The **quote** operator [Roz02] converts an attribute name to an attribute metadata. The attribute following the **quote** operator will not be evaluated during the query evaluation. The general syntax of **quote** is:

let <*Identifier*> **be quote** <*Identifier*>;

in which the first *Identifier* is the virtual domain being declared and the second *Identifier* is a declared domain used as the value of the virtual domain. The virtual domain is of the type ATTRIBUTE. Relation *Ordered* can be found in Figure 3.3. Figure 3.29 presents an example.

```
>let eqvT be equiv + of 1 by (aLWT djoin aT);
>C1<-[L,W,T,eqvT] in C;
>pr C1;
+-------------+-------------+-------------+-------------+
| L           | W           | T           | eqvT        |
+-------------+-------------+-------------+-------------+
| 1           | 1           | 5           | 2           |
| 1           | 1           | 10          | 2           |
| 2           | 2           | 5           | 1           |
| 3           | 4           | 20          | 1           |
+-------------+-------------+-------------+-------------+
relation C1 has 4 tuples
>let pfTW be par + of 1 order T by (aLWT djoin aT);
>C2<-[L,W,T,pfTW] in C;
>pr C2;
+-------------+-------------+-------------+-------------+
| L           | W           | T           | pfTW        |
+-------------+-------------+-------------+-------------+
| 1           | 1           | 5           | 1           |
| 1           | 1           | 10          | 2           |
| 2           | 2           | 5           | 1           |
| 3           | 4           | 20          | 1           |
+-------------+-------------+-------------+-------------+
relation C2 has 4 tuples
>
```

Figure 3.27: Examples of Set Operation in Equivalence Reduction and Partial Functional Mapping

```
>reC<-[[Attr] in aLWT] in C1;
>pr reC;
+-------------+-------------+-------------+
| L           | T           | W           |
+-------------+-------------+-------------+
| 1           | 5           | 1           |
| 1           | 10          | 1           |
| 2           | 5           | 2           |
| 3           | 20          | 4           |
+-------------+-------------+-------------+
relation reC has 4 tuples
>
```

Figure 3.28: An Example of Set Operation in Projection

```
>let computer be quote desktop;
>QuoteTest <-[company, quantity, computer] in Ordered;
>pr QuoteTest;
+----------------------+-------------+----------------------+
| company              | quantity    | computer             |
+----------------------+-------------+----------------------+
| "Canon"              | 100         | desktop              |
| "Dell"               | 50          | desktop              |
+----------------------+-------------+----------------------+
relation QuoteTest has 2 tuples
>
```

Figure 3.29: An Example of **quote** Operator

Eval

The **eval** operator [Roz02] evaluates the value of a unary relation which has an attribute of the type ATTRIBUTE. The value to be assigned is a constant, therefore the operation is a static evaluation. The general syntax of **eval** is:

let eval <*relation*> **be** <*constant*>;

where the *relation* is a unary relation. The type of the attribute of the relation is ATTRIBUTE. The *constant* is the value to be assigned to each attribute which is the domain of the unary relation. Figure 3.30 shows an example. Note that the value to be assigned does not have to be a constant. It can be any expression that results in a valid value of an attribute. Functions of **eval** will be extended in this thesis. Please refer to Section 4.3 for details.

```
>let eval aT be 0;
>LW<-[L,W,T] in [L,W] in C;
>pr LW;
+-------------+-------------+-------------+
| L           | W           | T           |
+-------------+-------------+-------------+
| 1           | 1           | 0           |
| 2           | 2           | 0           |
| 3           | 4           | 0           |
+-------------+-------------+-------------+
relation LW has 3 tuples
>
```

Figure 3.30: An Example of Static **eval** Operation

Application: Matrix Computation

With relational algebra and domain algebra introduced, one of the applications is to do matrix multiplication. For example, suppose there are two matrices as shown in Figure 3.31 and Figure 3.32. The product of A and B is matrix C in Figure 3.33. The matrices are modeled in relations and the relations for matrix A and B are shown in Figure 3.34. To obtain the product of A and B, equivalence reduction is used to do the math, as shown in Figure 3.35. Relation C in the figure corresponds to matrix C in Figure 3.33.

$$\begin{array}{c} \\ ra_1 \\ ra_2 \end{array} \begin{array}{c} \begin{array}{ccc} ca_1 & ca_2 & ca_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 3 \end{pmatrix} \end{array}$$

Figure 3.31: Matrix A

$$\begin{array}{c} \\ rb_1 \\ rb_2 \\ rb_3 \end{array} \begin{array}{c} \begin{array}{cc} cb_1 & cb_2 \end{array} \\ \begin{pmatrix} 1 & 4 \\ 3 & 0 \\ 0 & 2 \end{pmatrix} \end{array}$$

Figure 3.32: Matrix B

$$\begin{array}{c} \\ ra_1 \\ ra_2 \end{array} \begin{array}{c} \begin{array}{cc} cb_1 & cb_2 \end{array} \\ \begin{pmatrix} 1 & 8 \\ 6 & 6 \end{pmatrix} \end{array}$$

Figure 3.33: Matrix C

```
>domain ra, ca, rb, cb intg;
>domain va, vb intg;
>relation A(ra, ca, va)<-{(1,1,1),(1,3,2),(2,2,2),(2,3,3)};
>pr A;
+-------------+-------------+-------------+
| ra          | ca          | va          |
+-------------+-------------+-------------+
| 1           | 1           | 1           |
| 1           | 3           | 2           |
| 2           | 2           | 2           |
| 2           | 3           | 3           |
+-------------+-------------+-------------+
relation A has 4 tuples
>relation B(rb, cb, vb)<-{(1,1,1),(1,2,4),(2,1,3),(3,2,2)};
>pr B;
+-------------+-------------+-------------+
| rb          | cb          | vb          |
+-------------+-------------+-------------+
| 1           | 1           | 1           |
| 1           | 2           | 4           |
| 2           | 1           | 3           |
| 3           | 2           | 2           |
+-------------+-------------+-------------+
relation B has 4 tuples
>
```

Figure 3.34: Relation Model of Matrix A and B

```
>let vab be equiv + of va*vb by ra, cb;
>C<-[ra, cb, vab] in (A [ca:ijoin:rb] B);
>pr C;
+-------------+-------------+-------------+
| ra          | cb          | vab         |
+-------------+-------------+-------------+
| 1           | 1           | 1           |
| 1           | 2           | 8           |
| 2           | 1           | 6           |
| 2           | 2           | 6           |
+-------------+-------------+-------------+
relation C has 4 tuples
>
```

Figure 3.35: Relation C: The Production of A and B

Application: Finding Area

Another application of domain algebra is finding the area of a two dimensional polygon which is represented by sequences of points [Mer88]. The calculation is based upon the Stokes' theorem:

$$\int\int_A \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\right) dxdy = \oint (Pdx + Qdy)$$

If P = 0 and Q = x, the above formula gives the area, which is:

$$area = \Sigma(x_i y_{i+1} - x_{i+1} y_i)/2$$

The simplest example is used to show how jRelix deals with this type of application. Figure 3.36 shows a relation which models a triangle using three points together with the sequence number. Figure 3.37 presents the jRelix code and result. For each tuple in the relation *triangle*, virtual domain *x'* and *y'* give the coordinates of the next points based on the sequence number. The next point of the point with the biggest sequence number, 3, is the first point of the triangle. Hence, for each point of the triangle, both the co-ordinates of the point and its next points are available. The virtual domain *area* accumulates the value computed from each tuple according to the formula:

$$(x_i y_{i+1} - x_{i+1} y_i)/2$$

The final result is projected from the relation (By hand, the area of the triangle which matches the programming result can be easily calculated).

```
>domain Seq intg;
>domain x,y real;
>relation triangle (Seq, x,y) <- {(1, -1, 0), (2,  1, 0), (3,  0, 2)};
>pr triangle;
+-------------+---------------+---------------+
| Seq         | x             | y             |
+-------------+---------------+---------------+
| 1           | -1.0          | 0.0           |
| 2           | 1.0           | 0.0           |
| 3           | 0.0           | 2.0           |
+-------------+---------------+---------------+
relation triangle has 3 tuples
>
```

Figure 3.36: Relation *triangle*

```
>let x' be fun succ of x order Seq;
>let y' be fun succ of y order Seq;
>let area be red + of (x*y'-y*x')/2;
>Area <- [area] in triangle;
>pr Area;
+---------------+
| area          |
+---------------+
| 2.0           |
+---------------+
relation Area has 1 tuple
>
```

Figure 3.37: Calculation of Area

3.6 Update

The content of a relation can be changed by applying update operations to it. Three basic operations are provided in jRelix: **add**, **delete**, and **change**. The **add** operator has the same function as incremental assignment, which appends additional tuples to the target relation according to the given condition. The **delete** operation has the same function as **djoin** in μ-joins, which removes some tuples from the target relations based on the conditions specified in the statement. The **change** operation modifies the value of specified attributes given certain conditions. Detailed description of the operations are available in [Sun00].

The syntax for update is as follows:

update *Identifier* (**add**|**delete**) *Expression*;

update *Identifier* **change** (*StatementList*)? (*UsingClause*)?;

where *Identifier* is the name of the relation that to be updated. The optional *UsingClause* is defined as follows:

using JoinOperator *Expression*

The **ijoin** operator can be omitted in the *UsingClause*.

Figure 3.38 shows a simple example. Relation *Course* and *Student* can be found in Figure 3.12.

```
>update Course change course <- "Advanced Database " using Student;
>pr Course;
+-------------+----------------------+
| ID          | course               |
+-------------+----------------------+
| 121         | Advanced Database    |
| 135         | Algorithm            |
+-------------+----------------------+
relation Course has 2 tuples
>
```

Figure 3.38: An Example of update Operation in Flat Relation

3.7 Programming Language Constructs

Many programming languages have been embedded with database facilities to enhance the power of a programming language in manipulating databases. In fact, languages for databases could provide not only query languages but also full programming facilities [Mer99]. This is provided in the jRelix system by the existing functionalities of Computation [Bak98], Event handler [Sun00], and Abstract data type [Zhe02], etc.

3.8 Distributed Data Processing

Processing remote data across the Internet is another capability of jRelix. Remote relations, remote procedure call, and security management are accessed and executed using URL-based name structure, which is the local element name prefixed with a site identifier that is the host name plus path. Users utilize the same syntax when operations are performed locally and use the equivalent URL-based name and functions when operations are performed among different jRelix systems. Thorough discussion and rich examples are available in [Wan02].

Chapter 4

Metadata Operators User Manual

This chapter describes how to use metadata operators in the jRelix system to perform related queries and programming. The syntax for operators: **relation**, **typeof**, and **self** will be given and some points for attention in using these operators will be discussed via examples. Three metadata operators, **transpose**, **quote**, and **eval**, which were first introduced into a relational database system [Mer01] [1], will be further discussed in this chapter. A new syntax for operator **transpose** will be given, the syntax for **eval** will be augmented, and the use of **eval** will be extended from domain expressions to relational expressions. The wildcard was introduced in [Yu04] as part of the path expression in relational algebra. In this chapter, more powerful functions of the wildcard will be presented, especially when it combines with recursive virtual nested domains. Finally, redefining virtual nested relations will be discussed.

The remainder of this chapter is organized as follows: Section 4.1 introduces a new operator **typeof** as well as a new type TYPE. From Section 4.2 to Section 4.6, the syntax and usages of operator **quote**, **eval**, **self**, **relation**, and **transpose** are exhibited. Section 4.7 introduces the redefining of virtual nested relations. Section 4.8 shows the usage of the wildcard and the actualization of recursive virtual nested relation. We will end this chapter with two comprehensive examples on metadata operators, attribute path and schema discovery.

[1] operators **quote** and **eval** date back as far as LISP, a high-level programming language.

4.1 Type *TYPE* and the *typeof* Operator

To support the metadata operators, a new type TYPE is allowed for attributes. It is a primitive type whose values are valid types in jRelix system. The **typeof** operator is a utility to generate the string value of the type of an attribute or the type of a domain expression. It extracts the type of an attribute of type UNIVERSAL for each tuple in a relation, because an attribute of type UNIVERSAL may have different value (the type:value pair) in each tuple in a relation. At the same time, **typeof** is also a handy utility to help understand the construction of a relation, as it can simply give the type of any attribute in a relation.

4.1.1 Initialization of Type *TYPE*

Declaring a domain to be of type TYPE is the same as declaring a domain of other types, e.g., STRING, INTEGER. The value of type TYPE can be achieved via the operation of operator **typeof** or **transpose** which will be discussed in the following sections. It also can be input in the process of relation initialization. The data of attribute of type TYPE are strings of valid types in the jRelix system.

Table 4.1 summarizes the valid types in the current system with some examples. Two categories of type are available in jRelix: atomic or primitive, and complex. Atomic type includes INTEGER, BOOLEAN, etc. Complex type includes COMPUTATION, IDLIST (relation), and UNION. The UNION type includes names of declared attributes or relations, or combinations of any types. (Details of UNION type are available in [Gu05]).

Figure 4.1 presents the declaration of domain *tp* to be of type TYPE, as well as declarations of several other attributes and a relation which will be used in this section. Figure 4.2 shows examples of successful initialization of type TYPE. If the data of type TYPE contains neither a valid type nor a declared domain name, then the initialization will fail, as show in Figure 4.3.

Note that both the data of ATTRIBUTE and TYPE are strings without quotation marks, `" "`, and both the complete form and short form of type are acceptable.

<table>
<tr><th>Category</th><th>Type</th><th>Alias or short form</th><th>Examples in jRelix</th></tr>
<tr><td>atomic</td><td>boolean</td><td>bool</td><td>true, false</td></tr>
<tr><td></td><td>short</td><td></td><td>66, 87</td></tr>
<tr><td></td><td>integer</td><td>intg</td><td>10, 99</td></tr>
<tr><td></td><td>long</td><td></td><td></td></tr>
<tr><td></td><td>float</td><td>real</td><td>3.1415926</td></tr>
<tr><td></td><td>double</td><td></td><td></td></tr>
<tr><td></td><td>string</td><td>strg</td><td>“company”</td></tr>
<tr><td></td><td>text</td><td></td><td></td></tr>
<tr><td></td><td>expression</td><td>expr</td><td></td></tr>
<tr><td></td><td>statement</td><td>stmt</td><td></td></tr>
<tr><td></td><td>attribute</td><td>attr</td><td>name</td></tr>
<tr><td></td><td>universal</td><td>univ, anytype</td><td>string:name</td></tr>
<tr><td></td><td>type</td><td></td><td>intg, bool</td></tr>
<tr><td></td><td>“(” IDList“)”</td><td></td><td>(name, website)</td></tr>
<tr><td>complex</td><td>computation“(”IDList“)”</td><td>comp</td><td>comp(A,B,C)</td></tr>
<tr><td></td><td>relation</td><td></td><td>Company(name,address)</td></tr>
<tr><td></td><td>union</td><td></td><td>intg|strg|A, Company</td></tr>
</table>

Table 4.1: Valid Types in jRelix System

```
>domain tp type;
>domain attri attribute;
>domain name, address string;
>domain forBusiness boolean;
>domain company(name,address);
>relation T(name, forBusiness, company)<-{("Visa Gold", true, {("BMO","1234 university")})};
>
```

Figure 4.1: Domain and Relation Declaration

```
>relation TP1(attri, tp) <- {(name, string), (forBusiness, bool)};
>pr TP1;
+----------------------+----------------------+
| attri                | tp                   |
+----------------------+----------------------+
| forBusiness          | bool                 |
| name                 | string               |
+----------------------+----------------------+
relation TP1 has 2 tuples
>relation TP2(tp)<-{(company),(name|company)};
>pr TP2;
+----------------------+
| tp                   |
+----------------------+
| company              |
| name|company         |
+----------------------+
relation TP2 has 2 tuples
>
```

Figure 4.2: Initialization of Type TYPE

```
>relation TP3(tp) <- {(randomtype)};
Tuple 1 of relation TP3 has invalid value for
attribute 1; message: invalid TYPE value 'randomtype'.
>
```

Figure 4.3: An Example of a Failed Initialization of Type TYPE

4.1.2 Syntax of the *typeof* Operator

The syntax for operator **typeof** is as follows

typeof <*domain_expression*>

the *domain_expression* could be a name of an attribute which has already been declared, or a horizontal or vertical domain expression. Both actual and virtual attribute can be the operand. If the operand is the name of an attribute, then the attribute must be declared before reference. Otherwise, the operation will fail and error messages will be given out. The result of the **typeof** operator is the complete form of a type in jRelix.

4.1.3 Examples

As mentioned before, the **typeof** operator is especially useful to extract the type from an attribute of type UNIVERSAL so that the type information can be presented independently from its value in a relation. Figure 4.4 is relation *U* which contains an attribute *val* of type UNIVERSAL.

```
>domain val univ;
>relation U(val) <-{(boolean:true),(string:"Visa Gold")};
>pr U;
+----------------------+
| val                  |
+----------------------+
| boolean:true         |
| string:Visa Gold     |
+----------------------+
relation U has 2 tuples
>
```

Figure 4.4: A Relation Contains Attribute of Type UNIVERSAL

Query: Get all types appearing in attribute *val* in relation *U*.

The result is shown in Figure 4.5.

Similarly, when the type of **typeof** operand is ATTRIBUTE, the result is the type of the attribute, which is the value of the operand. Figure 4.6 shows this case.

```
>let tpof be typeof val;
>UTP<-[tpof] in U;
>pr UTP;
+----------------------+
| tpof                 |
+----------------------+
| boolean              |
| string               |
+----------------------+
relation UTP has 2 tuples
>
```

Figure 4.5: An Example of **typeof** on a UNIVERSAL Type Attribute

```
>let tpofatr be typeof attri;
>TTP <- [attri, tpofatr] in TP1;
>pr TTP;
+----------------------+----------------------+
| attri                | tpofatr              |
+----------------------+----------------------+
| forBusiness          | boolean              |
| name                 | string               |
+----------------------+----------------------+
relation TTP has 2 tuples
>
```

Figure 4.6: Examples of **typeof** with Operand of Type ATTRIBUTE

Here is another example showing the type of an attribute of type UNION. Figure 4.7 includes relation *UC* which has attribute *cust*, a UNION type attribute. Figure 4.8 shows the result of **typeof** *cust*.

Operator **typeof** also can be performed on an attribute or an expression of other types and the production is a constant, as shown in Figure 4.9.

In addition to the above discussion, operator **typeof** can also return the type of an **eval** expression, e.g., **let** *tpeval* **be eval** *A*. When the virtual domain *tpeval* is actualized, it has the type of the attribute, which is the data of attribute *A*. An example will be shown in Section 4.3.2, after the **eval** operator is introduced.

Note that, although the primitive types can be displayed in their short forms, such as "strg" for "string" and "bool" for "boolean", the complete form is used as the result of **typeof**. The reason is, as we discussed, that the names of attributes and relations can also be types and they do not have short forms, so we uniformly use complete form for all types.

```
>domain name, address strg;
>domain company(name,address);
>domain cust name|company;
>relation UC(cust) <-{(name:"Dell"),
                      (company:{("HD","Milton 1000"),("JJ","peel 300")})};
>pr UC;
+----------------------+
| cust                 |
+----------------------+
| company:1            |
| name:Dell            |
+----------------------+
relation UC has 2 tuples
>pr .company;
+----------------------+----------------------+----------------------+
| .id                  | name                 | address              |
+----------------------+----------------------+----------------------+
| 1                    | HD                   | Milton 1000          |
| 1                    | JJ                   | peel 300             |
+----------------------+----------------------+----------------------+
relation .company has 2 tuples
>
```

Figure 4.7: A Relation Contains Attribute of Type UNION

```
>let typeuc be typeof cust;
>UCTP <-[typeuc] in UC;
>pr UCTP;
+----------------------+
| typeuc               |
+----------------------+
| company              |
| name                 |
+----------------------+
relation UCTP has 2 tuples
>
```

Figure 4.8: An Example of **typeof** on a UNION Type Attribute

```
>let tpbusiness be typeof forBusiness;
>let tpcompany be typeof company;
>TPof1<-[tpbusiness,tpcompany] in T;
>pr TPof1;
+----------------------+----------------------+
| tpbusiness           | tpcompany            |
+----------------------+----------------------+
| boolean              | relation             |
+----------------------+----------------------+
relation TPof1 has 1 tuple
>
>let tpname be typeof name;
>let adr be [address] in company;
>let tpvdom be typeof red ujoin of adr;
>TPof2<-[tpname, tpvdom] in T;
>pr TPof2;
+----------------------+----------------------+
| tpname               | tpvdom               |
+----------------------+----------------------+
| string               | relation             |
+----------------------+----------------------+
relation TPof2 has 1 tuple
>
```

Figure 4.9: Examples of **typeof** Operator with Constant Result

4.2 The *quote* Operator

The **quote** operator converts any attribute name to attribute metadata [Mer01]. It suppresses the evaluation of an attribute in context where evaluation would ordinarily take place[Mer03]. And it can appear in both domain algebra expressions and relational algebra expressions. Details of syntax and examples of the **quote** operator in domain algebra can be found in [Roz02]. In this section we review the syntax and give an example showing the usage of the **quote** operator in relational algebra expression.

4.2.1 Syntax

The syntax for operator **quote** is as follows:

$$\textbf{quote}\ <\!\textit{attribute\_name}\!>$$

attribute_name is the name of a declared attribute. It could be an actual domain or a virtual domain of any type. The result of the **quote** operator is the name of the attribute defined by *attribute_name* and the type of the result is ATTRIBUTE.

4.2.2 Examples

Query: Get attribute *forBusiness* and its type in relation *TP1*.

Relation *TP1* can be found in Figure 4.2. Here we need to compare the value of domain *attri* to a given condition *forBusiness*. Intuitively, a query would be written as:

$$\textbf{where}\ \textit{attri} = \textit{forBusiness}\ \textbf{in}\ \textit{TP1};$$

But *forBusiness* is a declared attribute, and it will be evaluated in each tuple in actualization. To avoid the evaluation, the query needs to be modified to

$$\textbf{where}\ \textit{attri} = \textbf{quote}\ \textit{forBusiness}\ \textbf{in}\ \textit{TP1};$$

In the above query, with the **quote** operation, the right hand side of the comparison has a constant value *forBusiness* in each tuple during the comparison process. So that the tuples with data of *forBusiness* in attribute *attri* are selected to the target relation. The query and the result are shown in Figure 4.10.

```
>TpName <- where attri = quote forBusiness in TP1;
>pr TpName;
+----------------------+----------------------+
| attri                | tp                   |
+----------------------+----------------------+
| forBusiness          | bool                 |
+----------------------+----------------------+
relation TpName has 1 tuple
>
```

Figure 4.10: Examples of **quote** operator

4.3 The *eval* Operator

The **eval** is a unary operator used to evaluate the attribute whose name is the data of the **eval** operand in the corresponding tuple in a relation. The operand of **eval** can be a unary relation or an expression resulting in a unary relation. In both cases the unary relation must have an attribute of type ATTRIBUTE [Roz02]. In this section, the syntax of **eval** operator will be extended to allow the operand to be an attribute of type ATTRIBUTE. In following subsections, we will show this new syntax and explore its different usages.

4.3.1 Syntax of the *eval* Operator

The new syntax for operator **eval** is as follows:

eval <*attribute_name*>

or

eval `"("` **attribute** `")"` <*attribute_name*>

the *attribute_name* is the name of a declared attribute of type ATTRIBUTE in the first production, while it is a name of an attribute of any type in the second production. The **attribute** in the second production is used to cast the attribute type to type ATTRIBUTE. The result of **eval** operation are values of type UNIVERSAL, this is because the type of the attribute is ATTRIBUTE, whose data type are variable.

4.3.2 Examples

eval in Relational Expression

In Figure 4.11, relation *R0* contains domains *A*, *Y*, and *Z*. Domain *A* is of type ATTRIBUTE, and domain *Y* and *Z* are of type INTEGER. The selection operation in Figure 4.12 compares the data in domain *Y* and the result of **eval** *A*, which is a data of an attribute that is a data itself of domain *A* in each tuple of relation *R0*. All those tuples, which have the *true* value of the comparison, are selected to be the tuples in result relation *S0*. Note that the type of *Y* is INTEGER, while the result of **eval** *A* is of the type UNIVERSAL. So we need to do a **cast** to make the type of two comparison operands equal. The result of **eval** operation is thus cast to INTEGER.

```
>domain A attr;
>domain X,Y,Z intg;
>relation R0(A,Y,Z) <-{(Y,1,1),(Y,2,2),(Z,1,1),(Z,1,2)};
>pr R0;
+----------------------+-------------+-------------+
| A                    | Y           | Z           |
+----------------------+-------------+-------------+
| Y                    | 1           | 1           |
| Y                    | 2           | 2           |
| Z                    | 1           | 1           |
| Z                    | 1           | 2           |
+----------------------+-------------+-------------+
relation R0 has 4 tuples
>
```

Figure 4.11: Relation *R0*

```
>S0 <- where Y = (intg) eval A in R0;
>pr S0;
+----------------------+-------------+-------------+
| A                    | Y           | Z           |
+----------------------+-------------+-------------+
| Y                    | 1           | 1           |
| Y                    | 2           | 2           |
| Z                    | 1           | 1           |
+----------------------+-------------+-------------+
relation S0 has 3 tuples
>
```

Figure 4.12: **eval** in Relational Algebra

Let's take a close look at the process. For the first tuple in relation *R0*, the value of the left hand side operand of comparison is domain *Y* 's value, which is 1. For the right hand side, **eval** *A* is to evaluate the value of domain *A* in the first tuple, which is *Y*, and the result of evaluation is 1. So the result of **eval** *A* is 1. As both sides of the comparison operation have value 1, the result of comparison is *true*. Therefore the first tuple is selected as one of the tuples of result relation. The same evaluation goes to the second tuple. For the third tuple, in the left hand side, the value of domain Y is 1, while the value of domain A is Z. **eval** A in this case gets the value of domain Z in the current tuple, which is also 1. So the result of comparison is *true* and the tuple is selected. For the fourth tuple, the value of the left hand side is 1. In the right hand side, the value of domain A is Z. As domain Z has the value 2 in the fourth tuple, the result of **eval** A in this tuple is 2. So the two sides of the comparison are not equal and the tuple is not selected.

In short, when we need a result from the **eval** operation to act as the operand of a comparison, or a value to be assigned to another attribute as we will show in following section, the result of the **eval** operator in each tuple in the relation is decided by the value of the domain whose name is the value of the **eval** operand in the same tuple.

eval in Left Hand Side of Domain Declaration

We can also assign values to the attributes whose name is data of **eval** operand in domain declaration. Under this circumstance, the attribute whose name is the data of the **eval** operand in a tuple will be assigned the declared value. In Figure 4.14, the **let** statement declares the attributes resulting from attribute B to be the sum of X, Y, and Z(Relation *R1* is defined in Figure 4.13).

```
>domain B attr;
>domain X',Y',Z' intg;
>relation R1(A,B,X,Y,Z) <- { (X, X',1,2,3),(Y,Y',4,5,6),(Z,Z',7,8,9)};
>pr R1;
+---------------------+---------------------+------------+------------+------------+
| A                   | B                   | X          | Y          | Z          |
+---------------------+---------------------+------------+------------+------------+
| X                   | X'                  | 1          | 2          | 3          |
| Y                   | Y'                  | 4          | 5          | 6          |
| Z                   | Z'                  | 7          | 8          | 9          |
+---------------------+---------------------+------------+------------+------------+
relation R1 has 3 tuples
>
```

Figure 4.13: Relation *R1*

Because the type of X, Y, and Z are INTEGER, note that the type of "$X+Y+Z$" is also INTEGER. So we should make sure that the type of all attributes *X'*, *Y'*, and

```
>let eval B be X+Y+Z;
>S1<-[[B]in R1] in R1;
>pr S1;
+-------------+-------------+-------------+
| X'          | Y'          | Z'          |
+-------------+-------------+-------------+
| dc          | dc          | 24          |
| dc          | 15          | dc          |
| 6           | dc          | dc          |
+-------------+-------------+-------------+
relation S1 has 3 tuples
>
```

Figure 4.14: **eval** Declaration (in the left)

Z', is INTEGER. Otherwise the actualization of these attributes would fail during the stage of type validation checking. Figure 4.15 shows an example of this case. Generally, users should ensure that the type of all those declared attributes which are data of the attribute on the left hand side of **let** statement match the type of the definition on the right hand side.

```
>domain X'',Y'',Z'' univ;
>relation R1'(A,B,X,Y,Z) <- { (X, X'',1,2,3),(Y,Y'',4,5,6),(Z,Z'',7,8,9)};
>pr R1';
+---------------------+---------------------+-------------+-------------+-------------+
| A                   | B                   | X           | Y           | Z           |
+---------------------+---------------------+-------------+-------------+-------------+
| X                   | X''                 | 1           | 2           | 3           |
| Y                   | Y''                 | 4           | 5           | 6           |
| Z                   | Z''                 | 7           | 8           | 9           |
+---------------------+---------------------+-------------+-------------+-------------+
relation R1' has 3 tuples>
>let eval B be X+Y+Z;
>S1' <-[[B] in R1'] in R1';
error: new type of 'X''' is 'integer', but the old type is 'universal'.
>
```

Figure 4.15: An Example of **eval** Declaration Type Mismatch

The projection "[*B*] **in** *R1*", which is the projector in the projection "[[*B*] **in** *R1*] **in** *R1* ", leads to a set of attributes in the result relation *S1*.

eval in Right Hand Side of Domain Declaration

The result of **eval** operation also can be used to define a virtual domain, as shown in Figure 4.17(Relation *R2* is declared and initialized in Figure 4.16). Note that the virtual domain *P* is declared to be of type UNIVERSAL.

```
>domain C attr;
>domain E strg;
>domain F bool;
>domain X'',E'',F'' univ;
>relation R2(A,C,X,E,F) <-{(X,X'',1,"son",false),(E,E'',4,"sun",false),(F,F'',7,"song", true)};
>pr R2;
+----------------------+----------------------+-------------+----------------------+--------+
| A                    | C                    | X           | E                    | F      |
+----------------------+----------------------+-------------+----------------------+--------+
| E                    | E''                  | 4           | sun                  | false  |
| F                    | F''                  | 7           | song                 | true   |
| X                    | X''                  | 1           | son                  | false  |
+----------------------+----------------------+-------------+----------------------+--------+
relation R2 has 3 tuples
>
```

Figure 4.16: Relation *R2*

```
>let P be eval A;
>S2 <- [P] in R2;
>pr S2;
+----------------------+
| P                    |
+----------------------+
| boolean:true         |
| integer:1            |
| string:sun           |
+----------------------+
relation S2 has 3 tuples
>
```

Figure 4.17: **eval** Declaration (in the right)

eval in Both Side of Domain Declaration

In the following example in Figure 4.18, the **eval** operator appears at both sides of the declaration statement. The attributes on the data set of domain C should be of type UNIVERSAL.

```
>let eval C be eval A;
>S3 <-[[C] in R2] in R2;
>pr S3;
+----------------------+----------------------+----------------------+
| E''                  | F''                  | X''                  |
+----------------------+----------------------+----------------------+
| dc                   | boolean:true         | dc                   |
| dc                   | dc                   | integer:1            |
| string:sun           | dc                   | dc                   |
+----------------------+----------------------+----------------------+
relation S3 has 3 tuples
>
```

Figure 4.18: **eval** Declaration (in both side)

eval as Operand of *typeof*

As mentioned in Section 4.1.3, **typeof** can also apply to the **eval** operator. Here is an example to demonstrate such a case. The query in Figure 4.19 gets types of all attributes of data set of domain A in relation $R2$.

```
>let tpeval be typeof eval A;
>TpEval<-[tpeval] in R2;
>pr TpEval;
+----------------------+
| tpeval               |
+----------------------+
| boolean              |
| integer              |
| string               |
+----------------------+
relation TpEval has 3 tuples
>
```

Figure 4.19: **eval** as an Operand of **typeof**

4.4 The *self* Operator

The **self** operator returns the name of the relation in which it is actualized. It can appear in both normal expressions and path expressions.

4.4.1 Syntax

The syntax for operator **self** is as follows:

self

As **self** is a system reserved word, it is not supposed to be used as the name of a domain or a relation. Otherwise, an error message will be displayed and any operation involved will fail. If the relation in which **self** is actualized is an unnamed relation, then the name of the next higher-level named relation will be returned. But if **self** is actualized in a top-level relation which is not named, the system will give a warning and the **self** will be assigned a "null" value.

4.4.2 Examples

An Example: A Relation of Products

This section describes an example, which will be used in the following sections. We will give a brief introduction of the meaning of each attribute involved for the purpose of understanding the design of the database. Details of these products is beyond the scope of this thesis, but can be found in their website listed in Table 4.2.

This example is a product information database of hardware/chip programming devices, as shown in Table 4.2. It is a nested relation with three levels: the top-level relation has three attributes: attribute *family* which categories the product according to their different usages is an attribute of type STRING. Attribute *Lscale*, which gives the capacity of the family, is an attribute of type BOOLEAN. Types of these two attributes are both scalar types. The third attribute, *device*, which tells the information of specific devices in each family, however, is an attribute of type RELATION. It is designed with two attributes giving the device name and the information of the company which can provide the device. The nested relation *company* contains attributes of the company name and website. Definition of domains and relation initialization are shown in Figure 4.20.

products

(family	Lscale	device		)
		(name	company	)
			(name	website)
CPLDs	true	EPM 3512A	Altera	www.altera.com
		MACH 4000C	Lattice	www.latticesemi.com
FPGA	true	XC3s 200	Xilinx	www.xilinx.com
SPLD	false	GAL	Lattice	www.latticesemi.com
			Atmel	www.atmel.com

Table 4.2: The Display form of Relation *Products*

```
>domain family strg;
>domain Lscale bool;
>domain name, website strg;
>domain company(name, website);
>domain device(name, company);
>relation products(family, Lscale, device)<-
 {("CPLD",true, {("EPM 3512A ",{("Altera","www.altera.com")}),
                ("MACH 4000C",{("Lattice", "www.latticesemi.com ")})}),
  ("FPGA",true, {("XC3s 200",   {("Xilinx", "www.xilinx.com")} )}),
  ("SPLD",false,{("GAL",        {("Lattice", "www.latticesemi.com"),
                               ("Atmel", "www.atmel.com")})})
 };
>pr products;
+----------------------+--------+----------------------+
| family               | Lscale | device               |
+----------------------+--------+----------------------+
| CPLD                 | true   | 1                    |
| FPGA                 | true   | 4                    |
| SPLD                 | false  | 6                    |
+----------------------+--------+----------------------+
relation products has 3 tuples
>pr .device;
+----------------------+----------------------+----------------------+
| .id                  | name                 | company              |
+----------------------+----------------------+----------------------+
| 1                    | EPM 3512A            | 2                    |
| 1                    | MACH 4000C           | 3                    |
| 4                    | XC3s 200             | 5                    |
| 6                    | GAL                  | 7                    |
+----------------------+----------------------+----------------------+
relation .device has 4 tuples
>pr .company;
+----------------------+----------------------+----------------------+
| .id                  | name                 | website              |
+----------------------+----------------------+----------------------+
| 2                    | Altera               | www.altera.com       |
| 3                    | Lattice              | www.latticesemi.com  |
| 5                    | Xilinx               | www.xilinx.com       |
| 7                    | Atmel                | www.atmel.com        |
| 7                    | Lattice              | www.latticesemi.com  |
+----------------------+----------------------+----------------------+
relation .company has 5 tuples
>
```

Figure 4.20: Relation *Products*

Query Examples

The **self** keyword can either be used directly acting as a virtual domain or used to declare another virtual domain. In both cases it will be actualized to the name of the relation it is actualized in. Figure 4.21, Figure 4.22, and Figure 4.23 give some examples of the usage of **self** in normal expression. In Figure 4.21, **self** returns the name of top-level relations and the name of nested relation. In Figure 4.22 **self** returns the name of a virtual nested relation. Figure 4.23 shows the result if **self** is tried to actualize in a top level unnamed relation.

```
>SRProducts <- [self, family] in products;
>pr SRProducts;
+----------------------+----------------------+
| self                 | family               |
+----------------------+----------------------+
| products             | CPLD                 |
| products             | FPGA                 |
| products             | SPLD                 |
+----------------------+----------------------+
relation SRProducts has 3 tuples
>let srel be self;
>let srdevice be [srel] in device;
>SRDevice <- [red ujoin of srdevice] in products;
>pr SRDevice;
+----------------------+
| srel                 |
+----------------------+
| device               |
+----------------------+
relation SRDevice has 1 tuple
>
```

Figure 4.21: **self** being Actualized in Actual Relations

```
>let nm be [name] in device;
>let srvir be [self] in nm;
>SRVir <- [red ujoin of srvir] in products;
>pr SRVir;
+----------------------+
| self                 |
+----------------------+
| nm                   |
+----------------------+
relation SRVir has 1 tuple
>
```

Figure 4.22: **self** being Actualized in an Virtual Relation

As we mentioned, the **self** keyword can also be used in path expression. It is particularly useful in *finding path* queries. Examples will be shown in Section 4.6.2 and Section 4.9, and applications will be available in the next chapter.

```
>SRnull <-[self] in ([family] in products);
Warning: self can not be actualized on an unnamed top level relation.
>
```

Figure 4.23: A Failed Operation of **self**

4.5 The *Relation* Operator

The **relation** operator is used to group a set of attributes into a nested relation. It is an operator in domain algebra and is similar to nested relation declaration to some extent. For instance,

let `X` **be relation**`(A, B);`

can be compared with

domain `X(A, B);`

where X is the declared nested relation, and A and B are attributes of X. Although both the **relation** operator and the relation declaration create nested relations, the functionality of the former is more than the latter. The **relation** operator creates a singleton relation (i.e., a relation that has only one tuple) for each tuple of the source relation in which the operator actualized in. Its value depends on the value of the attributes in the current tuple in the source relation, while the result of nested relation declaration creates an arbitrary relation, whose value relies on the relation initialization. Also, while the domain declaration creates an actual domain, the **relation** operator generates a virtual domain. Furthermore, the virtual domain can be named(e.g., X in the above example) or unnamed (e.g., **let** *NameAdr* **be relation**(*name*) **ijoin** *address*; where *address* is a nested relation), but the declaration of nested relation only creates a named attribute.

4.5.1 Syntax

The syntax for **relation** operator is as follows:

relation `"(" <attribute_name> (,<attribute_name>)* ")"`

where **relation** is the keyword followed by one or more attributes of this relation. The attributes are delimited inside round parenthesis. Regardless whether actual or virtual, these attributes must be those that have already been declared and at least one attribute must be specified. Otherwise the system will give out error messages.

Top level relation operation such as:

`R <-` **relation**`(A)` **ujoin** `Q`

is **not valid**, because attribute A has no source relation in the query, therefore no data is available for the relation operation.

4.5.2 Examples

Query: Get product family and its device name in relation *products*.

The solution for the query is shown in Figure 4.24.

```
>let dname be [name] in device;
>let fmname be relation(family) ijoin dname;
>FName <- products/fmname;
>pr FName;
+----------------------+----------------------+
| family               | name                 |
+----------------------+----------------------+
| CPLD                 | EPM 3512A            |
| CPLD                 | MACH 4000C           |
| FPGA                 | XC3s 200             |
| SPLD                 | GAL                  |
+----------------------+----------------------+
relation FName has 4 tuples
>
```

Figure 4.24: An Example of **relation** Operator

In the above query, **relation** (*family*) groups the attribute *family* into a nested relation. It is a singleton relation nested in relation *products*. The **ijoin** of the relation and the nested relation *dname*:

let *fmname* **be relation**(*family*) **ijoin** *dname*;

gives the Cartesian products in each tuple of the upper level relation, *products*, as shown in Table 4.3, where all three virtual nested relations, **relation**(*family*), *dname*, and *fmname* are given. Suppose we give a name *Rf* to **relation**(*family*) and we hide the data of nested relation *company* due to the limited space.

If the **relation** operator has only one domain, as in our example, then the keyword **relation** can be omitted. The above query can be rewritten as in Figure 4.25.

products

(family	Lscale	device (name	) company) ( )	Rf (family)	dname (name)	fmname (family	 name)
CPLDs	true	EPM 3512A MACH 4000C		CPLDs	EPM 3512A MACH 4000C	CPLDs CPLDs	EPM 3512A MACH 4000C
FPGA	true	XC3s 200		FPGA	XC3s 200	FPGA	XC3s 200
SPLD	false	GAL		SPLD	GAL	SPLD	GAL

Table 4.3: The Cartesian Products of *fmname* in Relation *products*

```
>let dname be [name] in device;
>let fmname be family ijoin dname;
>FName <- products/fmname;
>pr FName;
+----------------------+----------------------+
| family               | name                 |
+----------------------+----------------------+
| CPLD                 | EPM 3512A            |
| CPLD                 | MACH 4000C           |
| FPGA                 | XC3s 200             |
| SPLD                 | GAL                  |
+----------------------+----------------------+
relation FName has 4 tuples
>
```

Figure 4.25: An Example of **relation** Operator Using Short Form

4.6 The *Transpose* Operator

transpose is an operator creating a nested relation, which contains the data of attributes in each tuple of a relation. The data includes the attributes' name, type, and the value. The operator gives all scalar attributes and only scalar attributes. No nested attribute will appear in the result.

4.6.1 Syntax

The syntax for the **transpose** operator is as follows:

transpose `"("` `<`*attribute_name*`>` `(,<`*attribute_name*`>)*` `")"`

where **transpose** is the keyword, followed by the attributes of the result relation. The attributes in the attribute list must have type ATTRIBUTE, TYPE, or UNIVERSAL/ANYTYPE in any order. The values of attributes of type ATTRIBUTE are the names of all scalar attributes; the values of attributes of type TYPE are any valid type in the jRelix system (please refer to Section 6.2.1 for valid types), while the values of attributes of type UNIVERSAL are "type: value" pairs in which the type is type of the attribute and the value is the value of the attribute in the corresponding tuple. At least one attribute must be defined after the **transpose** operator and the maximum number is three, representing no more than one of the three types, ATTRIBUTE, TYPE, and UNIVERSAL/ANYTYPE, respectively. The attributes used must be those that have already been declared and are delimited inside round parenthesis. An error will be given out if any of the above conditions is not satisfied.

4.6.2 Examples

A General Example

Figure 4.26 shows the result of the **transpose** operation on relation *products* (in Figure 4.20). As we can see, the **transpose** generates a nested relation *trans* and the number of tuples of each relation corresponding to each tuple of the relation in which it is actualized (*products* in this example) is determined by the number of the scalar attributes of the relation *products*, which is two.

```
>domain attri attr;
>domain tp type;
>domain val univ;
>let trans be transpose(attri,tp,val);
>TransProj<-[family, trans] in products;
>pr TransProj;
+----------------------+----------------------+
| family               | trans                |
+----------------------+----------------------+
| CPLD                 | 13                   |
| FPGA                 | 15                   |
| SPLD                 | 17                   |
+----------------------+----------------------+
relation TransProj has 3 tuples
>pr .trans;
+----------------------+----------------------+----------------------+----------------------+
| .id                  | attri                | tp                   | val                  |
+----------------------+----------------------+----------------------+----------------------+
| 13                   | family               | string               | string:CPLD          |
| 13                   | Lscale               | boolean              | boolean:true         |
| 15                   | family               | string               | string:FPGA          |
| 15                   | Lscale               | boolean              | boolean:true         |
| 17                   | family               | string               | string:SPLD          |
| 17                   | Lscale               | boolean              | boolean:false        |
+----------------------+----------------------+----------------------+----------------------+
relation .trans has 6 tuples
>
```

Figure 4.26: An Example of **transpose** Operator Result in Nested Relation

transpose Defined on Different Attributes

As we mentioned, the attributes of the relation resulting from the **transpose** operator can be attributes of any of the three types, TYPE, ATTRIBUTE, and UNIVERSAL, as long as no type appears more than once. In Figure 4.27, transpose is defined on attributes *attri* and *tp*. All scalar attributes name and their types in relation *products* are obtained. We use the level lifting technique, which was discussed in [Zhe02], in this query to avoid the inconvenience of referring to the dot relation of the virtual nested relation.

```
>let tatp be transpose(attri, tp);
>TATP<-[red ujoin of tatp] in products;
>pr TATP;
+----------------------+----------------------+
| attri                | tp                   |
+----------------------+----------------------+
| Lscale               | boolean              |
| family               | string               |
+----------------------+----------------------+
relation TATP has 2 tuples
>
```

Figure 4.27: **transpose** Defined on Attributes of Type ATTRIBUTE and TYPE

Figure 4.28 and Figure 4.29 show examples of **transpose** defined on the attribute of type ATTRIBUTE and TYPE, respectively. In Figure 4.29, we use the short form:

TTP <- `products / ttp;`

instead of

TTP <- **[red ujoin of** `ttp`**] in** `products;`

in the query.

```
>let tattr be transpose(attri);
>TATTR <-[red ujoin of tattr] in products;
>pr TATTR;
+----------------------+
| attri                |
+----------------------+
| Lscale               |
| family               |
+----------------------+
relation TATTR has 2 tuples
>
```

Figure 4.28: **transpose** Defined on Attributes of Type ATTRIBUTE

```
>let ttp be transpose(tp);
>TTP<- products/ttp;
>pr TTP;
+----------------------+
| tp                   |
+----------------------+
| boolean              |
| string               |
+----------------------+
relation TTP has 2 tuples
>
```

Figure 4.29: **transpose** Defined on Attributes of Type TYPE

transpose Selected Attributes

The **transpose** operation returns all data of scalar attributes. If we need to transpose only some particular attributes, we need to do a selection from the result of **transpose**. In our example, we can select those tuples, which have the value of attribute *attri* that equals the select value, say *family*. The input condition *family* should be a string, but *attri* is the type of ATTRIBUTE, and we are not able to compare a string "*family*" to the value of the domain *attri*. One solution is to perform a **cast** operation on the domain *attri* to get a value of type STRING and then compare it with "*family*". Figure 4.30 depicts the solution. The other solution is to compare the value of the domain *attri* with the attribute *family* using the **quote** operator to forbid the evaluation. Figure 4.31 shows such a case. We can see that operator **quote** performed on the attribute name *family* contributes one operand of the comparison operation and the value of the attribute *attri* does the other, both of which are of the type ATTRIBUTE. Therefore, the tuple where the value of the attribute *attri* is equal to the result of "**quote** *family*", which is *family*, becomes the result of the query.

```
>let trans be transpose(attri,tp,val);
>TransSelCast<- where (string)attri = "family" in products/trans;
>pr TransSelCast;
+----------------------+----------------------+----------------------+
| attri                | tp                   | val                  |
+----------------------+----------------------+----------------------+
| family               | string               | string:CPLD          |
| family               | string               | string:FPGA          |
| family               | string               | string:SPLD          |
+----------------------+----------------------+----------------------+
relation TransSelCast has 3 tuples
>
```

Figure 4.30: **transpose** Part of Attributes with **cast** Operator

```
>let trans be transpose(attri,tp,val);
>TransSelQuote<-where attri = quote family in products/trans;
>pr TransSelQuote;
+----------------------+----------------------+----------------------+
| attri                | tp                   | val                  |
+----------------------+----------------------+----------------------+
| family               | string               | string:CPLD          |
| family               | string               | string:FPGA          |
| family               | string               | string:SPLD          |
+----------------------+----------------------+----------------------+
relation TransSelQuote has 3 tuples
>
```

Figure 4.31: **transpose** Part of Attributes with **quote** Operator

***transpose*: More Examples**

Furthermore, we can **transpose** all attributes on a relation that itself is the result of a **transpose** operation. As in Figure 4.32, the relation *TransTrans* is the result of **transpose** on relation *Trans*, the result of **transpose** on relation *products*. And relation *TransTransTrans* is the result of transpose on *TransTrans*.

```
>Trans<- products/trans;
>pr Trans;
+----------------------------+----------------------------+----------------------------+
| attri                      | tp                         | val                        |
+----------------------------+----------------------------+----------------------------+
| Lscale                     | boolean                    | boolean:false              |
| Lscale                     | boolean                    | boolean:true               |
| family                     | string                     | string:CPLD                |
| family                     | string                     | string:FPGA                |
| family                     | string                     | string:SPLD                |
+----------------------------+----------------------------+----------------------------+
relation Trans has 5 tuples
>TransTrans<- Trans/trans;
>pr TransTrans;
+----------------------------+----------------------------+----------------------------+
| attri                      | tp                         | val                        |
+----------------------------+----------------------------+----------------------------+
| attri                      | attribute                  | attribute:Lscale           |
| attri                      | attribute                  | attribute:family           |
| tp                         | type                       | type:boolean               |
| tp                         | type                       | type:string                |
| val                        | universal                  | universal:boolean:false    |
| val                        | universal                  | universal:boolean:true     |
| val                        | universal                  | universal:string:CPLD      |
| val                        | universal                  | universal:string:FPGA      |
| val                        | universal                  | universal:string:SPLD      |
+----------------------------+----------------------------+----------------------------+
relation TransTrans has 9 tuples
>
```

Figure 4.32: **transpose** on Relations Resulted from Transpose Operation (1)

The transpose result of type UNIVERSAL results in form “universal:type:value”, for in the “type:value” pair of value of type UNIVERSAL, the type is UNIVERSAL, and the value is itself the "type:value" pair as in Figure 4.32 relation *TransTrans*. But the further transpose operation will not generate repetitive *universal*, The example in Figure 4.33 demonstrates this case.

```
>TransTransTrans<- TransTrans/trans;
>pr TransTransTrans;
+---------------------------+---------------------------+---------------------------+
| attri                     | tp                        | val                       |
+---------------------------+---------------------------+---------------------------+
| attri                     | attribute                 | attribute:attri           |
| attri                     | attribute                 | attribute:tp              |
| attri                     | attribute                 | attribute:val             |
| tp                        | type                      | type:attribute            |
| tp                        | type                      | type:type                 |
| tp                        | type                      | type:universal            |
| val                       | universal                 | universal:attribute:Lscale |
| val                       | universal                 | universal:attribute:family |
| val                       | universal                 | universal:boolean:false   |
| val                       | universal                 | universal:boolean:true    |
| val                       | universal                 | universal:string:CPLD     |
| val                       | universal                 | universal:string:FPGA     |
| val                       | universal                 | universal:string:SPLD     |
| val                       | universal                 | universal:type:boolean    |
| val                       | universal                 | universal:type:string     |
+---------------------------+---------------------------+---------------------------+
relation TransTransTrans has 15 tuples
>
```

Figure 4.33: **transpose** on Relations Resulted from Transpose Operation (2)

A Simple *finding path* Query

Figure 4.34 shows a simple query presenting the attribute path in relation *products*. The **transpose**(*attri*) creates a nested relation defined on the domain *attri* and the values of the relation is scalar attribute names in the relation *products*. The **self** in the virtual domain *path* is actualized to be *products*, because the relation **transpose**(*attri*) has no name, and so the name of upper level relation is obtained.

```
>domain attri attribute;
>let path be self/attri;
>PathProd<-([red ujoin of ([path] in transpose(attri))] in products);
>pr PathProd;
+---------------------+
| path                |
+---------------------+
| products/Lscale     |
| products/family     |
+---------------------+
relation PathProd has 2 tuples
>
```

Figure 4.34: A Simple Query for *Finding Path*

R

(A	V)
	B
1	x
	y
2	z

Table 4.4: The Display Form of Relation *R*

So far, we have presented the syntax for metadata operators and their usages. Before moving on to the next chapter, where applications of the metadata operators discussed in this chapter will be presented, two new functions will be exhibited in the following sections.

4.7 Redefining Virtual Nested Relation

Declaring a virtual nested relation will create a virtual domain, as well as a dot relation, where the actualized data of the nested relation is stored. Redefinition of the virtual nested relation will lead to two different results. In the following subsections, two such cases will be examined.

4.7.1 Redefining Keeping the Same Attributes

If the redefined domain has the same attributes as it does before redefining, the data of the new virtual relation is appended to the original dot relation. The following query depicts the case.

Table 4.4 is the display form of relation *R*, which contains a nested relation *V* defined on attribute *B*. Figure 4.35 shows the print form of *R*. The real data of nested relation *V* is stored in relation *.V* and is shown, as well. Figure 4.36 shows a source relation *S0* and the redefinition of nested relation *V*. *T0* is a target relation where the virtual nested relation *V* is actualized. As *V* has the same attribute *B* as before, its actualized data is appended to the relation *.V*, shown in Figure 4.36.

```
>domain A intg;
>domain B strg;
>domain V(B);
>relation R(A,V) <-{(1,{("x"),("y")}),(2,{("z")})};
>pr R;
+-------------+----------------------+
| A           | V                    |
+-------------+----------------------+
| 1           | 101                  |
| 2           | 102                  |
+-------------+----------------------+
relation R has 2 tuples
>pr .V;
+----------------------+----------------------+
| .id                  | B                    |
+----------------------+----------------------+
| 101                  | x                    |
| 101                  | y                    |
| 102                  | z                    |
+----------------------+----------------------+
relation .V has 3 tuples
>
```

Figure 4.35: Relation R and Nested Relation $.V$

```
>relation S0(B) <-{("9")};
>pr S0;
+----------------------+
| B                    |
+----------------------+
| 9                    |
+----------------------+
relation S0 has 1 tuple
>let V be relation(B);
>T0<- [B,V] in S0;
>pr T0;
+----------------------+----------------------+
| B                    | V                    |
+----------------------+----------------------+
| 9                    | 103                  |
+----------------------+----------------------+
relation T0 has 1 tuple
>pr .V;
+----------------------+----------------------+
| .id                  | B                    |
+----------------------+----------------------+
| 101                  | x                    |
| 101                  | y                    |
| 102                  | z                    |
| 103                  | 9                    |
+----------------------+----------------------+
relation .V has 4 tuples
>
```

Figure 4.36: Redefined V with the Same Attributes

4.7.2 Redefining with Different Attributes

On the other hand, if the new defined nested relation has attributes different from the original one, a new domain as well as a new dot relation will be generated and all references to the redefined domain will lead to the new domain. The new domain and the dot relation will be named after the original one but suffixed with a number. The number indicates the current available name for the same virtual domain. For instance, if the virtual domain *V* is redefined to **relation**(*A*), then a new domain *V_0* and a new dot relation, *.V_0* will be created. Later when it is actualized in relation *S1*, as in Figure 4.37, the access to domain *V* refers to domain *V_0* and new actualized data for domain *V* will be written to *.V_0*. This transformation is automatic and users can simply use the original name (*V*, in this case) whenever declaring virtual domains or doing queries in relational or domain algebra.

```
>relation S1(A) <- {(3)};
>pr S1;
+-------------+
| A           |
+-------------+
| 3           |
+-------------+
relation S1 has 1 tuple
>let V be relation(A);
>T1<-[A,V] in S1;
>pr T1;
+-------------+---------------------+
| A           | V_0                 |
+-------------+---------------------+
| 3           | 104                 |
+-------------+---------------------+
relation T1 has 1 tuple
>pr .V_0;
+---------------------+-------------+
| .id                 | A           |
+---------------------+-------------+
| 104                 | 3           |
+---------------------+-------------+
relation .V_0 has 1 tuple
>pr .V;
+---------------------+---------------------+
| .id                 | B                   |
+---------------------+---------------------+
| 101                 | x                   |
| 101                 | y                   |
| 102                 | z                   |
| 103                 | 9                   |
+---------------------+---------------------+
relation .V has 4 tuples
>
```

Figure 4.37: Redefined *V* With Different Attributes

The above examination of the attributes of a nested relation involves implementation consideration, but it is discussed here because the user must be aware of the suffixes and why they may appear. It is helpful when the nested relation is to be printed using a **pr** command as we did in Figure 4.37.

4.8 The Wildcard

In jRelix, the wildcard, ".", represents top-level relations or nested relations in a particular relation. It is useful in case the names of relations are not known or the structure of a relation is too complicated so that it is difficult to explicitly refer to the name of its nested relations.

4.8.1 The Wildcard Represents Top Level Relations

If the wildcard appears in the top-level query statement, it represents the top-level relations in the current system. Not all top-level relations will be picked up. Only those relations which satisfy two conditions, will be the relations which will replace the wildcard and take part in the query. The first condition is that the relation contains all actual domains involved in the query. The second condition is that all virtual domains involved in the query can be actualized in the relation. The operations performed on the wildcard are equivalent to operations performed on the relations the wildcard may represent. If the wildcard may represent more than one relation, **ujoin** operations are performed to combine the results of operations on each of the relation. For example, assume there are a total of three top-level relations in the system, *X1*, *X2*, *X3*, as shown in Figure 4.38, and we want to find out all relations that contain domain *A* and its value. With the query in Figure 4.39, relation *X1* and *X3* are found. The wildcard in this example represents both relation *X1* and *X3*, for domain *A* is an actual domain and relation *X2* does not contain it. Therefore query

RA <- **[self,** *A***] in .;**

is equivalent to

RA <- **[self,***A***] in** *X1* **ujoin [self,***A***] in** *X3***;**

In the example in Figure 4.40, domain *D* is a virtual domain defined on domain *C*. It can only be actualized in relation *X2* and *X3*, so the wildcard in the query is replaced by *X2* and *X3*.

```
>domain A,B,C strg;
>relation X1(A,B) <- {("a1", "b1")};
>relation X2(B,C) <- {("b2", "c2")};
>relation X3(A,C) <- {("a3", "c3")};
>pr X1;
+----------------------+----------------------+
| A                    | B                    |
+----------------------+----------------------+
| a1                   | b1                   |
+----------------------+----------------------+
relation X1 has 1 tuple
>pr X2;
+----------------------+----------------------+
| B                    | C                    |
+----------------------+----------------------+
| b2                   | c2                   |
+----------------------+----------------------+
relation X2 has 1 tuple
>pr X3;
+----------------------+----------------------+
| A                    | C                    |
+----------------------+----------------------+
| a3                   | c3                   |
+----------------------+----------------------+
relation X3 has 1 tuple
>
```

Figure 4.38: Three Top Level Relations

```
>RA<-[self, A] in .;
>pr RA;
+----------------------+----------------------+
| self                 | A                    |
+----------------------+----------------------+
| X1                   | a1                   |
| X3                   | a3                   |
+----------------------+----------------------+
relation RA has 2 tuples
>
```

Figure 4.39: The Wildcard Represents Top Level Relations

```
>let D be C;
>RC<-[self,D] in .;
>pr RC;
+----------------------+----------------------+
| self                 | D                    |
+----------------------+----------------------+
| X2                   | c2                   |
| X3                   | c3                   |
+----------------------+----------------------+
relation RC has 2 tuples
>
```

Figure 4.40: The Wildcard Represents Top Level Relations with Virtual Domain

Branch

(region	subbranch			dept		)
	(country	manager	salary)	(name	manager	salary)
Asia	China	Qiang	40000	Marketing	July	40000
	Japan	Xiazi	40000			
North America	Canada	Roy	50000	IT	George	54000
	US	Bob	55000	Marketing	Joe	50000

Table 4.5: The Display Form of Relation *Branch*

4.8.2 The Wildcard Represents Nested Relations

Furthermore, the wildcard can also be used to represent all available nested relations one level down in a relation.

Query : Get a list of managers and their salaries in relation *Branch*.

Table 4.5 is a relation named *Branch* in display form and Figure 4.41 is the initialization of the relation and the related domain declaration.

```
>domain region, country, name, manager strg;
>domain salary intg;
>domain subbranch(country, manager, salary);
>domain dept(name, manager, salary);
>relation Branch(region, subbranch, dept) <-
{("Asia",        {("China","Qiang",40000),("Japan","Xiazi",40000)}, {("Marketing","July", 40000)}),
 ("North America",("Canada","Roy",50000),("US","Bob",55000)}, {("Marketing","Joe",50000),
                                                              ("IT","George",54000)})
};
>pr Branch;
+----------------------+----------------------+----------------------+
| region               | subbranch            | dept                 |
+----------------------+----------------------+----------------------+
| Asia                 | 1                    | 2                    |
| North America        | 3                    | 4                    |
+----------------------+----------------------+----------------------+
relation Branch has 2 tuples
>pr .subbranch;
+----------------------+----------------------+----------------------+-------------+
| .id                  | country              | manager              | salary      |
+----------------------+----------------------+----------------------+-------------+
| 1                    | China                | Qiang                | 40000       |
| 1                    | Japan                | Xiazi                | 40000       |
| 3                    | Canada               | Roy                  | 50000       |
| 3                    | US                   | Bob                  | 55000       |
+----------------------+----------------------+----------------------+-------------+
relation .subbranch has 4 tuples
>pr .dept;
+----------------------+----------------------+----------------------+-------------+
| .id                  | name                 | manager              | salary      |
+----------------------+----------------------+----------------------+-------------+
| 2                    | Marketing            | July                 | 40000       |
| 4                    | IT                   | George               | 54000       |
| 4                    | Marketing            | Joe                  | 50000       |
+----------------------+----------------------+----------------------+-------------+
relation .dept has 3 tuples
>
```

Figure 4.41: Nested Relation *Branch*

Figure 4.42 shows the query and its result. Similar to the top-level case, the query

let *ms* **be red ujoin of** [*manager*, *salary*] **in** .;

is equivalent to:

let *ms* **be** (**red ujoin of** [*manager*, *salary*] **in** *subbranch*)

ujoin (**red ujoin of** [*manager*, *salary*] **in** *dept*);

```
>let ms be red ujoin of [manager, salary] in .;
>Manager <- Branch/ms;
>pr Manager;
+---------------------+-------------+
| manager             | salary      |
+---------------------+-------------+
| Bob                 | 55000       |
| George              | 54000       |
| Joe                 | 50000       |
| July                | 40000       |
| Qiang               | 40000       |
| Roy                 | 50000       |
| Xiazi               | 40000       |
+---------------------+-------------+
relation Manager has 7 tuples
>
```

Figure 4.42: The Wildcard Represents Nested Relations

4.9 Attribute Path and Schema Discovery

One of the most important applications of the **transpose** operator is to find the paths in a nested relation [Mer03]. For example, with the general query code in [Mer03], the paths in relation *products* in Figure 4.20 can be obtained as in Figure 4.43. The query result reflects the structure of the relation, but is independent of data. Although **transpose** does not evaluate nested attributes in a single invocation, note that the recursive code for *paths* allows it to do so.

Another application is to find the schema of a relation. The process is also data independent. So we use a relation with simple data set to demonstrate the application. Figure 4.44 shows the print form of a three level nested relation *O*. Table 4.6 is its display form. The general code for finding schema is from [Mer03]. The virtual nested relation *schema* is defined as a recursive one so that it can be actualized in all levels of the top-level relation. Figure 4.45 shows the query code and the target relation *Schema*, followed

```
>let path be self/attri;
>let path1 be self/path;
>let paths be relation(path);
>let paths be ([path] in(([path] in transpose(attri))
[path:ujoin:path1] ([path1] in ([red ujoin of paths] in .))));
>Paths<- products/paths;
>pr Paths;
+--------------------------------+
| path                           |
+--------------------------------+
| products/Lscale                |
| products/device/company/name   |
| products/device/company/website|
| products/device/name           |
| products/family                |
+--------------------------------+
relation Paths has 5 tuples
>
```

Figure 4.43: Path Discovery

by the compact form of the recursive nested relation, which provides the complete set of data for a user to figure out the schema in each level. Table 4.7 shows the display form of relation *Schema*. If we want to specifically check the recursive nested relation in each level, then simply use the system command **pr** with the name of the nested relation as shown in Figure 4.46.

```
>domain A,B,C strg;
>domain Q(C);
>domain P(B,Q);
>relation O(A,P) <-{("a", {("b",{("c")})})};
>pr O;
+----------------------+----------------------+
| A                    | P                    |
+----------------------+----------------------+
| a                    | 1                    |
+----------------------+----------------------+
relation O has 1 tuple
>pr .P;
+----------------------+----------------------+----------------------+
| .id                  | B                    | Q                    |
+----------------------+----------------------+----------------------+
| 1                    | b                    | 2                    |
+----------------------+----------------------+----------------------+
relation .P has 1 tuple
>pr .Q;
+----------------------+----------------------+
| .id                  | C                    |
+----------------------+----------------------+
| 2                    | c                    |
+----------------------+----------------------+
relation .Q has 1 tuple
>
```

Figure 4.44: The Display Form of Three Level Nested Relation *O*

O

(A	P	)
	(B	Q)
		(C)
a	b	c

Table 4.6: The Display Form of Nested Relation *O*

```
>let sattri be self;
>domain schema(sattri,schema);
>let schema be transpose(sattri) ujoin ([sattri, schema] in .);
>Schema<-[schema] in O;
>pr Schema;
+----------------------+
| schema_0             |
+----------------------+
| 8                    |
+----------------------+
relation Schema has 1 tuple
+----------------------+----------------------+----------------------+
| .id                  | sattri               | schema               |
+----------------------+----------------------+----------------------+
| 6                    | C                    | dc                   |
| 7                    | B                    | dc                   |
| 7                    | Q                    | 6                    |
| 8                    | A                    | dc                   |
| 8                    | P                    | 7                    |
+----------------------+----------------------+----------------------+
relation .schema_0 (Compact Form) has 5 tuples
>
```

Figure 4.45: Schema Discovery

Schema

(sattri	schema	)
	(sattri	schema)
		(sattri)
A	dc	
P	B	dc
	Q	C

Table 4.7: The Display Form of Relation *Schema*

```
>pr .schema_0;
+----------------------+----------------------+----------------------+
| .id                  | sattri               | schema_1             |
+----------------------+----------------------+----------------------+
| 8                    | A                    | dc                   |
| 8                    | P                    | 7                    |
+----------------------+----------------------+----------------------+
relation .schema_0 has 2 tuples
>pr .schema_1;
+----------------------+----------------------+----------------------+
| .id                  | sattri               | schema_2             |
+----------------------+----------------------+----------------------+
| 7                    | B                    | dc                   |
| 7                    | Q                    | 6                    |
+----------------------+----------------------+----------------------+
relation .schema_1 has 2 tuples
>pr .schema_2;
+----------------------+----------------------+
| .id                  | sattri               |
+----------------------+----------------------+
| 6                    | C                    |
+----------------------+----------------------+
relation .schema_2 has 1 tuple
>
```

Figure 4.46: Display the Nested Relation *Schema* in Each Level

Chapter 5

Applications

jRelix, the powerful high level database programming language based upon a relational data model and nested relations, has more capacities in dealing with applications for semistructured data by the implementation of attribute metadata. In this chapter, we present some of the classical queries solved with jRelix. Section 5.1 gives an example of querying a database with graphic characteristics. Section 5.2 presents the capacity of the operator **relation** and **eval** in data reorganization. Section 5.3 exhibits the ability of path expression in finding a specific part of the structure of a relation and determining the schema differences between two relations.

5.1 Integrated Graphical Query Ability

This application discusses the use of operator **relation** described in Section 4.5 and the virtual nested relation definition in Section 4.7.

Some applications contain data that can be represented in form of graph structures and queries posted on such data often involve transitive closure. In the early times, traditional relational query language could not solve those queries that contained transitive closure [AU79]. Many proposals to solve queries like these include the graphical query languages, G [CMW87], G+ [CMW88], and Graphlog [CM90]. G+ was the extension of G and Graphlog was evolved from G+. They were languages which were specially designed for graph queries, therefore, they provides an efficient way to pose queries on a graph including cyclic structure. They were complementaries to general purposes relational query language and queries that include both graphical data and relational data would require knowledge of both.

In Aldat, with a little effort which enriches its query language with a group and nest operator **relation**, queries for the same purpose as well as arithmetic functionality, if required, can be fulfilled by writing a small paragraph of code that consists mainly of a domain algebra and recursive view.

The following is an example based on the relation and a query in [CMW87]. The query is to "find the first and last cities visited in all round trips from Toronto, in which the first and last flights are with Air Canada and all other flights (if any) are with the same airline". We do the query with jRelix and also give the flight distance of each route which satisfies the query. The relation *Flights*, shown in Figure 5.1, which has one more attribute *Dist* added for the purpose of calculating the distance of a route, is from [Mer03]. The following code, presented in Figure 5.2, is from [Mer03] with a minor modification to compute the distance. The main purpose of the code is to get all possible trips provided and the airlines and distance of these trips. The recursive view fulfills the task. The nested relation *Visit* is used to record all intermediate stops of each trip. It is necessary to control the termination of the code under the circumstance that the route is a loop, i.e., a cycle in graph representation. It also provides a by-product of visited cities in a trip.

Figure 5.3 gives part of the results from the code of the transitive closure.

The next step is to get the trips that start and end with *Toronto* and the airline is *AC*. The **grep** [Gu05] operator extracts the tuples that satisfy the condition from the relation *FLTVFlightsTC*. Figure 5.4 shows all three round trips requested and their distance.

The cities visited by the round trip can be examined in nested relation *.Visit* shown in Figure 5.5. The values of the attribute *node* for each different *id* (780, 781, 782) in *.Visit* correspond to all the cities that have been visited in each round trip (shown in Figure 5.4). We did not show the sequence of these cities being visited in the round trip in this example. We can give the order of the cities, in fact, if we add one more attribute, say *seq*, in the attribute list of the nested relation *Visit* in the definition. What we also need to do is to modify the code of the recursive view above (Figure 5.2) to make sure

```
>domain From, Line,To strg;
>domain Dist intg;
>relation Flights(From, Line, Dist, To)
       <- { ("Tor", "AC", 4200, "Van"),  ("Tor", "AC", 880,  "Bos"),
            ("Tor", "AC", 690,  "NY" ),  ("Van", "AC", 4200, "Tor"),
            ("LA",  "AC", 4700, "Tor"),  ("Tor", "AA", 880,  "Bos"),
            ("Bos", "AA", 380,  "NY" ),  ("NY",  "AA", 5000, "LA" ),
            ("LA",  "AA", 1000, "SF" ),  ("SF",  "AA", 5200, "NY" )};
>pr Flights;
+----------------+----------------+-------------+----------------+
| From           | Line           | Dist        | To             |
+----------------+----------------+-------------+----------------+
| Bos            | AA             | 380         | NY             |
| LA             | AA             | 1000        | SF             |
| LA             | AC             | 4700        | Tor            |
| NY             | AA             | 5000        | LA             |
| SF             | AA             | 5200        | NY             |
| Tor            | AA             | 880         | Bos            |
| Tor            | AC             | 690         | NY             |
| Tor            | AC             | 880         | Bos            |
| Tor            | AC             | 4200        | Van            |
| Van            | AC             | 4200        | Tor            |
+----------------+----------------+-------------+----------------+
relation Flights has 10 tuples
>
```

Figure 5.1: Relation *Flights*

```
>let Node be To;
>let Visit be relation(Node);
>FLTVFlight <-[From, Line,Dist, To, Visit] in Flights;
>let Line' be Line;
>let Line'' be Line cat Line';
>let Line be Line'';
>let Dist' be Dist;
>let Dist'' be Dist + Dist';
>let Dist be Dist'';
>let Visit' be Visit;
>let Visit'' be Visit ujoin Visit';
>let Visit be Visit'';
>let From' be From;
>let To' be To;
>let To be To';
>relation FLTVFlightTC (From, Line, Dist, To, Visit);
>let Start be relation(From);
>FLTVFlightTC is FLTVFlight ujoin [From,Line,Dist,To,Visit] in [From,Line'',Dist'',To',Visit'']
  where ((Visit sep Visit') and  ((Start [From:sep:Node] Visit') or (From = To')))
  in (FLTVFlight[To:ijoin:From'][From',Line',Dist',To', Visit'] where From!=To in FLTVFlightTC);
>pr FLTVFlightTC;
>
```

Figure 5.2: Query Code for Finding Round Trip

```
+---------------+-------------------+------------+------------------+----------------+
| From          | Line              | Dist       | To               | Visit          |
+---------------+-------------------+------------+------------------+----------------+
| SF            | AAAA              | 10200      | LA               | 765            |
| SF            | AAAAAA            | 11200      | SF               | 766            |

| Tor           | ACAAAAAC          | 10960      | Tor              | 780            |
| Tor           | ACAAAC            | 10390      | Tor              | 781            |
| Tor           | ACAC              | 8400       | Tor              | 782            |
| Van           | AC                | 4200       | Tor              | 10             |
| Van           | ACAA              | 5080       | Bos              | 783            |

+---------------+-------------------+------------+------------------+----------------+
```

Figure 5.3: Partial Result of the Transitive Closure

```
+---------------+-------------------+------------+------------------+----------------+
| From          | Line              | Dist       | To               | Visit          |
+---------------+-------------------+------------+------------------+----------------+
| Tor           | ACAAAAAC          | 10960      | Tor              | 780            |
| Tor           | ACAAAC            | 10390      | Tor              | 781            |
| Tor           | ACAC              | 8400       | Tor              | 782            |
+---------------+-------------------+------------+------------------+----------------+
```

Figure 5.4: The Round Trip

that the recursion will terminate as if there were no attribute *id* in the relation *Visit*.

```
>pr .Visit;
+---------------------+---------------------+
| .id                 | Node                |
+---------------------+---------------------+
   ...
| 780                 | Bos                 |
| 780                 | LA                  |
| 780                 | NY                  |
| 780                 | Tor                 |
| 781                 | LA                  |
| 781                 | NY                  |
| 781                 | Tor                 |
| 782                 | Tor                 |
| 782                 | Van                 |
  ...
+---------------------+---------------------+
```

Figure 5.5: Relation *.Visit* : Cities Visited

The solution that is offered using the G query language is much more elegant than the code we showed. But we argue that the relational database query language armed with a little new concept could do more things than a specialized query language.

5.2 Data Reorganizing

This application discusses the use the operators **eval** in Section 4.3 and **relation** in Section 4.5.

Using attribute metadata operators, we can also do queries for the purpose of reorganizing data. The following is an application from [ABS00]. Without losing the features of **relation** and **eval** operator we are presenting, we only use part of the data from the database due to the restriction of space. Table 5.1 shows the relation *DB* containing only the *paper* part compared to the database in Chapter 4 in [ABS00]. We also add additional data to show how jRelix automatically eliminates the duplications after aggregation. Figure 5.6 shows the initialization of relation *DB*.

Our task is to "group papers under their year of publication". This needs a transformation from values to attributes, i.e., values of the attribute *year*, 2000, 2001, become attributes in the target relation. The value of each attribute is all papers published in that year. Two steps are adopted to fulfill the task. Code for the query is from [Mer03], with a minor modification as will be discussed.

First, a **relation** operation is used to form singleton relations defined on attributes *authors*, *title*, and *refersto* and these relations are grouped according to the value of year. Then the attribute *year* and the virtual attribute *titles*, which holds data of papers published in the year, are projected out from the relation resulting from the path expression "DB/biblio/paper" shown in Figure 5.7. The projection operation will eliminate the duplication of tuples.

```
>domain author, title strg;
>domain cite strg;
>domain year intg;
>domain authors(author);
>domain refersto(authors, cite);
>domain paper(authors,title, year, refersto);
>domain biblio(paper);
>relation DB(biblio) <- {({ ({
({("Suciu")},"Semi-Databases",2001,{({("Roux"),("Combalusier")},"RC76"),
           ({("Smith")},"Sm77"),  ({("Smith")},"Sm99"), ({("Suciu")},"Su01")  }),
({("Jones")},"Smith's DB Work",2000,{({("Smith")},"Sm77"), ({("Smith")},"Sm99")}),
({("Suciu")},"XML",2000,{({("Jones")},"Jo00")})  }) }) };
>
```

Figure 5.6: Relation *DB* Initialization

Next the **eval** operator is used to assign the specified values, *titles*, to the data of the

DB

(biblio				)
(paper				)
(authors	title	year	refersto	)
(author)			(authors	)
			(author	Cite)
Suciu	Semi-Databases	2001	Roux	RC76
			Combalusier	
			Smith	Sm77
			Smith	Sm99
			Suciu	Su01
Jones	Smith's DB Work	2000	Smith	Sm77
			Smith	Sm99
Suciu	XML	2000	Jones	Jo00

Table 5.1: The Display Form of Relation *DB*

```
>let titles be equiv ujoin of relation(authors,title,refersto) by year;
>next<-[year,titles] in DB/biblio/paper;
>pr next;
+-------------+--------------------+
| year        | titles             |
+-------------+--------------------+
| 2000        | 20                 |
| 2001        | 21                 |
+-------------+--------------------+
relation next has 2 tuples
>pr .titles;
+---------------+---------------+--------------------+---------------+
| .id           | authors       | title              | refersto      |
+---------------+---------------+--------------------+---------------+
| 20            | 9             | Smith's DB Work    | 10            |
| 20            | 13            | XML                | 14            |
| 21            | 3             | Semi-Databases     | 4             |
+---------------+---------------+--------------------+---------------+
relation .titles has 3 tuples
>
```

Figure 5.7: Relation *next*

attribute *year*. Here, the cast operation is applied to force the type of attribute *year* to be changed to the type ATTRIBUTE on the fly so that the data of *year* can be treated as attributes to be evaluated. A different line of code is used compared to the original code in [Mer03]: the attribute *year* is casted first, then the **eval** operation is performed. The set operation turns the value of attribute *year* to attributes as shown in Figure 5.8[1]. The cast operation coerces the type transformation.

```
>let eval (attr) year be titles;
>PaperbyYear<-[(attr) ([year] in next)] in next;
>pr PaperbyYear;
+--------------------+--------------------+
| 2000               | 2001               |
+--------------------+--------------------+
| dc                 | 23                 |
| 22                 | dc                 |
+--------------------+--------------------+
relation PaperbyYear has 2 tuples
>pr .2000;
+---------------+---------------+--------------------+---------------+
| .id           | authors       | title              | refersto      |
+---------------+---------------+--------------------+---------------+
| 22            | 9             | Smith's DB Work    | 10            |
| 22            | 13            | XML                | 14            |
+---------------+---------------+--------------------+---------------+
relation .2000 has 2 tuples
>pr .2001;
+---------------+---------------+--------------------+---------------+
| .id           | authors       | title              | refersto      |
+---------------+---------------+--------------------+---------------+
| 23            | 3             | Semi-Databases     | 4             |
+---------------+---------------+--------------------+---------------+
relation .2001 has 1 tuple
>
```

Figure 5.8: Relation *PaperbyYear*

5.3 Partial Structure and Structual Differences Discovery

This application discusses the use of operators **quote** in Section 4.2, **self** in Section 4.4, **transpose** in Section 4.6, and the wildcard in Section 4.8.

Many proposals have been made for general path expressions in order to provide better tools for querying data documents stored in an object base [CACS94], or to obtain

[1]It would be nice to get rid of the "dc" values in relation *PaperbyYear* and turn the relation into a singleton. But so far we do not have facilities to transform a sparse relation into a singleton. Please refer to Section 7.2.2 for detailed discussion.

O

(A	B	P1			P2	)
		(B	C	Q)	(C	Q)
				(C)		(C)
a	c	b1	c11	c12	c21	c22

Table 5.2: The Display Form of Relation *O*

O'

(C	P1		)
	B	(C	Q)
			(C)
c1	b3	c2	c3

Table 5.3: The Display Form of Relation *O'*

structure information embedded in the data of object-oriented systems (e.g. XSQL) [KKS92]; or to query semistructured data whose schema is irregular or incomplete (e.g. LOREL) [QRS⁺95]. Operators **self** and **transpose** are used to fulfill some queries related to relation structure discovery. The result of the path expression depends on the structure rather than the data of a relation. So relations O and O' are used as shown in Table 5.2 and Table 5.3 respectively to do the queries. Figure 5.9 and Figure 5.10 are print forms of these two relations.

In Figure 4.43, the query for finding all paths in a relation was showed. Here, another query to find special paths leading to a defined attribute are presented.

Query: Find all C in O and the paths they are on.

The code for the query comes from [Mer03]. Figure 5.11 shows the query and the result. The **where** clause is additional compared with the code in Figure 4.43 in order to limit the resulting paths to those containing attribute C. The **quote** operation forbids the evaluation of C and, therefore, makes the comparison of two operands of type ATTRIBUTE possible.

Similarly, by adding a **where** clause to the code for finding an entire schema of a relation (please refer back to Figure 4.45 for detail), we can also program code for querying parts of the schema with specified input as shown in Figure 5.12. This provides the result in Table 5.4.

Another application is to find the schema difference between two relations. Here is

```
>domain A,B,C strg;
>domain Q(C);
>domain P1(B,C,Q);
>domain P2(C,Q);
>domain attri attr;
>relation O(A,C,P1,P2) <- {("a","c", {("b1","c11",{("c12")})},{("c21",{("c22")})})};
>pr O;
+---------------+---------------+---------------+---------------+
| A             | C             | P1            | P2            |
+---------------+---------------+---------------+---------------+
| a             | c             | 1             | 3             |
+---------------+---------------+---------------+---------------+
relation O has 1 tuple
>pr .P1;
+---------------+---------------+---------------+---------------+
| .id           | B             | C             | Q             |
+---------------+---------------+---------------+---------------+
| 1             | b1            | c11           | 2             |
+---------------+---------------+---------------+---------------+
relation .P1 has 1 tuple
>pr P2;
+---------------+---------------+---------------+
| .id           | C             | Q             |
+---------------+---------------+---------------+
| 3             | c21           | 4             |
+---------------+---------------+---------------+
relation .P2 has 1 tuple
>pr .Q;
+---------------+---------------+
| .id           | C             |
+---------------+---------------+
| 2             | c12           |
| 4             | c22           |
+---------------+---------------+
relation .Q has 2 tuples
>
```

Figure 5.9: Relation *O*

```
>relation O'(C,P1) <- {("c1", {("b3","c2",{("c3")})})};
>pr O';
+---------------+---------------+
| C             | P1            |
+---------------+---------------+
| c1            | 72            |
+---------------+---------------+
relation O' has 1 tuple
>pr P1;
+---------------+---------------+---------------+---------------+
| .id           | B             | C             | Q             |
+---------------+---------------+---------------+---------------+
| 1             | b1            | c11           | 2             |
| 72            | b3            | c2            | 73            |
+---------------+---------------+---------------+---------------+
relation .P1 has 2 tuples
>pr .Q;
+---------------+---------------+
| .id           | C             |
+---------------+---------------+
| 2             | c12           |
| 4             | c22           |
| 73            | c3            |
+---------------+---------------+
relation .Q has 3 tuples
>
```

Figure 5.10: Relation *O'*

```
>domain attri attribute;
>let path be self/attri;
>let path' be self/path;
>let pathsC be relation(path);
>let pathsC be ([path] in (([path] where attri = quote C in transpose(attri))
              [path:ujoin:path']([path'] in ([red ujoin of pathsC] in .))));
>PathsC <- O/pathsC;
>pr PathsC;
+---------------+
| path          |
+---------------+
| O/C           |
| O/P1/C        |
| O/P1/Q/C      |
| O/P2/C        |
| O/P2/Q/C      |
+---------------+
relation PathsC has 5 tuples
>
```

Figure 5.11: Paths that Containing Attribute C

```
>let sattri be self;
>domain schemaC(sattri,schemaC);
>let schemaC be ([sattri] where sattri = quote C in transpose(sattri)) ujoin ([sattri,schemaC] in .);
>SchemaC<- O/schemaC;
>pr SchemaC;
+----------------------+----------------------+
| sattri               | schemaC_1            |
+----------------------+----------------------+
| C                    | dc                   |
| P1                   | 43                   |
| P2                   | 49                   |
+----------------------+----------------------+
relation SchemaC has 3 tuples
+----------------------+----------------------+----------------------+
| .id                  | sattri               | schemaC              |
+----------------------+----------------------+----------------------+
| 42                   | C                    | dc                   |
| 43                   | C                    | dc                   |
| 43                   | Q                    | 42                   |
| 48                   | C                    | dc                   |
| 49                   | C                    | dc                   |
| 49                   | Q                    | 48                   |
+----------------------+----------------------+----------------------+
relation .schemaC_1 (Compact Form) has 6 tuples
>
```

Figure 5.12: Code for Schema that Contains Attribute C

SchemaC

(sattri	SchemaC (sattri	) SchemaC) (sattri)
C	dc	
P1	C	dc
	Q	C
P2	C	dc
	Q	C

Table 5.4: The Display Form of Result of Schema that Contains Attribute C

the query:

Query: Find the structural difference between relation *O* and relation *O'*.

The solution is shown in Figure 5.13. The nested relation *paths* in the code is for finding all paths in a relation and is defined in Figure 4.43.

Firstly, we find the difference between the nested relations of the two relations. The first two lines of code do this task.

Secondly, we find the difference between the top-level scalar attributes of the two relations. The next two lines of code and "O/top djoin O'/top" in the following line do this task.

Finally, the two relations that hold the differences of the two source relations are unioned together and produce the final result.

```
>let justeP be [red ujoin of paths] in .;
>nestDif<-O/nesteP djoin O'/nesteP;
>let path be (strg) attri;
>let top be [path] in transpose(attri);
>OdifO'<- (O/top djoin O'/top) ujoin nestDif;
>pr OdifO';
+----------------------+
| path                 |
+----------------------+
| A                    |
| P2/C                 |
| P2/Q/C               |
+----------------------+
relation OdifO' has 3 tuples
>
```

Figure 5.13: The Difference of *O* from *O'*

Chapter 6

Implementation of Attribute Metadata Operators

This chapter presents the implementation strategies of the new metadata operators described in Chapter 4. Before plunging into the details of the metadata operators' implementation, a brief overview of the system architecture and development environment of the jRelix system will be given in Section 6.1. From Section 6.2 to Section 6.7, the implementation tactics of six metadata operators: **typeof**, **quote**, **eval**, **self**, **relation**, and **transpose**, will be discussed respectively. Section 6.8 will be about redefining virtual nested relations. Section 6.9 and Section 6.10 will present the implementation of the wildcard and the expansion of recursive virtual nested relations.

6.1 jRelix System Overview

This section is a background introduction for the implementation of new metadata operators in the jRelix system, including the jRelix development environment, system architecture, and associated classes.

6.1.1 Development Environment and Tools

The jRelix system is implemented in Java (which "j" in jRelix stands for), a platform independent and object-oriented programming language. The system is composed of classes which model the entities of the system, such as relation, domain, etc. These class files are first compiled, then the system can run in Windows, Unix, or Linux with the compatible version of the Java run-time environment being installed.

As we will see in Section 6.1.2, one of three portions of the jRelix system is a front-end processor, part of whose functionalities is to parse the user input command. JavaCC and JJTree are used to generate the parser. JavaCC (Java Compiler Complier), according to [SDV03], "is a tool which reads a grammar specification and converts it into a Java program that can recognize matches to the grammar". In jRelix, JavaCC reads in the specified syntax file and converts it to a class file, *Parser.java*, and later during parsing, any user input command will be checked against the specified syntax by this program. JJTree is a utility used together with JavaCC to provide a parser tree. According to [SDV03], JJTree "is a preprocessor for JavaCC [tm] utility that inserts parse tree building actions at various places in the JavaCC source". In our system, jRelix syntax is stored in the file *Parser.jjt*. Prior to system installation, the file is first input to JJTree and its output, *Parser.jj*, then is fed into JavaCC. Figure 6.1 gives a clearer depiction of their relationship.

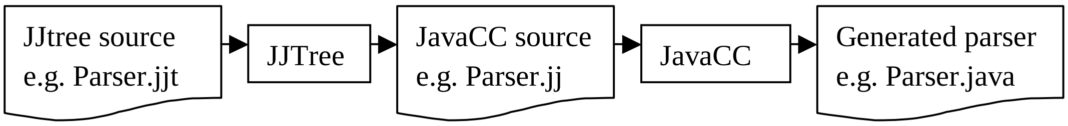

Figure 6.1: Process of Generating a Parser Using JJTree and JavaCC

6.1.2 System Architecture and Storage Format

System Architecture

The jRelix System contains three parts: the front-end processor, the execution engine, which is in charge of managing the database, and the database maintainer. The front-end processor includes the user interface, the parser, and the interpreter. User command input via user interface is intercepted first by the parser and syntax analysis is performed on the spot. A command will not be executed further if it has syntax errors. A grammar error free command is converted to a syntax tree and passed to the interpreter where the tree is traversed. The semantic checking is performed, the command is analyzed and the corresponding method in the execution engine is invoked for each function the command involved during the traverse. Methods in the execution engine implement the core function of the jRelix system, i.e., the relational algebra, the domain algebra, the computation, event handlers, and so on. The final results in this stage, such as user data, domains and relations, are stored to RAM or the disk in form of a system table by the database maintainer. Figure 6.2 depicts the structure of this system.

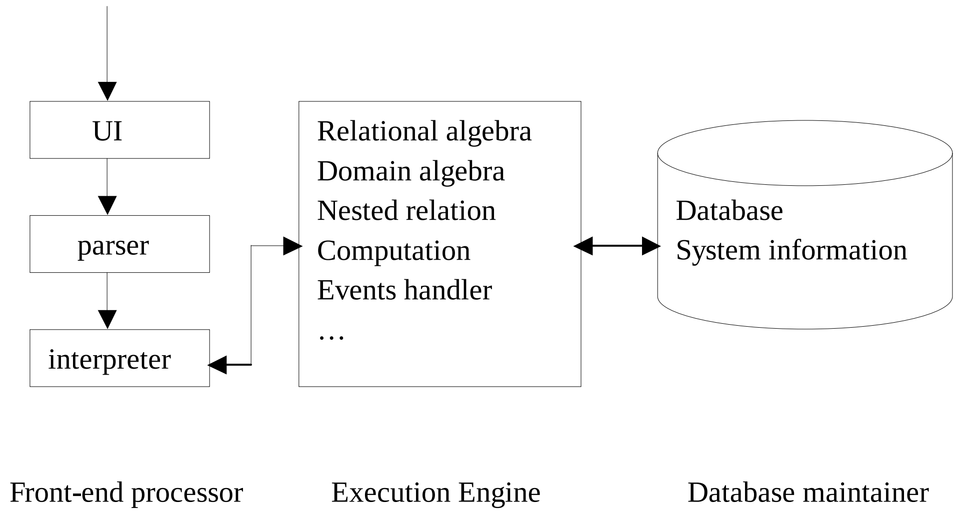

Figure 6.2: jRelix System Architecture

System Table

Two system tables are used to store information about attributes, relations, views, and computations of a database [Hao98]. They are stored in form of system relation *.dom*, *.rel*, *.rd* and two files, *.expr* and *.comp* on hard disk. These files are loaded and constructed into two system tables, *relTable* and *domTable*, once the jRelix system starts and written back to hard disk when the system exits. The relationship among the files on disk and the tables in RAM are presented in Table 6.1. The RAM version of domain table and relation table are maintained by *DomTable.java* and *RelTable.java* in the form of objects containing fields as shown in Table 6.2 and Table 6.3.

Please note that jRelix is a RAM based system; any individual relation must fit into primary memory.

disk files	description	RAM version
.rel	Information about relations	relTable
.dom	Information about attributes	domTable
.rd	Information links relations with the attributes on which relations are defined	relTable and domTable
.expr	Serialized syntax tree of views and virtual attributes	Virtual domains to domTable Views to relTable
.comp	Serialized syntax tree of computations	

Table 6.1: Relationship Between System Files on Disk and Their RAM Version

field	type	description
name	string	The name of the domain
type	integer	The type of the domain
tree	simpleNode	A syntax tree which is the definition for a virtual domain
numref	integer	The number of this domain being referenced
union	SimpleNode	The compositions of types of type UNION

Table 6.2: RAM Version of Domain Table: *domTable*

field	type	description
name	string	The name of the relation
rvc	integer	Type of the relation (relation, view, or computation)
numtuples	integer	The number of tuples in the relation
numattrs	integer	The number of attributes in the relation
tree	SimpleNode	Syntax tree for a view
domain	Domain[]	Array of domains
data	Object[]	Array of data

Table 6.3: RAM Version of Relation Table: *relTable*

category	name	description
member	name	Name of the node if the node is an IDENTIFIER
	type	Operator type
	opcode	Sub operator type
method	jjtCreatNode	Create and return a new SimpleNode
	jjtGetChild	Return a child node
	jjtGetParent	Return the parent node
	jjtReplace	Replace a node with the one input
	jjtGetNumChildren	Return the number of the input node's children
	jjtAddChild	Add the input node as a child to the parent node

Table 6.4: Frequently Used Members and Methods of the *SimpleNode* Class

Storage Format of Flat Relations and Nested Relations

In jRelix, flat relations are stored in a table format, first tuple-by-tuple and then attribute-by-attribute. Their storage forms are the same as their display forms and print forms, as we have seen in Chapter 3. Nested relations are stored as several flat relations, including one top level flat relation and several other flat relations which retain the data of nested relations. Each of them can be displayed using a **pr** command.

6.1.3 Introduction of Related System Class

The Syntax Tree and the SimpleNode Class

A node is the basic unit of a syntax tree generated by the parser from the user input command. For example, the input

$$S \texttt{<-} [A] \ \textbf{in} \ R;$$

is parsed to a tree as shown in Figure 6.3. The tree is rooted at the node *assignment*, the parent node of nodes S and *project*. Node S is the first or left child of node *assignment*, and node *project* is the second or right child of *assignment*. A node is encapsulated in the *SimpleNode* class in the jRelix system. The most commonly used members in the class are *name*, *type*, and *opcode*, while the most frequently referenced methods are *jjtCreateNode*, *jjtGetChild*, *jjeGetParent*, *jjtReplace*, *jjtGetNumChildren*, and *jjtAddChild*. A brief description of these members and methods are available in Table 6.4.

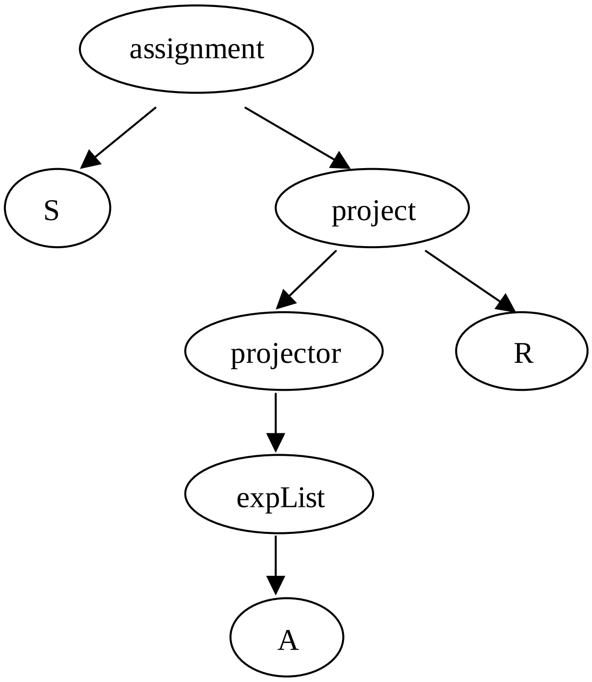

Figure 6.3: A Syntax Tree

The Interpreter Class

The interpreter class receives the syntax tree passed from the parser and explains the tree by traversing it starting from the root. Function calls are made during the traverse according to the type of the current processing node. When traversal of a syntax tree has completed, the functions of the query the tree portrayed are executed and the modifications to the database have been applied.

Some of the most important methods of the interpreter class are presented in Table 6.5. New methods for supporting the operator **relation** and **transpose** will be added to the class in our implementation[1].

The Actualizer Class

The actualizer class is responsible for the virtual domain actualization. It performs a run-time validity check for a virtual domain and also expands a virtual domain tree to an entire tree which the actualizing process can process. These functions are performed

[1]methods for operator **relation** and **transpose** are marked with a '*' in the table.

methods	comments
executeCommand	Analyze a command and call corresponding method
traverseNode	Basic validation check of a tree
traverseType	Type checking in virtual nested relation declaration
getIDList	Get the domain list involved in virtual nested relation declaration
evaluateTLExpression	Analyze an expression and call corresponding method
evaluateProject	Deal with a syntax tree for projection operation
evaluateSelect	Deal with a syntax tree for selection operation
evaluateJoin	Deal with a syntax tree for join operation
evaluateRelation*	Deal with a syntax tree for relation operation
evaluateTranspose*	Deal with a syntax tree for new version of transpose operation

Table 6.5: Some Important methods in the *Interpreter* Class

category	methods	description
	buildtree()	performing run time validation check
	actualizing()	calculate virtual domain value
cell methods	actIntCell()	for INTEGER or SHORT type domain
	actBoolCell()	for BOOLEAN type domain
	actLongCell()	for LONG INTEGER type domain
	actDoubleCell()	for FLOAT or DOUBLE type domain
	actStrCell()	for STRING type domain
	actRelCell()	for IDLIST(relation) type domain

Table 6.6: Important Methods in the *Actualizer* Class

by the method *buildtree()* in the class. Another important method is *actualizing()*. It is the method that calculates the values of virtual domains by invoking a different cell method according to the type of the virtual domain. A virtual domain is actualized in a tuple-by-tuple approach, where the value is calculated from the first tuple to the last tuple based on the value of the actual domains of the corresponding tuple [Yua98]. Different cell-methods are used for different types of data and Table 6.6 is a summary of important methods in the *actualizer* class. In our implementation, *actStrCell()* will be augmented for supporting operator **typeof**, **eval**, and **self** . Also *actBoolCell()* will be enhanced for accomplishing comparison of the two ATTRIBUTE type operands which indirectly support the **quote** operator when it acts as one of the operands of the comparison operation.

Type code	Description
INTEGER = 2	Code for integer type
STRING = 7	Code for string type
ATTRIBUTE = 18	Code for attribute type
TYPE = 27*	Code for type type
....	

Table 6.7: Type Code

Operation code	Description
OP_DECLARATION = 140	Code for declaration operation
OP_DOMAIN = 143	Code for domain operation
OP_RELATION = 141**	Code for relation operation
OP_EVAL = 404**	Code for eval operation
OP_QUOTE = 406**	Code for quote operation
OP_TRANSPOSE = 408	Code for transpose operation using first version of syntax
OP_TRANSPOSENEW = 409*	Code for transpose operation using new version of syntax
OP_TYPEOF = 415*	Code for typeof operation
OP_SELF = 416*	Code for self keyword
......	

Table 6.8: Operation Code

The TypeConstants Class and the Constants Interface

In the jRelix system, all type constants are specified in *TypeConstants.java*, and all operation type codes and sub-type codes are specified in *Constants.java*. Adding a new type or operator to the system will need to augment the corresponding files. In Table 6.7 and Table 6.8, examples of code for types and operators are displayed as well as the code for a new data type and operators added in this implantation[2].

[2]*: new added type the code or operator code; ** original operator code, new features augmented.

6.2 Implementation of Type TYPE and the *typeof* Operator

6.2.1 Type TYPE

A new data type, TYPE, is added to the jRelix system to store types in the system. The inner system representation for the data is string type. The TYPE data type is defined in file *TypeConstants.java* and can be referenced as a constant. (The code for the constant is 27). The OP_TYPE, which was defined in file *Constants.java* and is used as the operation type for all types, was borrowed to represent the operation type of TYPE. The input value of TYPE is an IDENTIFIER and need not be quoted by the quotation marks. It is assigned to the relation data array in the relation initialization process. This function is performed in method *assignIdentifier()* in *interpreter.java*. The method is also responsible for checking the validity of the data as a value of TYPE. Specifically, the input IDENTIFIER must be one of the system data types, the name of a declared actual domain or a defined virtual domain, or the name of a declared relation.

6.2.2 The *typeof* Operator

As the value of type is stored in STRING format in the system, the actualization of the **typeof** operator is performed in *actStrCell()* method. This is particularly true if the operand is of the type ATTRIBUTE, UNIVERSAL, or UNION, for the values of attributes of these types are usually different for each tuple in a relation. However, for attributes of other types, such as the type STRING or INTEGER, their type is a constant in a relation and the type info can be determined at the system level. Different steps, therefore, are taken to deal with the **typeof** operator in our implementation. The method *processTypeof()* deals with the different situations.

The method *processTypeof()* is invoked by a *buildTree()* method with the current node when the node is a **typeof** operator. Thus the input of the method is the node **typeof** with the operand tree as its child. In this method, firstly, the type of the operand is determined by recursively invoking the method *buildTree()* with the operand node. If the type is ATTRIBUTE, UNIVERSAL or UNION, then the program simply returns to the caller. The **typeof** tree is untouched and will be managed in the tuple-by-tuple process. Otherwise, if the type is none of the three types, a new node of type LITERAL which contains the type value will be created to replace the current node, the node of operator **typeof**. Figure 6.4 depicts the transformation.

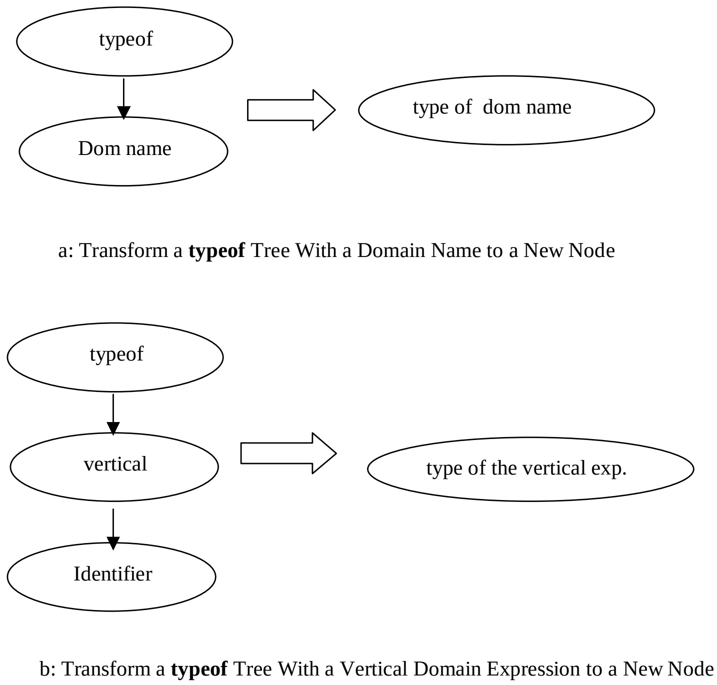

Figure 6.4: **typeof** Transformation

In the above discussion, the type result of the **typeof** of an attribute of type ATTRIBUTE, UNIVERSAL or UNION, remains undecided. It will be figured out in the process of tuple-by-tuple actualization. The method *actStrCellforTypeof()* does the actualization. This method is called by *actStrCell()* when the current node is a **typeof** operator. The input of the method is the node of **typeof** with the operand tree as its child.

Two cases are conducted in this method. The first is that the operand of **typeof** is an **eval** operator followed by its operand. In this case the data of the **eval** operand is first actualized by the recursively calling *actStrCell()* method. A domain then is looked up from the *domTable* using the obtained data as its name. Finally, the type of the domain is obtained from the type field of the domain.

The second case is that the operand of **typeof** is an IDENTIFIER. In this case, different steps are used to get the result of the operator according to the type of the domain the IDENTIFIER stands for. Only three types, ATTRIBUTE, UNIVERSAL, and UNION are possible here because other types already have been transformed to constant nodes in the method *processTypeof()*. If the domain type is ATTRIBUTE, then the value of the domain is fetched. A new domain will be looked up from the *domTable* using the value as the domain name and its type is grabbed from the type field and returned. If the domain is of a type UNIVERSAL or UNION, then when the value of the domain in the current tuple is fetched, the type part of the value will be extracted from the value. Because the value may be in form of "universal:string:name" for type UNIVERSAL as we have seen in Figure 4.33, we do extra steps to extract the exact type. A **while** statement is used to trace the type part in the value until it is not a type:value pair. The process of getting the type of type UNION is the same as that of the type UNIVERSAL. The pseudo code for the method is shown in Figure 6.5.

```
private String actStrCellForTypeof(SimpleNode node) {
   Get the operand node
   If the operand node is an eval operator
      Get the node of eval operand
      Get the value of eval operand by calling actStrCell with eval operand
      Look up the domain of the value from the domain table
      Get the type of the value domain
      return the type
   Else
      Get the domain of the operand node
      Get the value of the identifier by calling actStrCell with typeof operand
      switch(domain's type)
         case ATTRIBUTE:
            Look up the domain of the value from domain table
            Get the type of the value domain
            return the type
         case UNION:
         case UNIVERSAL:
            Extract type from the type:value pair
            While the value is a type:value pair
               Extract the value from the type:value pair
               Extract type from the type:value pair
               return the type
         default: print "wrong type"
}
```

Figure 6.5: Pseudo-code for *actStrCellForTypeof()*

6.3 Implementation of *quote* Operator

The **quote** operator is used to transform an attribute name into an attribute metadata [Mer01]. The **quote** operator in domain algebra has been implemented in [Roz02]. Here we present the implementation of the operator when it participates in relational algebra.

As **quote** is used to suppress the evaluation of the attribute following it, the implementation is to transform the **quote** tree into a literal node (type= OP_LITERAL, opcode = OP_ATTRIBUTE). The transformation is performed in the process of *buildtree()* in actualizer construction. For example, in the query

`TpName <- ` **where** `attri = ` **quote** `forBusiness` **in** `TP1;`

in figure 4.8, the **quote** *forBusiness* part in the syntax tree of the whole query is transformed to *forBusiness*. Figure 6.6 depicts the idea.

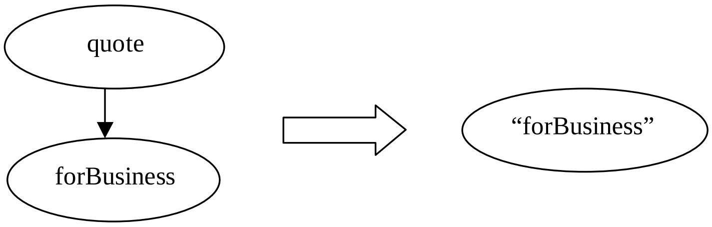

Figure 6.6: **quote** Transformation

In addition to transforming the **quote** tree, one more thing should be done to make the above query work. That is to augment the method *actBoolCell()* so that it can support comparison of two operands of type ATTRIBUTE. Because the inner storage of value of type ATTRIBUTE is string, the comparison is the same as that of two STRING type operands. Therefore we simply add a line of code, "**case** `ATTRIBUTE:`", to code where "**case** `STRING:`" is processed.

6.4 Implementation of *eval* Operator

6.4.1 Implementation of actualizing *eval* in Cell Method

The **eval** operator evaluates the value of each attribute in the data of **eval** operand in the corresponding tuple. Apart from the evaluation of the attribute as it would be evaluated in a normal situation, one more evaluation is specified by the **eval** operator for the attribute.

As the type of the result of **eval** operator is UNIVERSAL, the actualization happens in *actStrCell()*. In the tuple-by-tuple procedure, when the current node is of type

OP_EVAL, method *actStrCellForEval()* is invoked by *actStrCell()* method. The node of **eval** operator is passed to the method at the same time. Method *actStrCellForEval()* evaluates the attribute resulting from the **eval** operand in the current tuple, and returns the value. Six steps are taken to accomplish the evaluation.

First, the operand of **eval**, which is the child node of eval node, is obtained. Two cases could happen at this stage. The operand of **eval** could be an IDENTIFIER which is the name of an attribute of type TYPE. It also could be a **cast** operator with two children. One should be the node containing the cast type, ATTRIBUTE, the other is an attribute of any type. In any case, the proper node which is the name of an attribute is obtained.

Second, the value of the operand is actualized by recursively invoking the method *actStrCell()* with the obtained node.

Third, the domain of the value obtained in the second step is looked up from the domain table.

Fourth, a singleton relation which contains all domains of source relation and data of the current tuple of source relation, is created.

Fifth, the domain obtained in the third step is actualized in the relation obtained in the fourth step.

Finally, the type of the domain procured in the fifth step is attained from the type field and the type value is returned.

6.4.2 Implementation of *eval* in Left Hand Side of Declaration

In [Roz02] the static evaluation of **eval** is fulfilled in the process of declaration. The declaration

let eval `UR` **be** `"ANY";`

would assign the constant "*ANY*" to all domains which are data of unary relation *UR*. This is applicable when the value to be assigned is a constant. We have enhanced the implementation as following: if the assigned value varies from tuple to tuple, a new domain, named after the operand of **eval** and suffixed with "*_evaled*", is generated and put to the *domTable* to record the **eval** of the operand. Take the query in Figure 4.14 as an example. The result of "*X+Y+Z*" is different in each tuple in the relation. So, for the statement:

let eval `B` **be** `X + Y + Z`**;**

we create a new domain named *B_evaled*, a domain marked as the evaled [3] domain of *B*, and assign the definition tree *X*+*Y*+*Z* to the tree field of the domain. It is interpreted as "the value of any attribute which is the value of domain *B* is to be defined as *X*+*Y*+*Z* in the current tuple". This is accomplished in the process of executing domain declaration. The type of the evaled attribute is determined by its definition *X*+*Y*+*Z*.

In the previous implementation, an attribute was examined at two levels in the process of actualization. The first one is to test if it is an actual domain of the source relation. This is performed by a method *isDomainIn()* in *relation* class. If the result is *true*, then the attribute is one of the domains of the relation. If the result is *false*, then the second level examination is performed. This test is aimed to find out if the domain is a virtual domain by looking up the tree field of the domain. If the tree field is not empty, then it is a virtual domain. To ensure that this virtual domain is a valid one, its definition tree is recursively visited and all domains it is defined on are scrutinized against the two level exams. If in each iteration all domains are valid, the domain is actualized in the later process. Otherwise, the domain was invalid and the operation involved fail.

Now, one more level of examination is added. That is, if the domain is a data of another attribute, then the evaled attribute of that attribute is scrutinized. The domain with the evaled name, the name of the attribute suffixed with *_evaled*, is looked up in the domain table. If the evaled attribute exists, then the domain to be actualized is redefined on the definition stored in the evaled attribute, the tree in the evaled attribute tree field. If the evaled attribute does not exist, then it comes to the conclusion that the domain is invalid, an error message will be given, and all operations terminate. In our example, domain *X'*, *Y'*, and *Z'* neither are all actual domains in relation *R1*, nor virtual domains. But they are data of attribute *B* projected in relation *R1*. So the evaled domain of *B*, *B_evaled* is looked up from the *domTable*. Because it has been declared previously, domain *X'*, *Y'*, and *Z'* then follow the definition of *B_evaled*, which is *X*+*Y*+*Z*.

[3] "evaled" is used in this chapter to represent "**eval**uat**ed**"

Different from a normal declaration, in which we could declare domain *X'*, *Y'*, and *Z'* to be

let `X'` **be** `X+Y+Z;`

let `Y'` **be** `X+Y+Z;`

and

let `Z'` **be** `X+Y+Z;`

as the **eval** operator only has effect on the attribute which is the data of the **eval** operand in the current tuple, the value of this particular attribute in other tuples should have the value "dc" [Yan03]. We need, therefore to modify the declaration of attributes to:

let *X'* **be if** *B* = *X'* **then** *X* +*Y* +*Z* **else** *dc*;

let *Y'* **be if** *B* = *Y'* **then** *X* +*Y* +*Z* **else** *dc*;

and

let *Z'* **be if** *B* = *Z'* **then** *X* +*Y* +*Z* **else** *dc*;

The action of transforming a node to be evaled to a virtual domain defined on "*IFTHENELSE*" expression is accomplished in method *Eval2IFTHENELSE()*.

6.4.3 Modifications Related to Set Operation

We have two modifications associated with a Set Operation (Please refer to Section 3.4.2 for details of set operation). One is in the method *AttribsRel2DomList()* and the other is in *ExpressionListToDomains()*.

Originally, the method *AttribsRel2DomList()* was implemented to transform the data of a unary relation defined on an attribute of the type ATTRIBUTE to a set of attributes, and to turn all domains of a non-unary relation to a set of attributes. In the first case, as long as an attribute which was the value of the attribute on which the unary relation defined has been declared, i.e., it could be looked up from the *domTable*, it would be one of the target attributes. If any of them was an actual attribute but not in the source relation, then an error message would display in the stage of constructing an actualizer, and the operations involved would fail. For instance, the original method led to an error if we did the query shown in Figure 4.14. Figure 6.7 shows this case.

```
>let eval B be X+Y+Z;
>S1<-[[B]in R1] in R1;
domain 'X'' is not in the domain list of relation 'R1'.
>
```

Figure 6.7: An Unsuccessful Set Operation

Now, we give a general specification of the Set Operation in jRelix. That is, the Set Operation intends to obtain a set of attributes from a relation resulting from a relational expression. To accommodate this definition, modifications are made to method *AttribsRel2DomList()*.

First, we keep the portion that deals with non unary relations.

Second, the code that deals with unary relations defined on attributes of type ATTRIBUTE is taken out and replaced with code capable of the following functionalities:

For each value of the attribute that the unary relation defined on:

1. If the domain with the value as its name is an actual domain in the source relation, then it is taken to be one of the target attributes. The source relation is obtained from the relation name table *NRelName* (to be discussed in Section 6.5.1 for detail). Otherwise,

2. If the domain with the value as its name is a virtual domain, then it is taken to be one of the target attributes. Otherwise,

3. Check if the attribute the unary relation defined on is evaled by looking it up in the *domTable* with name of the attribute suffixed with "*_evaled*". If the evaled domain exists, then method *Eval2IFTHENELSE()* is invoked and the attribute being checked is declared after the definition of the evaled domain.

If none of the above conditions are satisfied, an error message will be thrown out and the program terminated.

Third, when the unary relation is defined on an attribute of type other than ATTRIBUTE and the attribute is evaled, then new virtual domain for each of the values of the attribute is generated following the definition of the evaled domain. The domain is created by invoking the method *Eval2IFTHENELSE()*, the name of the domain is obtained from the data array of the unary relation, and type casting is performed if necessary (e.g., if the value of the attribute is an integer, say, 2000, it will be cast to a string "2000").

In the method *ExpressionListToDomains()*, **cast** operation is now allowed to cast the domain set resulting from method *AttribsRel2DomList()* to type ATTRIBUTE when the expression is projection, selection, etc.

6.4.4 Implementation of *eval* in Right Hand Side of Declaration

In the statement

let *P* **be eval** *A*;

the data of attribute A may not be a constant, so the type of the domains which have the name of the data may be different, hence, the type of P is specified as UNIVERSAL. This is dealt with in the process of declaration. In the *traverseType()* method, the case OP_EVAL is added and the type UNIVERSAL is returned. Later during actualization, P will be actualized the same way as a normal virtual domain and the **eval** operator in its definition will be processed in the way discussed in Section 6.4.1.

6.5 Implementation of *self* Operator

As shown in Chapter 4, the **self** operator can appear in both normal expressions and path expressions where the name of a relation may be ".". In following sub-sections we discuss the implementation of these two cases separately.

6.5.1 Implementation of *self* in Normal Expression

In a query in the form of a normal expression, the **self** is treated as a special virtual domain which has no definition tree. During run time, the **self** domain is created and saved to the *domTable* the first time the **self** node is traversed. The type of **self** domain is ATTRIBUTE, and the name of the domain is *self*. The actualized value of the **self** domain is the name of the current level relation.

Two variables are used in the actualizing of **self**. The first variable is a table named *NRelName*. It is a vector of the type STRING, used to store the name of each relation which is being traversed. Each time the program traverses to a relation, the name of the relation is added to the table, and the name is deleted from the table when the traverse of the current relation is finished. The second variable is an index, *relLevel*, which traces the level of relations the current actualizing process is in. Each time the program traverses to a relation, the index value is increased by one and it is decreased by one when the traverse of the current relation is finished. The value of the index indicates the position where the name of the current relation is in the relation table, *NRelName*, hence, when there is a **self** keyword needed to be actualized, the name of the relation can be retrieved from the relation name table according to the index. The processes of adding, deleting, and retrieving the name of a relation are performed by the methods *addRelName()*, *delRelName()*, and *getRelName()*, respectively.

The actualizing of the **self** domain is implemented in the method *actStrCellForSelf()*, invoked by method *actStrCell()*. It obtains the relation name by calling the method *getRelName()*. It also sends warning information if **self** is to be actualized in an unnamed top level relation. Take the query in Figure 4.22 as an example. Initially, the relation name table is empty, and the index *relLevel* equals 0. When the command is started to be executed, the first relation node which is traversed is *products*. So the first element being added to the table is *products* and *relLevel* is increased by 1. In processing the projector of the projection statement, the virtual domain *srvir* is defined to be a projection operation, then the second relation node encountered is *nm*. So that the second element in the table is *nm* and the index now is equal to 2. The name *device* is added to the vector and the index is increased to 3 when the program deals with the node *device*, the third relation node in the query. After the actualization of the projection "[*name*] **in** *device*;" the name *device* is deleted from the vector, and the index is decreased to 2. Then the program processes the projection "[*self*] **in** *mn*;", in actualizing the domain *self*, the relation name pointed by the index *relLevel*, which is *nm*, is obtained from the table, and the value of domain *self* is *nm*. The element *nm* is deleted from the vector and the index is decreased to 1 after the evaluation of the projection. Finally, when the projection "[*red ujoin of srvir*] **in** *products*;" is fulfilled, the element *products* is deleted from the table and the index is decreased to 0.

6.5.2 Implementation of *self* in Path Expression

The user input of **self** in path expression, such as **self**/attri, will be parsed into a syntax tree which represents

`(STRING)` **self cat** `"/"` **cat** `(STRING) attri`

as in Figure 6.8. New code is added in *parser.jjt* for this purpose. The **self** then is actualized in run time in the same way it appears in the normal expression, as discussed in Section 6.5.1.

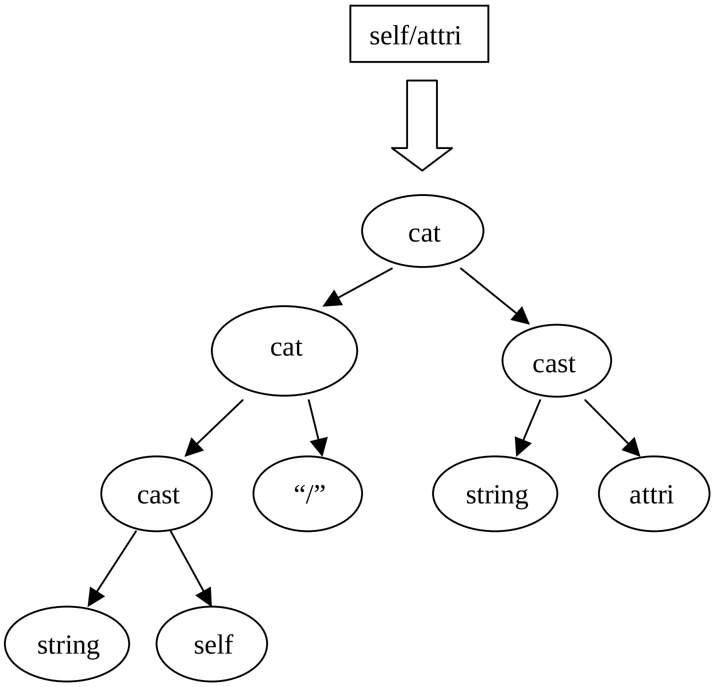

Figure 6.8: Parsing of **self**/attri

6.6 Implementation of *relation* Operator

The **relation** operator is a nest operator which groups one or more attributes into a nested relation. Each instance of the nested relation corresponds to the value of the attribute(s) of one tuple in the source relation. The result of the operator is a singleton relation.

In the procedure of tuple-by-tuple actualization, if the current node is a **relation** operator, the value of the cell is obtained by evaluating the relation expression in the source relation. This is performed by the method *evaluateRelation()*. Figure 6.9 is the pseudo code for *evaluateRelation()* method. The returned singleton relation is then

```
Relation evaluateRelation(SimpleNode node, Hashtable htable, Environment env)
{
    Get the source relation ``srcR'' in which the relation operator is actualized;
    Get the domain list (``domList'') on which the relation operator is defined;
    Actualize the ``domList'' in the ``srcR'', the new relation is ``tempR'';
    Decide the position of each domain of ``domList'' in tempR;
    Create a relation ``R'' which defined on the ``domList'', number of tuple is one;
    Construct a data array ``Data'' for the new created relation ``R'';

    For each domain in ``domList'',
        Get the data in ``tempR'' and assign it to the data array ``Data'',
        according to different type of domain;
        Assign the data array ``Data'' to the new relation ``R'';
    return ``R'';
}
```

Figure 6.9: Pseudo-code for *evaluateRelation()*

back to normal actualization and put into the dot relation, which has a system generated name and is created the first time this relation operator is evaluated. A surrogate will be generated which links the real data to the destination relation where the relation expression is actualized.

In a **join** operation, if only one attribute is to be nested, the keyword **relation** can be omitted. An example was shown in Figure 4.25. To allow this short form, we do modifications in *traverseType()*. When the type of the left join operand and the right operand is obtained, one more step is taken to check if it is a non IDLIST Identifier. If it is a scalar domain and it has been declared, then the method *ScalarDom2Relation()* is invoked and the node of the domain is transformed into a syntax tree rooted at the node of a **relation** operator. Figure 6.10 depicts the idea by transforming the syntax tree of "*family*" to that of "**relation** (*family*)" in query in Figure 4.25.

This short form is also applicable for reduction, equivalent, function mapping, and partial function mapping operation. Particular examination is accomplished before doing the transformation from an Identifier to relation operation, i.e., only if the sub-operation type is of **join** operation will be the node of identifier to be transformed.

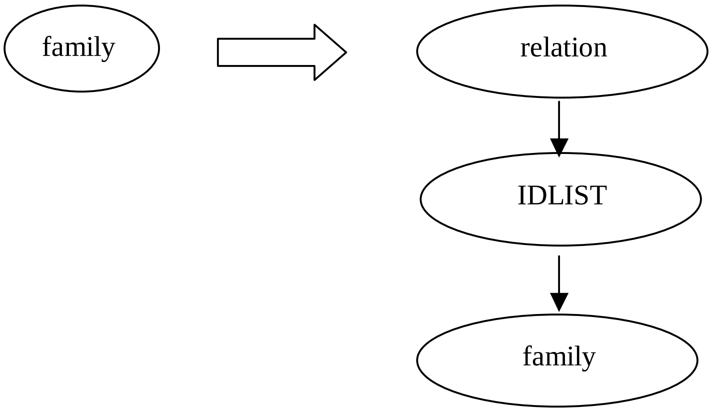

Figure 6.10: Transforming a Domain Node to Relation Operation

6.7 Implementation of *transpose* Operator

The function of the **transpose** operator is to get any of the type, attribute, and value of the scalar attributes in a relation and group them into a nested relation. Each instance of the nested relation contains the data of all scalar attributes in the corresponding tuple in the source relation. So the number of tuples of a nested relation is equal to the number of scalar attributes of the source relation.

In the process of tuple-by-tuple actualizing, if the current node is a **transpose** operator, the value of the cell is obtained by evaluating the transpose expression in the source relation. This is performed by the method *evaluateTranspose()*. Figure 6.11 shows the pseudo code for this method.

```
Relation evaluateTranspose(SimpleNode node,  Hashtable htable, Environment env)
{
    Get the source relation ‘‘srcR’’ in which the relation operator actualized;
    Get the domain list (‘‘domList’’) on which the transpose operator defined and check their validation;
    Get the scalar domain list (‘‘ScalarDomL’’) in ‘‘srcR’’;
    Decide the position of each scalar domain in ‘‘srcR’’;
    Create a relation ‘‘R’’ which defined on the ‘‘domList’’;
    Construct a data array ‘‘Data’’ for the new created relation ‘‘R’’;

    For each domain in ‘‘domList’’,
    For each domain if the domain in ‘‘ScalarDomL’’,
    Get the name/type/value of the domain and assign it to the data array,

    Assign the data array ‘‘Data’’ to the new relation ‘‘R’’;
    return ‘‘R’’;
}
```

Figure 6.11: Pseudo-code for *evaluateTranspose()*

In this method, first, the source relation on which the **transpose** operation is working is obtained.

Second, the domains which will form the domain list of the new nested relation are obtained and their numbers and types are detected. Each type of attribute of the three: ATTRIBUTE, TYPE, and UNIVERSAL can at most appear once so that the total number of domains should be less than four and there should be at least one domain defined.

Third, as we mentioned, only scalar attributes of the source relation will be transposed, so we need to get the scalar attributes of the source relation. This is performed by the method *getScalarAttributes()*, a method defined in *relation* class. It checks each attribute of the current relation, picks the scalar attributes, and stores them in a domain array. After getting all the scalar domains, it returns the domain array to the method called it.

Fourth, construct a relation with the domains in *domList*. The number of tuples of the relation equals the number of scalar domains of the source relation.

Fifth, get the data of current tuple from the source relation to form the nested relation. The value of the attribute of type ATTRUBUTE is a scalar attribute in the source relation; the value of the attribute of type TYPE is the type of the scalar attributes; the value of the attribute of type UNIVERSAL is a "type:value" pair: the type part is the type of the attribute and the value part is the value of the current tuple of the source relation.

Finally, the newly generated relation is returned.

The returned relation is then back to normal actualization and put into the dot relation, which has a system-generated name and is created the first time this transpose operator is evaluated. A surrogate will be generated and link the real data to the destination relation where the relation expression is actualized.

6.8 Redefining Nested Virtual Domain

In the previous implementation, the declaration of a virtual nested relation will create a domain as well a dot relation, where the actualized data of the domain is held. These functions are fulfilled in *AddVirtDom2DomTable()*, which will invoke the *newVirDomain*

() to create the new domain. After the domain is actualized, the dot relation and its data need to be kept as long as the domain is being referenced. This brings up the consideration in implementation that when a virtual nested relation is redefined, the domain and its related dot relation cannot be put into the system table again. Otherwise, the actualized data in the dot relation will be wiped out. Modifications in the two methods mentioned above are made to ensure that if the domain being declared is of the type IDLIST and has already existed, the same domain and the dot relation are not put into the system table. This is shown in Figure 6.12 and Figure 6.13. Original code is commented out and replaced with modified code.

```
AddVirtDom2DomTable(SimpleNode node,  Environment env) throws InterpretError
  {
    .....
    Domain newDom = newVirDomain(domname, tempnode, env);
    Relation rel_saved = env.lookupRel("."+domname,true);

    /*
    env.put(newDom);
    if (rel_saved != null)    env.put(rel_saved);
    */  // original code being commented out

    if (newDom.type != IDLIST ||
        (newDom.type == IDLIST && env.lookupDom(newDom.name,true) == null)) { // new code
             env.put(newDom);
             if (rel_saved != null)    env.put(rel_saved);
   }
}
```

Figure 6.12: Modifications in *AddVirtDom2DomTable()*

```
Domain newVirDomain(String name, SimpleNode node, Environment env) throws InterpretError
{
    ......
    if new domain is type of IDLIST
    {
    ....
    /*
    Relation r = new Relation ("." + name, newDoms, RELATION, 0, newDoms.length, 0, null, env);
    env.put(r);
    */  // original code being commented out
    doment = VirtualNestedRelationCreation(doment, node, newDoms, env);  // new code
    }
    return doment;
}
```

Figure 6.13: Modifications in *newVirDomain()*

In the method *newVirDomain()*, in lieu of creating the dot relation for the new declared domain and putting it into the relation table, the method *VirtualNestedRelationCreation()* is invoked. The purpose of the method is to check whether or not the nested relation has been defined the first time; if it has not, then check whether or not the domains of the redefined nested relation are the same as the previous nested relation. Different approaches are adopted to different cases.

In the first case, if the nested relation is a new one, the dot relation is created and both the domain and the relation are stored in the system table. (This is basically what the previous implementation does.)

In the second case, if the nested relation has been declared and its domains are the same as the redefined one, then do nothing (and, the new actualized data will be appended to the dot relation).

In the third case, if the nested relation has been declared, but is defined on different domains than the old one, then a new domain and a dot relation are created and put into the system tables. The name of the domain is the name of the original domain but suffixed with a number. The number indicates the next available name for the domain. For instance, as seen in Section 4.7.2, initially, domain V is defined to be $V(A)$. Later, V is redefined to be $V(B)$, then a new domain *V_0* and dot relation *.V_0* are created and put into the system table. If domain V is redefined again, to be $V(C)$, then the next available name for V is *V_1*, and *.V_1* for the dot relation, so on and so forth. The process to find the name of a matched nested relation for a virtual nested relation is performed in the method *LookupDotRelation()*. It takes the name of the virtual nested relation and the domains on which the new virtual nested relation is defined, as its input parameter, and creates or finds the name of the relation which has the same domains.

In the actualization of the redefined nested relation, the name of the virtual nested relation which is used by users and has no suffix is examined by invoking the same *LookupDotRelation()* method. The name of the matched domain whose dot relation contains domains equal to the domains of the redefinition will be found and used in the later process.

6.9 The Wildcard

The wildcard represents top level relations or nested relations of the current level relation. In both cases, each relation will be checked against the domains involved in the query, and only those relations in which the domains can be actualized will be selected as the relation which the wildcard represents. The wildcard is evaluated in the method *WildcardToRelations()*.

In this method, relations are selected by looking up the relation table, *NRelName* (please refer to Section 6.5.1 for detail) to obtain the name of the current level relation. The two possible results, get a null value or get a relation name from the vector correspond to the two different cases mentioned above.

In one case, if the relation table is empty, all relations in the system table will be looked up. Take the query in Figure 4.39 as an example. For each top level relation, *X1*, *X2*, and *X3*, the domains of the projection operation will be checked with the domains of the relation. If the domains of the relation contain all projector domains, then the relation will be added to a temporary relation table. If the projector domains contain virtual domains, relations where the virtual domains can be actualized will be selected. The functionality of checking whether or not a set of domains can be actualized in a relation is performed in the method *actualizableInRel()*.

For the other case, if the relation table contains names of relations, then the name of the current level relation is obtained according to the table index. The nested relations of the relation are obtained by checking each of the domains of the relation. If the domain is of the type IDLIST, then the dot relation is obtained and the domains of the dot relation are compared against the domains of the projection operation. It is the same as the implementation for the top level relation. By invoking the method *actualizableInRel()*, nested relations which may replace the wildcard are selected.

A node for each selected relation will be created and put to a node array and returned to the routine invoking.

Figure 6.14 shows the pseudo code of the method.

Interpreting a wildcard in a projection operation in normal expression is accomplished by the method *WildcardInProject()*. It is invoked by the method *evaluateProject()* before the expression of source relation is evaluated. In *WildcardInProject()*, the input node first is checked if it is a wildcard. No change occurs if the node is not a wildcard. But if the node is a wildcard, then the relations which the wildcard represents are selected by invoking the method *WildcardToRelations()*. Upon receiving an array containing relation which may represent the wildcard, if only one relation is in the relation table,

```
SimpleNode[] WildcardToRelations(SimpleNode exprnode, SimpleNode node, Environment env)
{
    relName = get the relation name from NRelName;
    if(relName==null){ // "." represents top level relations
        for each top level relation in system table,
            if the projector domains are actualizable in the relation
                add the relation to the temporary vector
    }else { // look up nested relations
        get the relation name from the name vector and get the relation;
        for each nested relation in the relation,
            if the projector domains are actualizable in the relation
                add the relation to the temporary vector
    }
    for each relation in the temporary vector {
        create a node of type IDENTIFIER, contains the name of the relation;
        put it to a node array;
    }
    return the node array;
}
```

Figure 6.14: Pseudo-code for *WildcardToRelations()*

then the wildcard is replaced by this relation. If there are more than one relation in the table, then for each relation, a new tree of projection operation is created, with the node of the relation name as the source node of the projection. These projection trees are combined together by the **ujoin** operation node. Finally, a new tree containing the **ujoin** of projection operations replaces the original projection tree which contains the wildcard as the source node. Figure 6.15 shows the syntax tree for the original query with wildcard and Figure 6.16 is the syntax tree for the equivalent **ujoin** query.

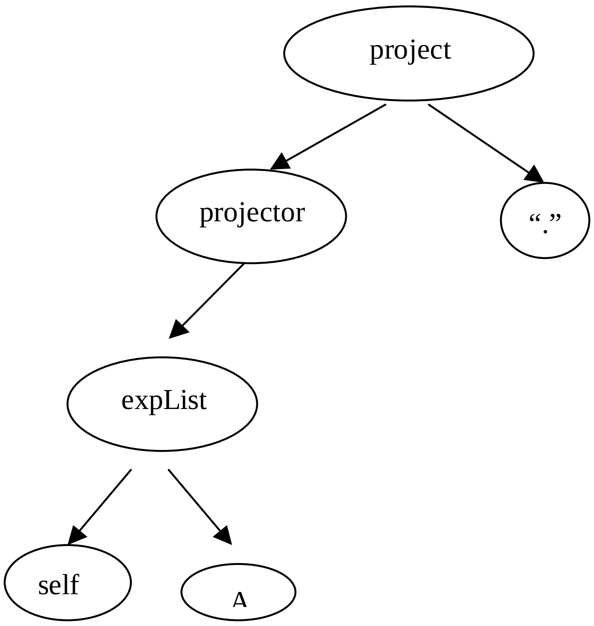

Figure 6.15: Wildcard in Projection

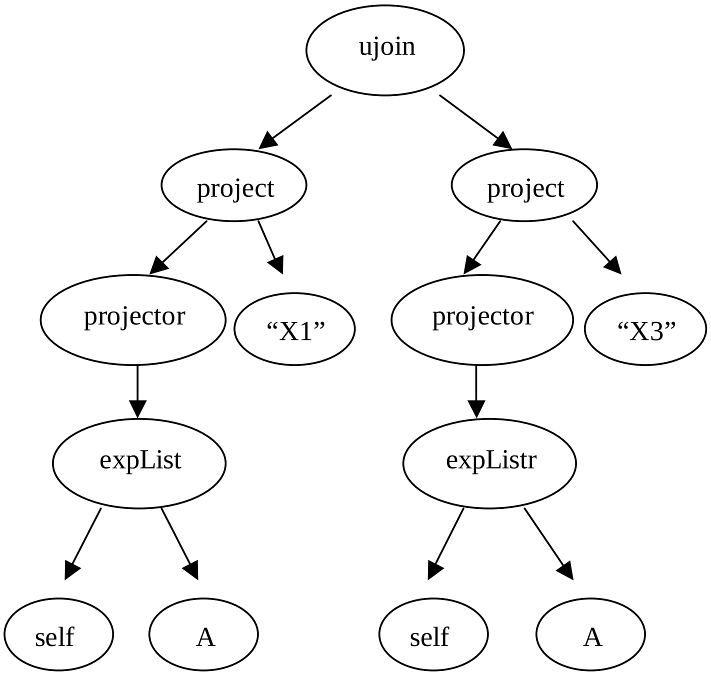

Figure 6.16: The Syntax Tree for Equivalent Operations in Figure 6.15

6.10 Recursive Virtual Nested Relations

A recursive virtual nested relation is a nested relation declared on attributes, one of which is the nested relation itself. It is declared with a wildcard so that it can be actualized in each nested relation of the upper level relation. A recursive virtual nested relation node is expanded to a tree in run time when the recursive domain is actualized in the first tuple of a relation. The expansion of the recursive node replaces the node in the definition expression with the recursive definition itself. For each nested relation in each level, the replacement of the tree happens once. We use different names for the recursive nested relation in different levels to embody the hierarchy information of the source relation. The name is the name of the recursive nested relation suffixed with a number, which indicates the level of the source relation in which it is actualized. The process of expanding the tree is to use the lower level tree to define the current level recursive nested relation. So we use a bottom-up strategy to do the expansion. The following describes the expanding process.

Before doing the specific expansion in each level, the total nested level of the source relation is obtained by invoking the method *getNestedLevel()*. With the nested level number, the process of expansion can be implemented in a loop. In each iteration of the loop from the lowest level to the top level, the following five steps are taken:

First, the name of the recursive nested domain in the current level is obtained;

Second, a copy of the definition tree is made;

Third, the recursive node and the order number as a child node of its parent is obtained;

Fourth, the node with the lower level definition tree is replaced. (This is how the tree is expanded.) If this is the lowest level, then the recursive node in the tree is removed; otherwise, a node is created with the name of the lower level recursive name and then the recursive node in the tree with this node is replaced;

Fifth, a domain with the name and the lower level definition tree is generated.

Take the query in Figure 4.45 as an example. The recursive nested domain *schema* is expanded three times when it is actualized in relation O, which has three level nested relations. The level of the recursion is three corresponding to the relation O, P, and Q. We do not need to know the name of these relations at this moment and the wildcard in the virtual nested relation definition tree will be left untouched. Figure 6.17 through Figure 6.20 show the expansion procedure. The arrow pointing to the node of the wildcard with a node of a relation in Figures 6.17, 6.18, and 6.19 indicate the relations that will replace the wildcard in the later process. Table 6.9 is a summary of this expansion, which includes the expansion level, the new name of the virtual nested relation used in that level, the source relation in which the virtual nested relation at the level will be actualized later, the action in the defined tree in the current level, and the corresponding reference figure which shows the process.

Note that the wildcard in the definition tree of the lowest level is kept, although the node of the virtual nested relation in the tree is removed. The wildcard will be replaced with a *null* value in the method dealing with a wildcard discussed in Section 6.9.

The top level virtual nested relation is generally defined to

domain *schema*(*sattri*, *schema*);

before the definition of the virtual nested relation. And it is redefined to

domain *schema*(*sattri*, *schema_1*);

level	domain name	relation	action in the definition tree	figure of definition tree
2	*schema_2*	Q	remove the recursive node	Figure 6.17
1	*schema_1*	P	replace the recursive node with node schema_2	Figure 6.18
0	*schema_0*	O	replace the recursive node with node schema_1	Figure 6.19

Table 6.9: Summary of Recursive Nested Relation Expansions in Relation O

in the first level expansion, as we have seen in Figure 6.19. Therefore, we need to rename the virtual nested relation, as well as modify the query tree, as shown in Figure 6.20.

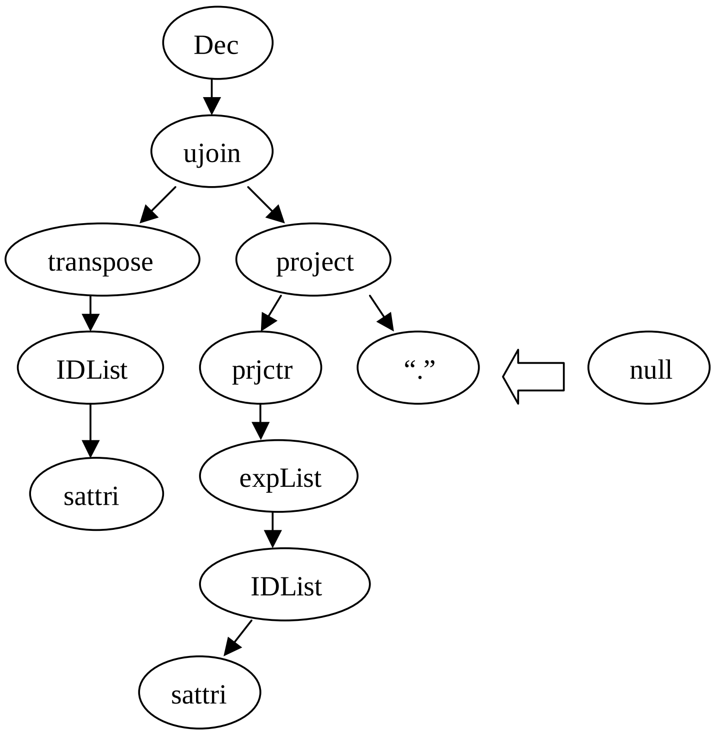

Figure 6.17: The Definition Tree of Domain *schema_2*

The expansion of the recursive nested relation is performed in the method *ExpandRecursiveNode()*.

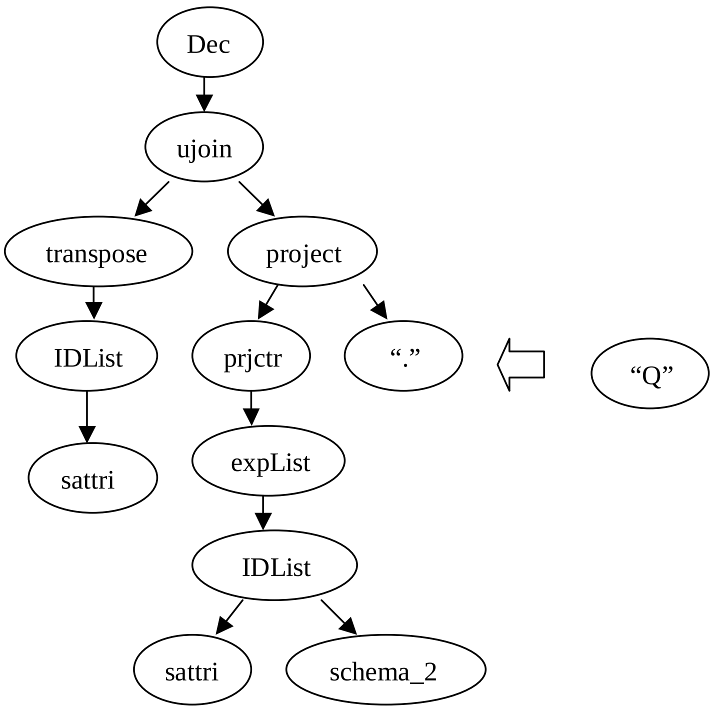

Figure 6.18: The Definition Tree of Domain *schema_1*

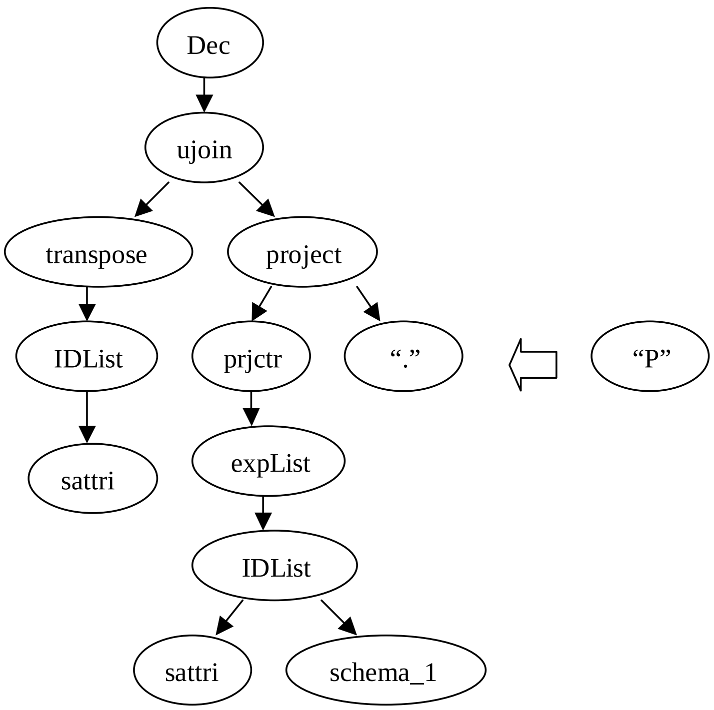

Figure 6.19: The Definition Tree of Domain *schema_0*

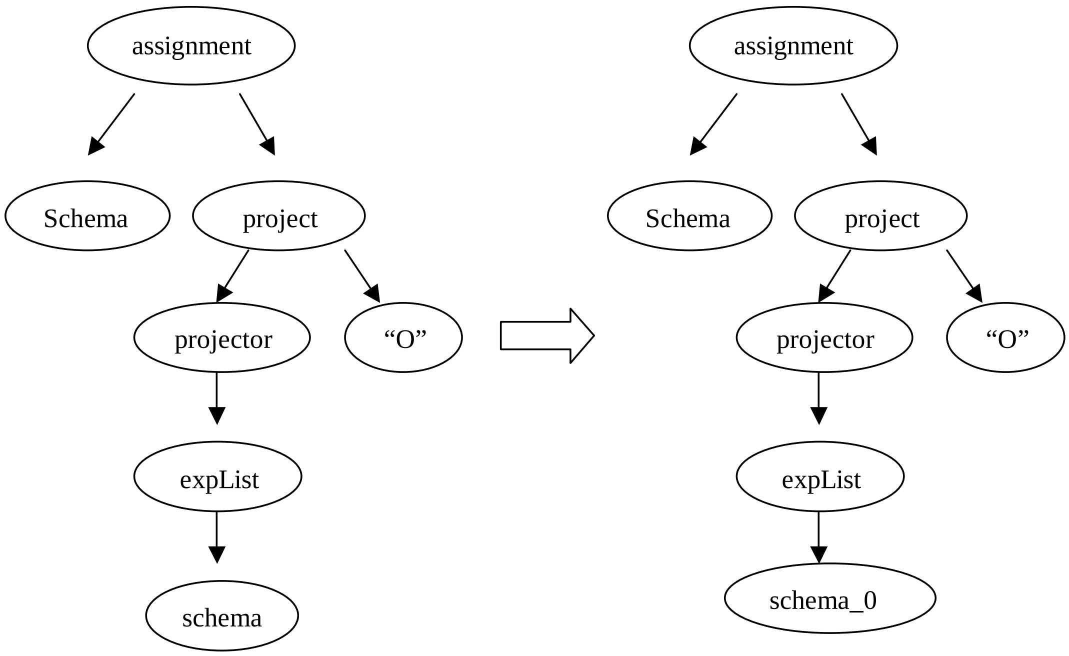

Figure 6.20: The Modification of the Top Level Nested Relation

Chapter 7

Conclusion and Future Work

This chapter provides a summary of the work that has been accomplished, followed by a discussion of future work.

7.1 Thesis Summary

This thesis presents the design and implementation of new features of attribute metadata in jRelix to support semistructured data.

Operator **quote** suppresses the evaluation of an attribute. It has been augmented to be applicable to relational expressions.

Operator **eval** evaluates the data of an attribute of type ATTRIBUTE. It has been enhanced to be applied to both relational algebra and domain algebra. In the domain algebra, the **eval** operator can be applied either to the virtual domain to be defined or to the value that is to be assigned to a virtual domain. When **eval** appears in the left side of the virtual domain definition statement, the value that is assigned to the attributes which it produces could be of any value that is valid for a domain.

The **transpose** operator generates data of all scalar attributes of a source relation. It has been re-implemented with a new version of syntax, which focuses on the data (attributes and their types and values) of the attributes to be transposed. The selection of attributes to be transposed is left to the selection operation of the relational algebra and the newly developed **quote** operation is required in this situation.

The functionality of the wildcard has been enhanced and it can be part of a domain expression. A wildcard can represent top-level or nested relations in a particular relation. One of the applications of the wildcard is in recursive virtual nested relations so that the corresponding operations are performed on each level of nested relations.

The **typeof** operator is implemented to obtain the type of an attribute. This is especially useful when the attribute is of type UNIVERSAL or UNION, whose data type may vary from tuple to tuple. A new data type TYPE is allowed to support metadata operators.

The **relation** operator is implemented to group attributes into a nested relation. The functionality of the nested relation declaration has been augmented with the ability to deal with redefining virtual nested relation and therefore this redefinition will not wipe out the data of any previously defined nested relation.

The **self** operator is implemented to trace the parent relation of an attribute. Users can now find the attributes as well as the parent relation of a nested relation by using this operator. The **self** will return the first named parent of the relation if it is applied to an unnamed relation.

By the combination of some of the above operators, together with the operations from the original system, attribute path and schema discovery and data reorganization can be done along with queries involving transitive closure on graphical structured data.

7.2 Future Work

7.2.1 Links

Our implementations so far are based upon the tree structure of data. Data is organized in hierarchies and no data sharing is allowed under this structure. But in practice, data sharing is important because it not only can save storage space but also can avoid updating inconsistencies: several subtrees which have the same content of data may not be updated simultaneously by accident. Using the link technique, which include labels and pointers, the above problem can be solved. A label is used for the data that is to be shared and pointers are used for pointing from other places where the data is needed. A pointer is defined as an “attribute: label” pair. The possibility of implementing data sharing relies on the fact that the implementation of nested relations in jRelix using a flat

Company

(name	Supplier				)
	(product	name	Supplier		)
			(product	name	Supplier)
Depco	pumps	AGI	rubber	Kenko	dc
			plastics	PIAB	dc
	wraps	Unaflex	dc		
AGI	rubber	Kenko	dc		
	plastics	PIAB	dc		

Table 7.1: The Display Form of Relation *Company*

relation does not pose restrictions on a directed cyclic graph (DAG) and cycles [Mer03]. In the following two subsections, examples of links, one for common sub expressions and the other for cross-references, are investigate.

Example 1: Common Subexpressions

If the data is organized under a strict tree structure, then the following relation *Company*, which contains name of companies and their products' providers, will have repeated data for company *AGI* as shown in Table 7.1. The data for *AGI* has different entry levels, one is as a company and the other is as a supplier of another company *Depco*. The actual storage taken for the relation also has repeated data represented by different surrogates which link to the actual data as shown in Figure 7.1.

```
Company
(name   Supplier)
 Depco      2
 AGI        3

Supplier
(.id product   name      Supplier)
  2   pumps     AGI         4
  2   wraps     Unaflex     dc
  3   rubber    Kenko       dc
  3   plastics  PIAB        dc
  4   rubber    Kenko       dc
  4   plastics  PIAB        dc
```

Figure 7.1: Storage of Relation *Company*

On the other hand, if the link is used, i.e., one of the *AGI* data entries is assigned a label, and the data of attribute *Supplier* in the other entry is replaced with a pointer which points to the label, as shown in Figure 7.2, then there will be no data repetition in the relation.

Company

(name	Supplier					)
	(product	name	Supplier			)
				(product	name	Supplier)
Depco	pumps	AGI	AGI:	rubber	Kenko	dc
				plastics	PIAB	dc
	wraps	Unaflex		dc		
AGI	Supplier:AGI					

Table 7.2: The Display Form of Relation *Suppliers*: Using Links

The implementation proceeds with the same idea: the entry for the same data uses the same surrogates. The above relation then can be stored in the way shown in Table 7.2. (By the way, Table 7.4 is an example of how to write the link the other way around.)

```
Company
(name   Supplier)
 Depco      2
 AGI        3

Supplier
(.id product   name      Supplier)
  2   pumps     AGI          3
  2   wraps     Unaflex      dc
  3   rubber    Kenko        dc
  3   plastics  PIAB         dc
```

Figure 7.2: Relation *Company*: Data Sharing

In jRelix, such data that has labels can be represented as xML where the labels are the attributes of xML documents and the pointers are "attribute : label" pairs (here the attribute is in the sense of a relation). The relation *Company* then can be in form of a file as in Figure 7.3. To avoid ambiguity that may be produced by omitted data, the hidden <.tuple> and </.tuple> pair is used to separate tuples.

Example 2: Cross-references

The above example exhibits the possibility for using links to deal with data sharing of common subexpressions in a relation. Cyclic data also can use links. The following example presents such a possibility. Table 7.3 shows the relation *Papers* without data sharing. Table 7.4 presents the ideal representation.

The reference of papers are papers, therefore, the link in the reference refers back

```
<Company>
    <.tuple>
        <name> Depco </name>
        <Supplier>
            <.tuple>
                <product> pumps </product>
                <name> AGI </name>
                <Supplier subex = "AGI">
                    <.tuple>
                        <product> rubber </product>
                        <name> Kenko </name>
                    </.tuple>
                    <.tuple>
                        <product> plastics </product>
                        <name> PIAB </name>
                    </.tuple>
                </Supplier>
            </.tuple>
            <.tuple>
                <product> wraps </product>
                <name> Unaflex </name>
            </.tuple>
        </Supplier>
    </.tuple>
    <.tuple>
        <name> Depco </name>
        <Supplier: AGI/>
    </.tuple>
</Company>
```

Figure 7.3: Relation *Company*: xML Representation

Papers

(title	author	reference			)
		(title	author	reference)	
Semistructured Data System		Semistructured Data Overview	William	dc	
Query XML	Cole	dc			
Semistructured Data Overview	William	dc			

Table 7.3: The Display Form of Relation *Papers*

Papers

Papers	(title	author	reference	)
dc	Semistructured Data System	Stone	Papers: SDOverview	
dc	Query XML	Cole	dc	
SDOverview:	Semistructured Data Overview	William	dc	

Table 7.4: The Display Form of Relation *Papers*

to *Papers*. This example presents the cycle of schema. In fact, even the data cycle is possible. [Mer03] gives detailed explanations with an example. This can be implemented by extending the outer relation *Papers* with an attribute *.id*, which contains surrogates for data entries that have to be shared. Figure 7.4 shows the storage form of the relation.

```
Papers
(.id  title                         author  reference)
  dc  Semistructured Data System    Stone     34
  dc  Query XML                     Cole      dc
  34  Semistructured Data Overview  William   dc
```

Figure 7.4: Relation *Papers*: Data Sharing

Figure 7.5 exhibits the file of relation *Paper* to be represented in xML form.

```
<Papers>
    <.tuple>
        <title> Semistructured Data System </title>
        <author> Stone </author>
        <reference:Papers:SDOverview>
    </.tuple>
    <.tuple>
        <title> Query XML </title>
        <author> Cole </author>
    </.tuple>
    <.tuple coex=SDOverview>
        <title> Semistructured Data Overview </title>
        <author> Eric </author>
    </.tuple>
</Papers>
```

Figure 7.5: Relation *Papers*: xML representation

7.2.2 One More Extension of the Wildcard

The wildcard implemented is for representation of top level relations or nested relations. The relations that can be represented are those that contain all the actual attributes in the query or those in which all virtual attributes can be actualized. One more functionality of the wildcard could be that it represents all attributes in a relation if no attribute is specified in the query. The result of a query involving a wildcard under this circumstance is the **ujoin** of the relations resulting from the specified operations applied on each attribute. Take the relation in Figure 5.8 as an example and consider the following query:

```
PaperbyYear' <- [ .  ]in PaperbyYear;
```

This query is expected to transform the sparse relation into a singleton relation as shown in Figure 7.6 (for nested relation *.2000* and *.2001*, please refer back to Figure 5.8 in Section 5.2).

```
PaperbyYear'
+--------------------+--------------------+
| 2000               | 2001               |
+--------------------+--------------------+
| 22                 | 23                 |
+--------------------+--------------------+
```

Figure 7.6: Relation *PaperbyYear'*

The wildcard in the above query represents both attributes *2000* and *2001*. The query is equivalent to:

PaperbyYear' <- ([*2000*] **in** *PaperbyYear*)
ujoin ([*2001*] **in** *PaperbyYear*);

This assumes that the projection operation will eliminate the "dc" value in a unary relation. For instance, if we have a relation R as shown in Figure 7.7, we expect that the projection operation on the attribute A will disregard the "dc" value and generate a singleton relation as in Figure 7.8. A little modification of the null value behavior in the current jRelix is needed.

```
>domain A,B strg;
>relation R(A,B)<-{("a",dc),(dc,"b")};
>pr R;
+----------------------+----------------------+
| A                    | B                    |
+----------------------+----------------------+
| dc                   | b                    |
| a                    | dc                   |
+----------------------+----------------------+
relation R has 2 tuples
>
```

Figure 7.7: Relation R Involving the "dc" Value

```
RA<-[A] in R;
RA
+----------------------+
| A                    |
+----------------------+
| a                    |
+----------------------+
```

Figure 7.8: The Expected Result

7.2.3 Integrate Attribute Metadata Operator into *Update* Operation

The operators for supporting semistructured data discussed and implemented have focused on data retrieval and schema discovery. One other aspect of data management could be schema modifications. The existing jRelix system has mechanisms to perform database modifications. Because the attributes have been treated as data with extra operators, we can integrate the attribute metadata operators into the **update** operator syntax.

For example, we expect to perform the following statement to add or remove the attribute C from relation R.

update `R` **add** **quote** `C`;

or

update `R` **delete** **quote** `C`;

The first statement is usually followed by further operations such as the **change** operation in **update** to accomplish a complete update of a relation. The second statement generates the same result as a projection operation except that it is performed on the original relation. Accordingly, no new relation is generated and the attribute deleted is permanently lost along with its data. Merrett discusses an alternative way for updating relation schema. Readers can refer to [Mer03] for the detail.

Bibliography

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Systems*, 1(2):97–137, 1976.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholmand, W. P. Cockshott, and R. Morrision. Ps-algol: A language for persistent programming. In *In In 10th Austrian National Computer conference*, pages 70–79, Melbourne, Australia, 1983.

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory*, pages 1–18. Springer-Verlag, January 08-10 1997.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. Data on the web: From relations to semistructured data and XML. Morgan Kaufmann, San Francisco, 2000.

[ACC+97] Serge Abiteboul, Sophie Cluet, Vassilis Christophides, Tova Milo, Guido Moerkotte, and Jme Simeon. Querying documents in object databases. *Int. J. on Digital Libraries*, 1(1):5–19, 1997.

[AM87] Andrew W. Appel and David B. MacQueen. A standard ML compiler. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119, San Antonio, Texas, 1979. ACM Press.

[Bak98] Patrick Baker. Design and implementation of database computations in Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 1998.

[BBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *Proceedings of the 4th International Conference on Database Theory (ICDT)*, volume 646, pages 140–154, Berlin, Germany, October 1992. Springer-Verlag.

[BCF+04] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. *http://www.w3.org/TR/xquery/*, 2004.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 505–516, Montreal, Quebec, Canada, June 04-06 1996. ACM Press.

[BDS95] Peter Buneman, Susan B. Davidson, and Dan Suciu. Programming constructs for unstructured data. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, page 12, Gubbio, Umbria, Italy, 6-8 September 1995. Springer-Verlag.

[BFS00] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, 2000.

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Male, and François Yergeau. Extensible markup language (XML) 1.0 (third edition). *http://www.w3.org/TR/REC-xml/*, 2004.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. *SIGMOD Rec*, 23(2), 1994.

[CCS94] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E.F. Codd and Associates, 1994.

[CD99] James Clark and Steve DeRose. XML path language XPath. *http://www.w3.org/TR/xpath*, 1999.

[Cha02] Andy S. Chang. Implementation of sigma-joins in a nested relational algebra, 2002. Master's Report, School of Computer Science, McGill University, Montreal, Canada.

[Cla99] James Clark. XSL transformations. *http://www.w3.org/TR/xslt*, 1999.

[CM82] A. J. Cole and R. Morrison. An introduction to programming with s-algol. Cambridge University Press, 1982.

[CM90] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proc. of the ACM Symp. on Principles of Database Systems, PODS'90*, pages 406–416, 1990.

[CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 323–330, San Francisco, California, United States, 1987. ACM Press.

[CMW88] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: Recursive queries without recursion. In *Expert Database Conf.*, pages 645–666, 1988.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. *http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/*, 1998.

[FG85] Patrick C. Fischer and Dirk Van Gucht. Determining when a structure is a nested relation. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases*, pages 171–180, Stockholm, Sweden, August 21-23 1985. Morgan Kaufmann.

[FW04] David C. Fallside and Priscilla Walmsley. XML schema part 0: Primer second edition. *http://www.w3.org/TR/xmlschema-0/*, 2004.

[GSS04] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. Oracle essentials: Oracle database 10g. O'Reilly Media, Inc., 2004.

[Gu05] Yu Gu. Basic operators for semistructured data in a relational programming language. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2005.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.

[Hao98] Biao Hao. Implementation of the nested relational algebra in Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 1998.

[HSW75] G. D. Held, M. R. Stonebraker, and E. Wong. INGRES: a relational database system. In *Proc. AFIPS National Computer Conference*, pages 409 – 416. AFIPS Press, 1975.

[Jac96] Dana Jacobsen. Bibtex. *http://www.ecst.csuchico.edu/∼jacobsd/bib/formats/bibtex.html*, December 1996.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proceedings of the First ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, 1982.

[Kan01] Sungsoo Kang. Implementation of functional mapping in nested relation algebra, 2001. Master's Report, School of Computer Science, McGill University, Montreal, Canada.

[KKS92] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 393–402, San Diego, California, United States, 1992. ACM Press.

[KS95] David Konopnicki and Oded Shmueli. W3QS: A query system for the world wide web. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *21st Conference on Very Large Databases*, pages 54–65, Zrich, Switzerland, 1995.

[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the WEB. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering (RIDE'96) Interoperability of Nontraditional Database Systems*, pages 12–21. IEEE Computer Society, 1996.

[MAG$^+$97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of the 3rd International Conference on Very Large Data Bases*, pages 447 – 453, Tokyo, Japan, 1977.

[MBC$^+$02] T. H. Merrett, Y. Bédard, D. J. Coleman, J. Han, B. Moulin, B. Nickerson, and C. V. Tao. A tutorial on database technology for geospatial applications. *http://www.cs.mcgill.ca/∼tim/geodem/tutorial.ps.gz*, May 27 2002.

[McB94] Oliver A. McBryan. Genvl and www: Tools for taming the web. In *In Proc. 1st Intl. World Wide Web Conf*, Geneva, Switzerland, May 1994.

[Mer76] T. H. Merrett. MRDS: An algebraic relational database system. In *Canadian Computer Conference*, pages 102–124, Montreal, Canada, 1976.

[Mer77] T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, 1977.

[Mer84] T.H. Merrett. *Relational Information Systems*. Reston Publishing Co., Reston, VA, 1984.

[Mer88] T. H. Merrett. Experience with the domain algebra. In Umeshwar Dayal Catriel Beeri, Joachim W. Schmidt, editor, *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness,*, pages 335–346, June 1988.

[Mer91] T. H. Merrett. Relixpert–an expert system shell written in a database programming language. *Data Knowl. Eng.*, 6(2):151–158, 1991.

[Mer01] T. H. Merrett. Attribute metadata for relational OLAP and data mining. In *Proceedings of the Eighth Biennial Workshop on Data Bases and Programming Languages*, pages 65–76, Monteporzio Catone, Roma, Italy, September 2001.

[Mer03] T. H. Merrett. A nested relation implementation for semistructured data. *http://www.cs.mcgill.ca/∼tim/semistruc/recnest.ps.gz*, December 1 2003.

[ML94] Michael L. Mauldin and John R. R. Leavitt. Web agent related research at the center for machine translation. In *SIGNIDR meeting*, McLean, Virginia, August 1994.

[MMM97] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.

[Mor79] R. Morrison. S-algol language reference manual. Technical report, Dept. of Computational Science, Univ. of St Andrews, Scotland, 1979.

[NAM97] Svetlozer Nestorov, Serge Abiteboul, and Rajeev Motwani. Inferring structure in semistructured data. *SIGMOD Rec.*, 26(4):39–43, 1997.

[NUWC97] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 79–90, 1997.

[OFS84] J. Ong, D. Fogg, and M. Stonebraker. Implementation of data abstraction in the relational database system INGRES. *SIGMOD Records*, 14(1):1–14, March 1984.

[OH86] S. L. Osborn and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Transactions on Database Systems*, 11(3):357–373, September 1986.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[QRS+95] Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Querying semistructured heterogeneous information. In *DOOD '95: Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 319–344. Springer-Verlag, 1995.

[RCF00] Jonathan Robie, Don Chamberlin, and Daniela Florescu. Quilt: an XML query language. *http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html*, 2000.

[RH98] Dave Raggett and Arnaud Le Hors. HTML 4.0 specification. *http://www.w3.org/TR/WD-html40/*, 1998.

[Roz02] Andrey Rozenberg. Implementation of attribute metadata with application to data mining, 2002. Master's Report, School of Computer Science McGill University, Montreal, Canada.

[Sch77] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. Database Syst.*, 2(3):247–261, 1977.

[SDV03] Sriram Sankar, Rob Duncan, and Sreenivasa Viswanadha. Java Compiler Compiler (JavaCC)-The Java Parser Generator. JavaCC web site at: www.webgain.com/products/javacc/documentation.html, 2003. The web site contains documentation softwares for JavaCC and JJTree.

[SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of abstract data types and abstract indices to CAD databases. In *Proceedings of Database Week, Engineering Design Applications*, pages 107–114, San Jose, May 1983.

[Sto86] M. Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of the 2nd IEEE Data Engineering Conference*, Los Angeles, 1986.

[Sun00] Weizhong Sun. Updates and events in a nested relational programming language. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2000.

[TF86] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis, editor, *Advances in Computing Research III, The Theory of Databases*, pages 269–307. JAI Press, 1986.

[TMD92] Jean Thierry-Mieg and Richard Durbin. acedb — a c.elegans database syntactic definitions for the acedb data base manager. *http://www.acedb.org/Cornell/syntax.html*, December 1992.

[Wan02] Zhongyan Wang. Implementation of distributed data processing in a database programming language. Master's thesis, School of Computer Science McGill University, Montreal, Canada, 2002.

[Xie04] Jiantao Xie. Text operators in a relational programming language. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2004.

[Yan03] YiYi Yang. Casting and null values for numeric database types, 2003. Master's Report, School of Computer Science, McGill University, Montreal, Canada.

[Yu04] Zhan Yu. Implementation of recursively nested relation of jrelix, 2004. Master's Report, School of Computer Science, McGill University, Montreal, Canada.

[Yua98] Zhongxia Yuan. Implementation of the domain algebra in Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 1998.

[Zhe02] Yi Zheng. Abstract data types and extended domain operations in a nested relational algebra. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2002.