Abstract Data Types and Extended Domain Operations in a Nested Relational Algebra

Yi Zheng

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

August, 2002

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 two enhancements to the Aldat database programming language: abstract data types (ADTs) and extensions to the domain algebra.

Utilizing a nested relational model and an improved procedural abstraction facility, ADTs are declared as computations encapsulating states with their accessor/modifier methods. Objects of an ADT can be instantiated via a single join. As computation calls are embedded into updates and virtual domain actualization, objects are manipulated and accessed solely through the methods exported by the ADT.

The vertical domain algebra empowers Aldat with the capability to combine values along a domain using system defined operators. A mechanism has now been installed to run user defined computations as well. This, coupled with ADTs, opens up the opportunity for Aldat to handle applications such as GIS which require at once the capacity of a traditional DBMS and the computational power of a modern programming language.

Résumé

Ce mémoire illustre la conception et l'implantation de deux perfectionnements apportés au langage de programmation de bases de données Aldat : les types de données abstraits (TDA) et les extensions de l'algèbre des domaines.

Lorsqu'on utilise un modèle relationnel imbriqué et une installation perfectionnée d'abstraction procédurale, on dit que les TDA sont des calculs qui encapsulent des états dans leurs mécanismes d'accès/modification. Les objets d'un TDA peuvent être instanciés par une seule jointure. Tandis que les $\ll$ computations $\gg$ (une forme d'appel de procédure généralisé propre à Aldat) sont imbriqués dans les mises à jour et l'actualisation des domaines virtuels, les objets sont manipulés et sollicités exclusivement par les méthodes exportées par le TDA.

Le domaine algèbre vertical permet à Aldat de combiner des valeurs le long d'un domaine en utilisant des opérateurs définis par le système. De nouveaux mécanismes ont maintenant été installés pour exécuter également les calculs définis par l'utilisateur. Conjugué aux TDA, cela permet à Aldat de recevoir des applications comme les SIG qui nécessitent à la fois la capacité d'un SGBD traditionnel et la puissance de calcul d'un langage de programmation moderne.

Acknowledgments

This thesis is not possible without the support from many individuals. First, I wish to thank my thesis supervisor Professor Tim Merrett for his guidance, advice, encouragement and financial support throughout the research and preparation of this thesis. I have benefited enormously from his valuable insights and mentoring on a great number of occasions. Under an impossibly tight time schedule, he scrutinized every aspect of the text and made countless suggestions for improving the accuracy and quality of this thesis.

Many thanks to my colleagues in Aldat lab who have provided consultation and comments to my research. Andrey Rozenberg kindly answered many of my questions about the system with extreme patience. Hongyu Zhao devoted long hours to testing and partial integration, which made my life much easier. I am fortunate to have worked with a talented group of graduate students in CS 617, the seminar course that led to the conception of this thesis. Zongyan Wang, Xuechun Lu, Yiyong Pan, and Qifang Zheng all offered precious suggestions and constructive criticism during my presentations of the topics covered by this thesis.

I wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to Diti Anastasopoulos, Vicki Keirl, Lise Minogue, and Lucy St-James, for easing the procedures of dealing with the school.

Last but not least, I owe special thanks to my loving husband Wei Xu, for his unconditional support, understanding and sacrifice during my study.

Chapter 1

Introduction

This thesis describes two enhancements to the Aldat database programming language — abstract data types (ADTs) and extensions to the vertical domain algebra. The motivation for this work is given in Section 1.1. In Section 1.2, we briefly outline the structure of the thesis.

1.1 Motivation for the Thesis

The relational data model, first introduced by Codd [Cod70], has attracted much attention from both academia and industry. Relational database systems have improved the application development process in large data-intensive environments by providing a single, uniform view of data expressed in structure-independent terms (i.e. relations, tuples, domains, etc.). Other benefits include facilities for controlled sharing of data, system controlled data integrity maintenance, and highly tuned routines for data formatting and access [AH90].

The power of the basic relational model lies in its decriptive ability, however, not in its computing ability, as admitted by Codd himself in his seminal paper. Codd envisioned a "data sublanguage" to be developed on the basis of the relational model and embedded in a variety of host languages. Arithmetic functions needed in the qualification for data retrieval, for example, were deemed appropriate in the host language rather than

the data sublanguage. One problem in developing database applications using this "two-language" approach is the *impedance mismatch* between the data manipulation language (DML) of the database and the general purpose programming language (PL) in which the rest of the application is written [BM88]. One aspect of the mismatch is the difference between the declarative paradigm of DML and the imperative nature of PL. The other aspect is the mismatch of type systems.

On the other hand, increasingly, requirements have emerged for database systems to handle new data types and their associated operations. Many applications, exemplified by office automation, computer-aided design and geographic information systems, could benefit from databases capable of handling complex objects typically used in programming these systems. Such non-business applications not only need a database to archive huge amounts of data, but also to provide extensibility to capture domain-specific data semantics. The basic model would clearly not be appropriate for such applications without some enhancements.

Since the 1980's there has been a significant trend in database research addressing the inadequacy of the basic relational model. Various extensions to the base type system of databases have been explored. Nested relations [Mak77] were proposed to offer a direct mapping from applications with hierarchical structures to databases. More significantly, the concept of data abstraction [LZ74, Gut77] was adapted from the programming language paradigm to the database systems [SRG83, OFS84], thus enabling databases to answer the call of modelling complex objects.

The problem of impedance mismatch, however, still needs to be addressed. To this end, a new breed of languages, *database programming languages*, have been proposed. This type of language either incorporates the database types and operations into a programming language, or extends a database system with programming language constructs. However, both of these ways threaten to shift the mismatch from the syntactic to the conceptual level [Mer93].

At McGill, a project called Aldat (standing for the *Algebraic approach to data*) has produced several implementations of a relational database programming language. The

approach taken by Aldat researchers is to study the similarities and differences between database and programming language concepts, and then provide generalizations to bridge the gap. The earlier versions of the language, under the common name *Relix*, not only fully implemented the relational model proposed by Codd, but also extended it with the *domain algebra* [Mer76], a collection of operations on attribute values, orthogonal to the relational algebra. This enhancement empowered Relix with versatile arithmetic, grouping, ordering, and aggregation capabilities, most of which are not seen in commercial SQL implementations. Procedural abstraction facilities (functions and procedure) and one level of nesting were also implemented in Relix. A recent incarnation of Relix, called *jRelix* (Relix in Java) [Yua98, Hao98, Bak98, Sun00, Roz02, Cha02, Zha02, Kan01], provides full support for the deeply nested relational model by subsuming the relational algebra into the domain algebra. It also has integrated support for updates and event handlers. The two procedural abstraction facilities in Relix have been merged into one, now called *computation*. It is a special case of relation and can be manipulated by the relational algebra.

With the support for nested relations and procedural abstraction in place, what is now left to be implemented is a mechanism for data abstraction. The introduction of user-defined data types to model real-world complex objects, such as bank accounts or maps, will make a large or complicated program more manageable to the application programmer. This thesis is mainly devoted to the design and implementation of the data abstraction mechanism in jRelix. It also discusses extensions to the domain algebra that further enhance the flexibility of the system in coping with custom domain types.

The approach of our work is conservative in that we strive to keep the characteristics of the relational model. Other research groups have started out developing completely different models and systems. Object-oriented databases are one of them. Although they have been heralded as capable of overcoming all the obstacles faced by the traditional relational model [MS90, AH90, BCG+90, MD90, LRV90], we have found, in the course of work on Aldat, that all the important features of the object orientation can be implemented in our system with few or even no new concepts. A brief discussion of

this aspect can be found in the conclusions to this thesis. For this reason, we will not elaborate on object-oriented databases in later chapters. Interested readers may consult the references listed above.

1.2 Thesis Outline

This introductory chapter has indicated the intent and purpose of this thesis. The next chapter reviews literature on the relational model, its extensions, and the concept of data abstraction. A brief overview of the Aldat project and the two main versions of the Relix language is also given. Chapter 3 presents the use of the relational and domain operations supported by jRelix in a tutorial fashion. Newly incorporated features, namely abstract data types and extended domain operations, are illustrated by examples in Chapter 4. Chapters 5 and 6 describe the implementation details of these new features. Finally, a summary of the work presented in this thesis is given in Chapter 7, along with suggestions for future work.

Chapter 2

Background and Related Work

This chapter contains a review of the literature on the relational data model, its extensions, and the concept of data abstraction. An overview of the related work in the Aldat project will also be given.

2.1 Relational Model

2.1.1 Flat Relational Model

The relational model introduced by Codd [Cod70] represents the database as a collection of time-varying relations. The relation is a simple and uniform data structure which consists of rows and columns. A relation resembles a table, in which each row contains a collection of related data values. The term "tuple" is used to refer to a row, and "attribute" refers to the header of a column. The data type of values that can appear in a column constitutes a "domain". A relation is formally defined as a subset of the Cartesian product of its domains.

Normal forms have been introduced on relations to reduce storage redundancy and miminize the effort of updates. Codd defined the rules for a relation to be in the first-normal-form (1NF), as follows:

- all tuples are distinct,
- the order of the tuples is immaterial,

- each attribute is unique and the ordering of columns is irrelevant,
- attribute values are atomic. That is, the values are no decomposable as far as the relation is concerned.

Relations satisfying these requirements are also called flat relations. Codd dealt with the subject of normalization more vigorously in [Cod72a] and [Cod72b]. A series of higher normal forms have been introduced since then, which define increasingly stringent requirements. A thorough discussion about normalization techniques can be found in [Dat81] and [Ull82].

The two most widely used prototypes of the relational model were System R and INGRES, according to [SH98]. System R was developed at IBM's San Jose Research Laboratory in California during the late 70s [ABCea76]. INGRES was produced by a project at the university of California at Berkeley [HSW75]. Much of the current commercial landscape shows the influence of these systems. In particular, the query optimization architecture and optimization techniques of System R are generally lauded and form the basis of the algorithms used in most commercial systems. The Structured Query Language (SQL[1]) has its roots in System R. On the other hand, INGRES is highly regarded for the cleanliness of its relational sublanguage QUEL and the query modification algorithms for views, protection, and integrity control.

Relational Algebra

The relational algebra is first suggested in [Cod70]. It is a collection of operations applied on relations. All operations take relation(s) as operands and return a relation in result. This is called the "closure principle" of the relational algebra. By following this principle, it is possible to construct complex relational expressions using a series of simple operations.

Traditionally, the relational algebra has five independent operators, some of them generalized from the mathematical set operations: set union ($\cup$), set difference ($-$),

[1]SQL is now a firmly established relational database query language.

relational product ($\times$), relational selection (σ), and relational projection (π). More operators have been added by various extensions to the relational model. In general, operators can also be categorized according to the number of operands they require. Therefore, we can speak of *unary operators* (e.g. projection) and *binary operators* (e.g. joins).

Domain Algebra

The need for arithmetic and related operations on the value of attributes has given rise to the domain algebra [Mer76] which consists of two categories of operations for manipulating attribute values in tuples. They are:

- horizontal operations: new attribute value is calculated based on other attribute values within a tuple
  - constant
  - attribute renaming
  - unary operations, e.g. negation
  - binary operations, e.g. plus, minus, join
  - if-then-else
  - built-in functions
- vertical operations: new value is generated from values along an attribute
  - reduction
  - equivalence reduction
  - functional mapping
  - partial functional mapping

2.1.2 Extensions to the Relational Model

The relational model proved a great success in the world of business applications. However, it encountered obstacles in modeling complex data objects for non-business applications, such as geographic information systems and computer-aided design. For this and other practical reasons, a significant amount of work has been devoted to the extension of the relational model since the 1980's. In this section, we review some of these

extensions in three directions: (1) extensions to base types, (2) extensions to structure, and (3) extensions to query languages.

Extending Data Types

Traditionally, only a small number of atomic data types (or attribute domains) are offered for numbers and strings in a database system. Some systems include types for date, time and currency.

The first class of extension to the relational model addresses the base types a tuple is constructed from. One direction is to enrich the collection of base types, by including types such as those for geometric data (points, polylines and polygons), text and image. The other direction is to incorporate more operations on the attribute level. Commercial SQL typically allows arithmetic expressions to occur in a projector list, with or without aliasing. The domain algebra introduced in the previous section takes this direction a step further by establishing an algebra for attributes, orthogonal to the relational algebra.

A special case of the extension to base types is the inclusion of abstract data types (ADTs). Several research projects have incorporated ADTs into attribute domains. Since ADT is an important concept of its own right, we will discuss this case in Section 2.2.

A second class of data type extension concerns providing an extensible database system architecture. Some well-known projects in this area are documented in [CDRS86, DMB+87, LMP87, PSS+87]. Problems regarding access path support and special storage for user-defined data types were investigated by [WSSH88, Wol89]. Others explored mechanisms to enhance the database optimizer in view of the extended types [Fre87, GD87, Loh88].

Nested Relations

Nested relations [Mak77] are an attempt to extend the structure of the relational model without introducing new syntax. The First Normal Form restriction for a relation is

dropped; attributes need no longer be atomic, they can also be structured. In other words, relations can contain relation-valued attributes. Thus nested relations are also known as "non-first-normal-form ($(NF)^2$)" relations. In a sense, the nested relational model introduces some aspect of the hierarchical data model. The robustness of the nested relational model has been proven by [Mak77, SP82, FT83, AB84, SS86, KK89].

Following [Mak77], Jaeschke and Schek [JS82] introduced a generalization of the ordinary relational model by allowing relations with set-valued attributes and adding two restructuring operators, *nest* and *unnest*, to manipulate such (one-level) nested relations. Thomas and Fischer [TF86] generalized this model and allowed nested relations of arbitrary but fixed depth. The definition of recursively nested relations was discussed in [LS88].

The benefits offered by nested relations include:

- nested relations improve logical design by means of a more direct mapping of application onto the database,
- nested relations provide an elegant way of physical database design due to the possibility of internally materializing frequent joins,
- the conceptual gap between relational and nested relational models is minimal, and most theories and techniques established for the relational model apply in the case of nested relations as well.

[OY85] showed that nested relations provide a way of directly representing certain multi-valued dependencies by nesting. Therefore, fewer relations have to be split into smaller pieces during logical database design. In contrast, ordinary relational database design often ends up in a large number of tables due to the need for decomposition in the case of set-valued attributes. As a result, joins that are necessary to present all data in a traditional relational database become unnecessary with nested relations, as more data are contained in one tuple.

The benefits gained with nested relations in physical database design have been investigated by [SPS87]. One point of interest is the transformation of queries from the logical to the physical level by formal manipulation with a nested relational algebra.

[Bid87, Sch86] explored ways to recognize and eliminate joins from the users' queries in the process of such transformation. Some joins are unnecessary since they are internally materialized. Optimization problems associated with this type of query transformation are also solved in these papers, mainly with relational techniques.

Besides implementing nested relations in a one-to-one mapping to storage structures, there have been a number of attempts at other implementation choices. Due to space and time limit, we will not elaborate on them here, but refer the reader to [DKA+86, DvG88].

In the meantime, query languages have been enriched or invented for the nested relational model. We will cover this topic in the next sub-section.

Extended Query Languages

A relational query language is said to be relationally complete if it is at least as powerful as the relational algebra [Cod70]. However, more expressive power is often needed for practical purposes. Commercial DBMS often provides SQL with built-in functions, aggregates, ordering, grouping, and updates. Others have come up with recursive query facilities which take advantage of nested relations [Sch89, Jia90, Lin90].

An enormous amount of work was devoted to nested relational query languages. The initial papers on nested relations focused on the addition of two operators: *nest* and *unnest* [SP82, JS82, FT83]. The *nest* operator creates partitions based on equivalence classes. Tuples having the same value for some attributes are equivalent and all equivalent tuples are replaced with a single tuple in the resulting nested relation. The resulting relation is then defined on all the attributes for which equivalence is defined, and a nested attribute defined on the rest of the attributes of the original relation. The *unnest* operation reverses the effect of the *nest* operator. The motivation for adding the two operators was that whenever relation-valued attributes are encountered, one could first unnest, apply the standard relational operators and then nest to obtain the final result. This approach will not work in general, because the effect of an unnest may not be reversible by a nest [JS82, FT83]. The conditions under which a relation can be unnested losslessly was investigated by [FvG85].

Another approach to query nested relations is to apply relational algebra on the attribute level [AB84, SS86, OOM87, RKS88, Mer01]. The rationale is that since the nested relational model is a direct extension to the relational model, we do not need a new query language. We need, however, to allow relational operators to become applicable on the attribute level, as relations may now occur not only at the top-level, but also nested within tuples. [SS87] found that all relational operators can be nested into selections or projections. The valid operands of nested operators include sub-relations at each nesting level as well as top-level relations. The expressive power of such nested relational algebra has been shown in [Bee88]. In contrast to the "nest/unnest" approach, a nested algebra expression can be directly executed by a query processor, without resorting to extra procedures.

There have also been extensions made to languages like SQL. In general, the approach is to allow nested Select-From-Where blocks in both the projector list and the where clause. In particular, the where clause can now also refer to sub-relations. Examples of extended SQL-style languages are given by [Lar88, PA86, RKB87].

In jRelix, the database programming language that this thesis builds upon, the support for nested relations requires virtually no new concepts beyond the relational model. Our approach to nested relations is to allow attributes ("domains") to be relations and to subsume the relational algebra into the domain algebra. Examples arise in Sections 3.1.3 and 3.4.2 in Chapter 3, and in Chapter 4.

2.2 Abstract Data Types

2.2.1 ADT as a Programming Language Concept

The concept of data abstraction[2] [LZ74, Gut77] is one of the two fundamental kinds of abstraction[3] in contemporary programming languages. It is a weapon against complexity, a means of making large or complicated programs more manageable. An abstract

[2]In this thesis, we use data abstraction and abstract data type interchangeably

[3]The other is procedural abstraction

data type introduces a new type of data object which is deemed useful in the domain of the problem being solved. It is formally defined as a data type that satisfies the following two conditions (as per [Seb96]):

- *Encapsulation*: The representation of the type and the operations on objects of the type are contained in a single syntactic unit. Also, other program units can create objects of the defined type.
- *Information hiding*: The representation of objects of the type are hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the type's definition.

Encapsulation provides a method of organizing a program into logical units that can potentially be compiled separately. In addition, it allows any modifications on the representations or operations of the type to be done in a single area of the program.

One of the advantages of information hiding is that program units that use the type cannot "see" the representation details, thus their code cannot depend on that representation. As a result, the representation can be changed at any time without affecting the program units making use of the type[4]. Another important benefit of information hiding is increased reliability. Program units cannot change the underlying representation directly, either intentionally or by accident, therefore the integrity of such objects is protected.

A classic example is a stack ADT, which holds an internal storage for items and a memory for the current position of the top item. It also exposes an accessor method, *top()*, which tells what the top item is, and two modifier methods, *pop()* and *push()*, to make the stack shrink and grow, respectively. Programs using this ADT do not know or care whether the internal storage is an array or a linked list. However, they do rely on the abstract interface of the stack ADT, namely the three methods, to accomplish their programming tasks.

SIMULA 67 provided the first construct for encapsulating data objects with their operations — the class. However, the class construct does not support information

[4]Such program units may need to recompile, although no code change is involved

hiding [Seb96]. Today, most programming languages provide full support of ADTs, although the terminology differs from one language to another. For example, the data abstraction facility is provided by *class* in C++ and Java, but *package* in Ada.

Note that the notion of *state* came from the object-oriented paradigm, not from data abstraction. It is defined as the qualifiable condition of an object resulting from or affecting its behavior or properties [Zam99]. State is hidden together with other implementation details of an object and is accessible only through particular interface methods. In this sense, state participates in *information hiding*. The language Smalltalk supports objects with a private memory (state) and public behavior acting on that memory. The jRelix system described in this thesis provides for ADTs with state.

2.2.2 ADT in Databases

The introduction of user-defined abstract data types (ADTs) into the type system of a relational database was an evolutionary step toward moving databases beyond the traditional realm of business applications. The concept of ADTs was adapted from the programming language paradigm to databases in the 1980's in a number of projects.

ADT-INGRES [SRG83, OFS84], developed at UC Berkeley, was one of the pioneers. It allowed ADTs to be defined for domains, and allowed operations to be defined on them, including aggregates. To define a new ADT, a user would have to define its representation and write its methods in an external programming language (e.g.,C). The new type would then be registered with the database system, making the system aware of its size and available methods. Among the methods provided would be methods to input and output instances of the ADT. Once registered, the ADT could be used to define the type of an attribute of a relation, just like any built-in type. ADT methods could be used in queries and loaded as needed at run-time.

RAD [OH86], developed over the same period of time as ADT-INGRES, was an experimental database system which resembled INGRES in its approach to ADTs. Operations on new data types included primitive operations (for constant definition, comparison, inserting and updating values, and displaying a value on the screen), aggregates

(such as COUNT, SUM, etc.), and transformations (which take a relation as parameter and return a relation as result). The last type of operation, transformation, had no equivalent in ADT-INGRES. RAD also allowed variable-length string values and functions that take more than two arguments. However, functions in RAD could only return values of type BOOLEAN. With RAD, it was not possible to "undefine" data types or operations, or to define functions on built-in data types, as could be done in ADT-INGRES.

In the mid-80's, the POSTGRES project started as a follow-on to INGRES, initially to provide query optimizers with information about the properties of ADTs and their methods [Sto86a]. Another goal of POSTGRES was to provide support for storing and querying complex objects by a "procedure as a data type" approach [Sto86b]. The idea of inter-object reference (as used in object-oriented database systems) was rejected. Precomputation and query re-writing techniques used to avoid excess overheads in a procedure-centered approach.

Relational databases extended with data abstraction facilities became known as Object-Relational[5] Database Systems (ORDBMS) in the 1990's [Sto96]. Products are available from several vendors, including Informix's Universal Server, IBM's DB2 UDB, Oracle's ORACLE8, to name a few. They all support user-defined data types. Besides, ready-made ADT-based type extension packages have also become available to handle data types such as text, spatial data and image. These add-ons come under different names: "datablades" for Informix, "data extenders" for DB2, and "data cartridges" for Oracle. There are two characteristics of the current support for ADTs in ORDBMS [Ses98]:

- Each ADT is built as a module, so that it can be added or removed without affecting the rest of the system. Modularity and extensibility are essential features of an ORDBMS.
- Each ADT is like a black box. It reveals only the name and the signature of a function, often written in C, C++ or SQL.

[5]ADTs are also supported by Object-Oriented databases, but they will not be discussed here.

The foundation of many ORDBMSs is SQL3, now called SQL:1999 [GP99]. This is a third version of the SQL standard, and intended to be a major enhancement over the previous version: SQL-92. Abstract data type in SQL:1999 is termed "structured user-defined type". The most important properties of such types are:

- They can be defined on one or more attributes. Each of these attributes can be of any SQL type, or even another structured user-defined type (nesting).
- Their behavior may be specified by functions, methods, and procedures.
- Access to their attributes is only provided through system-generated "get" and "set" methods.
- Comparisons of their values are accomplished through user defined functions.
- Type inheritance is allowed.

Consider the following example of a structured type definition:

```
CREATE TYPE emp_type
  UNDER person_type
AS (EMP_ID   INTEGER
    SALARY   REAL)
INSTANTIABLE
NOT FINAL
INSTANCE METHOD
   GIVE_RAISE
   (ABS_OR_PCT   BOOLEAN,
    AMT          REAL}
   RETURNS REAL
```

The new type is a subtype of another structured type used to define a person in general (inheritance). It adds two additional attributes, employee ID and salary (states). This type is instantiable (can create multiple instances with their own states) and can have subtypes defined under it (not final). There is also a method to give an employee a raise and return the raised salary, which can be applied to instances of type "emp_type". The implementation code of the method can be written in a different language or SQL and must be registered with the system. The new ADT "emp_type" can now be used to define the type of a column in a relation. The instance method may be invoked in a query like this (assuming emp is of type "emp_type" and name is an attribute of "person_type"):

```
SELECT emp.EMP_ID, emp.GIVE_RAISE(true,3000)
FROM tblEmployee
WHERE emp.name LIKE 'John'
```

The query will acually raise the salary for employees whose name contains "John" by 3000 dollars and return a relation that contains these employees' IDs and updated salaries.

Almost all the systems mentioned above require a second language to implement the methods of an ADT. This is cumbersome and may exacerbate impedance mismatch [AH90]. JRelix takes a different approach by implementing ADTs as computations with state and their methods as nested-level computations, all in one language. Furthermore, the methds are made first class by being returned by the enclosing ADT, which, according to Atkinson and Morrison [AM84], is enough for supporting data abstraction.

2.3 ALDAT, Relix and jRelix

Aldat, standing for *ALgebraic approach to DATa*, is the name of a project at McGill University by T. H. Merrett. The Aldat project aims at unifying the fundamental concepts of database systems and general purpose programming languages. Over the years, work in this project has established generalizations to the relational algebra and created the domain algebra. Several versions of an experimental database programming language have been produced incorporating most of the concurrent research and development in databases since the 1970's. The work described in this thesis is based on an existing Aldat implementation, *jRelix*.

2.3.1 A Little History of Relix and jRelix

Relix

Relix, a **Re**lational database programming **l**anguage in Un**ix**, was developed by the Aldat laboratory, starting in 1986 [Lal86]. The data manipulation language of Relix

is Aldat, proposed by Merrett [Mer77]. Relix was intended to provide an interactive environment for exploring the concepts of the relational database model as described in [Mer84].

Relix is an interpreted language written in C. It supports relational algebra operations incuding selection, projection, μ-joins and σ-joins. μ-joins are derived from set operations such as intersection, union and difference. They are natural join, union join, symmetric difference join, left join, right join, left difference join and right difference join. σ-joins generalize set comparison operators. They include division (superset), proper superset, equal set, proper subset, subset, intersection, and the negation of these operators (e.g. the negation of intersection is not overlap).

The domain algebra proposed by Merrett [Mer76] is also implemented in Relix. It consists of two sets of operations to manipulate attributes: horizontal and vertical. Horizontal operations are mainly used for mathematics, while vertical operations provide functionalities for grouping and ordering. Horizontal operations generate new attribute values based on the other attributes in a tuple. Expressions used in a horizontal operation are built from mathematic expresions, predefined functions, conditional expressions, constants and attribute names.

Vertical domain operations include reduction, equivalence reduction, functional mapping and partial functional mapping. Reduction produces a single value from the values of an attribute across all tuples of a relation. Equivalence reduction applies reduction within groups; each group consists of tuples that have the same value for a designated set of attributes. Functional mapping processes tuples of a relation in an order determined by some control attributes. The last type of vertical operation, partial functional mapping, performs functional mapping within groups.

In addition to the extended relational algebra and the domain algebra, Relix provides facilities for updates (insertion, deletion and modification of tuples), relation and domain declarations, and assignments. Control structures for looping and recursion are also implemented. A number of commands are provided for the user to examine and manipulate relations and domains, and even to run unix shell commands. Other

language features of Relix include procedural abstraction in the form of functions and procedures, and one-level of nesting in relations [He97].

JRelix

Implemented in Java, jRelix is a second incarnation of Aldat. It inherits most of the functionalities from its predecessor Relix. The core concepts of relational algebra and domain algebra remain the same, but many new constructs have been added since 1997. Among other things, jRelix supports nested relations with arbitrary but definite depth by introducing new syntax, adopting a surrogate-based implementation, and subsuming the relational algebra into the domain algebra [Hao98, Yua98]. Computations have been re-designed and implemented as a means of procedural abstraction [Bak98]. They replace functions and procedures in Relix. More recent additions include updates for nested relations, active databases [Sun00], and attribute metadata with application to data mining [Roz02]. The following have been re-implemented for nested relations: functional mapping and partial functional mapping [Kan01], sigma-joins [Cha02], and QT-expressions [Zha02]. Besides the data abstraction facilities presented in this thesis, work is currently underway to incorporate high-precision numerical data type and networking capabilities into jRelix.

It may already have been noticed that many of the past and current issues in database research have been addressed in Relix and jRelix. According to our systematics for categorizing extensions to the relational model, Relix and jRelix :

- *extend the base type system* by introducing the domain algebra and data abstraction. It is worth noting that even without special types for things such as spatial and text data, Aldat is powerful enough to handle applications like GIS systems [Mar98] and text processors [And99].
- *provide full support for nested relations* by subsuming the relational algebra into the domain algebra and special implementation. This, coupled with support for abstract data type, ensures the capability of handling complex objects.

- *achieve more than query languages* by connecting relations to programming language constructs, such as typing (including type check and metadata), iterative abstraction (loops), parametric abstraction (computations and ADTs), event programming (active databases), and process communication (networked databases), to name a few.

2.3.2 Programming Concepts in Computations

We conclude this chapter with a discussion of the programming concepts adapted in our design and implementation of computations.

The computation is intended as a *procedural abstraction* facility in jRelix. We regard a computation as a "symmetric predicate", generalized from functions. With a computation, various subsets of its parameters can be used as inputs, and their complements as outputs. For example, to calculate the sum of two numbers, a Java function could be defined as:

```
int Sum (int a, int b)
{ return a + b;}
```

Correspondingly, we could define a jRelix computation, as follows:

```
comp Sum (a, b, c) is
{ c <- a + b;}
alt
{ a <- c - b;}
alt
{ b <- c - a;};
```

The apparently redundant code of the computation actually achieves what it would take three Java functions to do[6]:

- $Sum[1, 2]$ gives $1 + 2$,
- $Sum[, 1, 2]$ gives $2 - 1$, and
- $Sum[2, , 1]$ gives $1 - 2$

[6]We use computation name followed by parameters in square brackets as an invocation syntax in jRelix

It is therefore obvious that our computation is more flexible than functions in general programming languages.

A second programming concept involved in computations is the *referencing environment*. A referencing environment, or environment for short, of a statement, is a collection of bindings for the names that are visible in the statement. When a computation is invoked, a local environment is created to hold the bindings for the actual parameters and local variables defined in the computation. It is discarded when the computation exits. The statements in the computation may also reference variables defined outside. JRelix assumes that all such non-local variables are to be found in the environment in which the computation is declared. This is similar to the binding rules used by Pascal and Scheme.

The last programming concept we discuss here is *parameter passing*. In general, there are three parameter passing methods: pass-by-value, pass-by-reference, and pass-by-name. When a parameter is passed by value, its actual value is used to initialize the corresponding formal parameter, which then acts as a local variable in the called subprogram. As jRelix allows relation valued parameters, and relations can get arbitrarily large, pass-by-value is out of the question. The second method uses the address of the actual parameter to initialize the formal parameter. It is not chosen by jRelix either, due to the possibility of unintentional alteration of the actual parameter. JRelix uses pass-by-name, which was first introduced by Algol 60 [Seb96]. With this method, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. The main advantage of this method is flexibility, particularly in the context of computation invocation. Consider the `Sum(a,b,c)` computation given earlier in this sub-section. If we invoke it using $Sum[X, Y]$, the computation turns into:

```
comp Sum (X, Y, c) is
{ c <- X + Y;}
alt
{ X <- c - Y;}
alt
{ Y <- c - X;};
```

With a little modification to the current implementation, we can also pass compu-

tations arbitrary expressions without having to evaluate them at parameter binding time.

Chapter 3

Overview of jRelix

The purpose of this chapter is to introduce jRelix syntax and features so that the rest of the thesis is made intelligible to the reader. Section 3.1 describes how to start the jRelix system running and declare and initialize relations and domains. A number of frequently used commands are also introduced. Next we show the assignment operation. Section 3.3 discusses relational expressions. Domain algebra is the topic of Secion 3.4. Section 3.5 briefly describes updates. Finally we introduce computations in Section 3.6.

In an attempt to make this and subsequent chapters easy reading, we adhere to the following conventions:

- **Lanuage Syntax** is written in `typewriter font` and expressed in Backus-Naur form as explained in Appendix A.
- **Code Samples** are given in `small typewriter font` within boxes.

3.1 Getting Started

3.1.1 Starting the Engine

JRelix runs on any platform that has the Java Runtime Environment version 1.1 or up. A typical installation involves compiling the source files into java class files, creating a java archive file (extension .jar) from the class files, and adding the archive file to the

Java environment variable CLASSPATH. Supposing the typical installation procedure is followed, type the following to start the jRelix interpreter

```
java JRelix
```

If the environment variable CLASSPATH is not set, one needs to use the following command instead

```
java -classpath [classpath] JRelix
```

where *classpath* points to the directory where the class files or jar file are located.

Upon a successful start-up, a greeting screen appears and jRelix is ready to accept commands at the prompt '>'.

3.1.2 Commonly Used Commands

The following is a summary of the most frequently used commands in jRelix. Appendix A contains a complete listing of all commands.

quit Quit the interpreter. Information on the database is saved upon exit.

pr EXPR Display the result of a relational expression evaluation.

sd (ID)? Show a description of the attribute ID. If ID is omitted, all attributes are shown.

sr (ID)? Show a description of the relation ID. Show all relations if ID is omitted.

dd *IDList* Delete attributes specified in *IDList*.

dr *IDList* Delete relations, views or computations specified in *IDList*.

debug Toggle the debug mode. Used to display the syntax tree of each statement or the hidden states of a relation.

3.1.3 Domain and Relation Declaration

Domain Declaration

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

```
"domain" IDList Type ";"
```

where `IDList` contains a list of domains being declared, and `Type` denotes the type of these domains. Table 3.1 summarizes the valid domain types in the current jRelix system. They come in two categories: atomic and complex. The atomic types are primitive types such as `double`, `integer`, `string`, etc. Nested relations (`IDList`) and computations are examples of complex types. Types `attribute` and `universal` are added to implement attribute meta-data [Roz02] which is out of the scope of this thesis.

Type	Alias	Category	Corresponding Java Type
`boolean`	`bool`	atomic	`true, false`
`short`		atomic	signed short int, 2 bytes
`integer`	`intg`	atomic	signed int, 4 bytes
`long`		atomic	signed long, 8 bytes
`float`	`real`	atomic	signed float, 4 bytes
`double`		atomic	signed double, 8 bytes
`string`	`strg`	atomic	String
`attribute`	`attr`	atomic	String
`universal`	`univ`	atomic	String
`"("`*IDList*`")"`		complex	
`computation "("`*IDList*`")"`	`comp`	complex	

Table 3.1: Domain Types in jRelix

Information (name, type, etc.) about a domain can be shown with the `sd` command. A domain may be deleted using the `dd` command followed by its name. However, any existing relation is defined on this domain, the deletion will fail with a warning.

Relation Declaration and Initialization

The following syntax is used to declare and initialize a relation:

```
"relation" IDList "("IDList")" (Initialization)?  ";"
```

The first `IDList` specifies a list of relations being declared. The second contains the domains on which these relations are defined. `Initialization`, when present, consists of a number of comma delimited, parenthesized tuples enclosed in a pair of curly braces.

Three commands commonly used with relations are: `sr`, `pr`, and `dr`. See Section 3.1.2 for details.

Examples

Example 1. *Various domain declarations*

Figure 3.1 shows the declaration of four domains. The first is an `integer`, and the second a nested relation defined on the first. The third is another non-primitive typed domain, which contains a nested relation as its own domain. Finally we give a computation typed domain whose parameter is of primitive type.

```
>domain a intg;
>domain A(a);
>domain B(a, A);
>domain C comp(a);
```

Figure 3.1: Example: Declaration of Domains

Example 2. *Relation declarations*

In the example presented in Figure 3.2, a nested relation `Company` is defined on two domains, `dept` and `employee`. Domain `dept` is of primitive type `string`, while `employee` is a relation itself, defined on `name` and `sal`. The relation is shown in Table 3.2.

```
>domain name, dept strg;
>domain sal float;
>domain employee(name, sal);
>relation Company(dept, employee) <-
   {("Sales", {("John Manley", 50000),
               ("Allen Smith", 45000)}),
    ("HR",    {("Jay Ashman", 44000),
               ("Tim Gordon", 33000)}),
    ("R&D",   {("Albert Einstein", 80000)})};
```

Figure 3.2: Example: Declaration and Initialization of Relations

dept	employee (name	sal)
Sales	John Manley	50000
	Allen Smith	45000
HR	Jay Ashman	44000
	Tim Gordon	33000
R&D	Albert Einstein	80000

Table 3.2: The `Company` Relation

Note if we issue the command

```
pr Company;
```

The relation will be displayed as in Figure 3.3. This is because for a nested relation, surrogates are used to replace the actual values of relation typed domains. The actual relation for such a domain is stored in a separate relation whose name is the name of the domain prefixed with a dot ('.'). An informal name for this associated relation is "dot relation". The dot relation is defined on all the attributes that the relation typed domain is defined on, plus an additional domain, ".id". The values of ".id" link to the surrogates in the parent relation.

```
>pr Company;
+----------------------+----------------------+
| dept                 | employee             |
+----------------------+----------------------+
| HR                   | 2                    |
| R&D                  | 3                    |
| Sales                | 1                    |
+----------------------+----------------------+
relation Company has 3 tuples
>pr .employee;
+-------+----------------------+---------------+
| .id   | name                 | sal           |
+-------+----------------------+---------------+
| 1     | Allen Smith          | 45000.0       |
| 1     | John Manley          | 50000.0       |
| 2     | Jay Ashman           | 44000.0       |
| 2     | Tim Gordon           | 33000.0       |
| 3     | Albert Einstein      | 80000.0       |
+-------+----------------------+---------------+
relation .employee has 5 tuples
```

Figure 3.3: Representation of Nested Relations

3.2 Assignments

jRelix provides two assignment operators, one is *assignment* (<-) and the other is *incremental assignment* (<+). The *assignment* operator creates a relation with the name specified to the left of the operator and with the same domains as the source relation. The data of the source relation is copied to the destination relation. If a relation with the same name as the destination relation already exists, it is first removed. The *incremental assignment* operator behaves in exactly the same way if there does not already

exist a relation with the name of the left operand. If such a relation exists, the destination relation becomes the union of the source and the destination relations provided that both relations are defined on the same set of domains. In any case, the relation on the right-hand side of the operator is not affected. The syntax for the assignment operators is shown below. The second form is used to rename attributes in `ExpressionList` to the ones specified in `IDList`. Examples of use are provided in Figure 3.4.

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

```
>domain Title strg;
>domain Price, Cost float;
>relation myBook(Title,Price) <-
 {("Java 2", 68), ("XML Black Book", 60)};
>yourBook [Title,Cost <- Title,Price] myBook;
>pr yourBook;
+----------------+---------------+
| Title          | Cost          |
+----------------+---------------+
| Java 2         | 68.0          |
| XML Black Book | 60.0          |
+----------------+---------------+
relation yourBook has 2 tuples
>relation newBooks(Title, Cost) <-
 {("SQL in 21 Days", 25)};
>yourBook <+ newBooks;
>pr yourBook;
+----------------+---------------+
| Title          | Cost          |
+----------------+---------------+
| Java 2         | 68.0          |
| SQL in 21 Days | 25.0          |
| XML Black Book | 60.0          |
+----------------+---------------+
relation yourBook has 3 tuples
```

Figure 3.4: Example: Assignments

3.3 Relational Algebra

The relational algebra consists of a set of functional operators which act on either one or two relations and produce a relation in result. This closure property of relational algebra allows complex expressions to be constructed by chaining relational operators.

3.3.1 Unary Operators

jRelix supports six unary operators. These are *projection*, *selection*, *T-selection*, *QT-selection*, *pick*, and *AttribsOf*. In all cases, the source relation is not affected by the operator.

Projection

Projection extracts a subset of the source relation consisting of the domains named in `IDList`. Duplicates are removed. If `IDList` is omitted,, the result relation contains just one tuple with a boolean domain "`.bool`". The value of the tuple is `true` if and only if the source relation has at least one tuple. The result of evaluating `Expression` provides the source relation. The syntax for projection is:

```
"[" (IDList)?  "]" "in" Expression
```

Example 1. *Retrieve the Titles of relation* **yourBook** (see Figure 3.5).

```
>yourTitles <- [Title] in yourBook;
>pr yourTitles;
+----------------------+
| Title                |
+----------------------+
| Java 2               |
| SQL in 21 Days       |
| XML Black Book       |
+----------------------+
relation yourTitles has 3 tuples
```

Figure 3.5: Projection: Example 1

Example 2. *Check whether relation* **yourBook** *is empty* (see Figure 3.6).

```
>pr ([] in yourBook);
+---------+
| .bool   |
+---------+
| true    |
+---------+
expression has 1 tuple
```

Figure 3.6: Projection: Example 2

Selection

Selection returns a subset of the source relation which satisfy the conditions specified in `SelectClause`. As with projection, the source relation may be the result of evaluating any arbitrary relational `Expression`.

```
"where" SelectClause "in" Expression
```

Example 1. *Retrieve the tuples of relation* **yourBook** *in which the Cost is higher than 50 dollars* (see Figure 3.7).

```
>expensiveBook <- where Cost > 50.0 in yourBook;
>pr expensiveBook;
+----------------------+---------------+
| Title                | Cost          |
+----------------------+---------------+
| Java 2               | 68.0          |
| XML Black Book       | 60.0          |
+----------------------+---------------+
relation expensiveBook has 2 tuples
```

Figure 3.7: Selection: Example 1

T-Selection

Projection and selection can be combined into one expression called T-selection. The syntax is:

```
"[" (IDList)?  "]" "where" SelectClause "in" Expression
```

Example 1. *Retrieve the Titles in relation* **yourBook** *for which the Cost is higher than 50 dollars* (see Figure 3.8).

```
>costlyTitles <- [Title] where Cost > 50.0 in yourBook;
>pr costlyTitles;
+----------------------+
| Title                |
+----------------------+
| Java 2               |
| XML Black Book       |
+----------------------+
relation costlyTitles has 2 tuples
```

Figure 3.8: T-Selection: Example 1

The array syntax is the syntactic sugar for a special case of the T-selection — when the select clause consists of a conjunction of comparisons for equality and the projector list contains all domains not mentioned in the select clause. The syntax is:

```
Identifier "[" (Identifier)?  (, (Identifier)?)* "]"
```

For example, the following three statements are equivalent.

```
costlyTitles <- [Cost] where Title = "Java 2" in yourBook;
costlyTitles <- yourBook["Java 2"];
costlyTitles <- yourBook["Java 2",];
```

Normally there need to be $n-1$ commas in the square brackets after the relation name, if the relation has n domains. The first comma comes after the value of the first attribute, the second comma after the second attribute, and so forth. If the attribute to appear in the projector list happens to be the last in the source relation, the comma before it can be omitted.

The array syntax proves to be useful in the implementation of abstract data types.

QT-Selection

QT-Selectors are extensions to T-Selectors by adding quantifiers such as QT-Count (#), QT-Proportion (.), and QT-Percentage (%). These remove from T-selection the restriction that the select clause must evaluate to true or false on *each* tuple of the relation. The syntax for QT-selection is:

```
("["(ExprList)?"]")?  "quant" QTList ("where" Condition)?  "in"
                                Expression
```

where `ExprList` is a list of domains on which to define the result relation. If nothing precedes **quant**, the QT-Selection results in a relation defined on all the domains of the source relation. `QTList` is a list of conditions which the resulting tuples must satisfy. The interested reader can find examples of QT-Selection in [Zha02].

Pick

The pick operator randomly selects a tuple from a source relation and assigns its value to the destination relation. The destination relation is thus guaranteed to be singleton. The syntax of pick operation is:

```
"pick" Expression
```

where `Expression` evaluates to the source relation.

Example 1. (see Figure 3.9).

```
>RandomR <- pick SUPPLY;
>pr RandomR;
+--------+---------+-------+-------+
| ITEM   | COMP    | DEPT  | VOL   |
+--------+---------+-------+-------+
| String | Playsew | Toy   | 10    |
+--------+---------+-------+-------+
relation RandomR has 1 tuple
```

Figure 3.9: Pick: Example 1

AttribsOf

This operator creates a relation of all the domains of the operand. The syntax of the AttribsOf operation is as follows:

```
"AttribsOf" Expression
```

where `Expression` evaluates to a relation. The output relation is defined on a single domain of type `attribute` and its values are the names of all the domains of the source relation. Details of this operation can be found in [Roz02].

3.3.2 Binary Operators

jRelix supports 19 binary relational operators. They come in two categories: μ-joins and σ-joins. μ-joins are set operations generalized for relations, and σ-joins are generalization of logical operations [Mer84]. These operators satisfy the algebraic closure property. The syntax for the join operations are as follows:

```
Expression JoinOperator Expression
                    |
Expression "["ExprList":"JoinOperator":"ExprList"]" Expression
```

In the first production, the two operands join on their common domains. When there is no common domain, the second production can be used to explicitly name the join domains; the domains in the first `ExprList` match those in the second on a by position basis. `JoinOperator` may be any one of the 19 operators discussed shortly.

μ-joins

This category of join operators extend the mathematical set operations including union, intersection and difference. With the exception of the difference joins, the result of their application is a relation which has as its domains the union of the domains from the two input relations. μ-joins are summarized in Table 3.3.

The μ-joins can be defined in terms of three parts: the left wing, the center wing, and the right wing. The definitions of the three wings are:

- For relations $R(X, Y)$ and $S(Y, Z)$ sharing a common attribute set, Y

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

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

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

- For relations $R(W, X)$ and $S(Y, Z)$ sharing no common attribute set

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

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

  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 following examples of μ-joins are based on the two relations shown in Figure 3.10.

Example 1. *Find the items each agent is responsible for and the location of the agent* (see Figure 3.11).

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

Table 3.3: Summary of μ-joins

```
>pr Responsibility;
+----------------------+----------------------+
| Agent                | Item                 |
+----------------------+----------------------+
| Hung                 | Micro                |
| Raman                | Micro                |
| Raman                | Terminal             |
| Smith                | VCR                  |
+----------------------+----------------------+
relation Responsibility has 4 tuples
>pr Location;
+----------------------+-------------+
| Item                 | Floor       |
+----------------------+-------------+
| Micro                | 1           |
| Terminal             | 1           |
| Terminal             | 2           |
| Videodisk            | 2           |
+----------------------+-------------+
relation Location has 4 tuples
```

Figure 3.10: Relations Used in μ-join Examples

```
>AgentInfo <- Responsibility ijoin Location;
>pr AgentInfo;
+----------------------+----------------------+-------------+
| Item                 | Agent                | Floor       |
+----------------------+----------------------+-------------+
| Micro                | Hung                 | 1           |
| Micro                | Raman                | 1           |
| Terminal             | Raman                | 1           |
| Terminal             | Raman                | 2           |
+----------------------+----------------------+-------------+
relation AgentInfo has 4 tuples
```

Figure 3.11: μ-join: Example 1

Example 2. *Find all items that don't have an agent* (see Figure 3.12).

```
>NoAgent <- Responsibility drjoin Location;
>pr NoAgent;
+----------------------+-------------+
| Item                 | Floor       |
+----------------------+-------------+
| Videodisk            | 2           |
+----------------------+-------------+
relation NoAgent has 1 tuple
```

Figure 3.12: μ-join: Example 2

σ-joins

The σ-joins extend the truth-valued comparison operations on sets to relations by applying them to each set of values of the join domain for each of the other values in the two operand relations. The result of their application is a relation whose domains are the symmetric difference of the two sets of domains of the operands. The `JoinOperator` in the production shown earlier can also be any of the *sigma*-joins given in Table 3.4[1].

sigma-join	Operator	Description	Set Operator
natural composition	`icomp` or `natcomp`	overlap	$\not\emptyset$
equal join	`eqjoin`	equal	$=$
greater than or equal join	`gejoin` or `sup` or `div`	superset	$\supseteq$
greater than join	`gtjoin`	proper superset	$\supset$
less than or equal join	`lejoin` or `sub`	subset	$\subseteq$
less than join	`ltjoin`	proper subset	$\subset$
empty intersection join	`iejoin` or `sep`	not overlap	$\not\emptyset$

Table 3.4: Summary of σ-joins

We can define the σ-joins using the following notation. In relations $R(W, X)$ and

[1]The table lists only 7 of the 12 σ-joins. The five that are not shown correspond to the logical negation of the entries except `icomp` and `sep`. They can be formed by prefixing a "!" or "not" to the operators of their negation. For example, "`!eqjoin`" or "`not eqjoin`" is the negation of "`eqjoin`". `sep` is the negation of `icomp`. The tuples from one operator complement those of its negation.

$S(Y, Z)$, 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, but 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 **sup** $S \equiv \{(w, z) | R_w \supseteq S_z\}$
- R **gtjoin** $S \equiv \{(w, z) | R_w \supset S_z\}$
- R **lejoin** $S \equiv \{(w, z) | R_w \subseteq S_z\}$
- R **ltjoin** $S \equiv \{(w, z) | R_w \subset S_z\}$
- R **eqjoin** $S \equiv \{(w, z) | R_w = S_z\}$

The negations of these are defined accordingly.

The following examples use the relations defined in Figure 3.10 in the previous subsection.

Example 1. *Find the agents for all items on a floor* (see Figure 3.13).

```
>VersatileAgents <- Responsibility sup Location;
>pr VersatileAgents;
+----------------------+-------------+
| Agent                | Floor       |
+----------------------+-------------+
| Raman                | 1           |
+----------------------+-------------+
relation VersatileAgents has 1 tuple
```

Figure 3.13: σ-join: Example 1

Nop

When applied to two operand relations, the nop operation assigns one of them to the result relation provided the two are defined on the same set of attributes. For example, in the following example

```
R1 <- A nop B;
```

Relation `R1` has a 50% chance of being a copy of `A` and a 50% chance of being a copy of `B`. Consider the next example

```
R2 <- A nop B nop C;
```

This time, `R2` still has a 50% chance to be a copy of `C`. But its odds of becoming `A` or `B` are just 25% each. That is, the nop operator is left associative.

In Section 3.4.2, we will see how the `nop` operator can be used in combination with reduction to implement level lifting of a special kind of nested relation.

3.4 Domain Algebra

The *domain algebra* [Mer77, Mer84] is an algebra on attributes and it fills the gap in the *relational algebra* with its ability to do, among other things, arithmetic. The two main components to the domain algebra are: scalar operations and aggregate operations. In view of the table representation of a relation, these can be thought of as "horizontal" and "vertical", respectively. Horizontal domain operations work within the tuples and vertical operations work across the tuples. A taxonomy of the domain algebra was given in Section 2.1.1.

The domain algebra is used by associating an expression with a virtual domain at the declaration time of the latter. This expression can be anything from a constant value to a relational expression that instructs the system on how to build the value of the virtual domain when needed. The domain expression can be defined in terms of the actual domains or other virtual domains. Recursive definition is not permitted. A virtual domain may appear anywhere an actual domain is expected. It is actualized only when referred to through the relational algebra. The syntax for virtual domain declaration is:

```
"let" Identifier "be" Expression ";"
```

The following sub-sections briefly discuss the two types of operations with examples. An in-depth coverage of this topic can be found in [Yua98].

3.4.1 Horizontal Operations

A virtual domain can be defined as a constant or as an alias of another domain, as shown in Example 1.

Example 1. *Constant and renaming* (see Figure 3.14).

```
>let ONE be 1;
>let FALSE be false;
>let UNE be ONE;
```

Figure 3.14: Constant Virtual Domain and Renaming

The most frequently used unary operators in the domain algebra are "−" for numericals, "not" for truth values, and the unary relational operators for domains of type `IDList`. See Example 2 below.

Example 2. *Unary operations* (see Figure 3.15).

```
>let AGE be -Age;
>let TRUE be not FALSE;
```

Figure 3.15: Virtual Domain with Unary Operation

Examples 3 shows the declaration of virtual domains defined on binary operations. For domains of atomic types (see Table 3.1), binary operators normally include binary arithmetic operators and logical operators. For those of complex types, this usually means the relational operators such as `ijoin`, `ujoin`, etc.

Example 3. *Binary operations* (see Figure 3.16).

```
>let TotalCost be UnitCost * Quantity;
>let AllEmployee be Staff ujoin Professor;
```

Figure 3.16: Virtual Domain with Binary Operation

Declaring a virtual domain based on conditions is also possible. Example 4 gives a definition of *absolute value* by means of the domain algebra. However, jRelix has its own built-in function to handle this. This is shown in Example 5.

Example 4. *Conditional expression* (see Figure 3.17).

Example 5. *Built-in functions* (see Figure 3.18).

```
>let ABSX be if X >= 0 then X else -X;
```

Figure 3.17: Virtual Domain with a Conditional Expression

```
>let ABSX be abs(X);
```

Figure 3.18: Virtual Domain with Built-in Functions

3.4.2 Vertical Operations

The vertical operations work across tuples, which permits, for example, summing all the values of a domain. The two main classes of vertical operations are *reduction* and *functional mapping*. What distinguishes the two main classes is **order**. Within each class, there are two subclasses: one with a grouping facility and the other without. Thus a total of four vertical operations are available in jRelix. The examples given next are all based on the relation given in Figure 3.19, except when noted otherwise.

```
>domain Dept, Name, Gender strg;
>domain Salary float;
>relation Company(Dept,Name,Gender,Salary) <-
 {("Accounting","J. White","F",45000),
  ("Accounting","M. Scholl","M",45500),
  ("Accounting","K. Holmes", "M",39000),
  ("Sales","J. Cioffy","M",60000),
  ("Sales","E. Malon","F",50000),
  ("HR","D. Johns","F",48000),
  ("HR","N. Lovejoy","F",50000)};
>pr Company;
+------------+-------------+---------+----------+
| Dept       | Name        | Gender  | Salary   |
+------------+-------------+---------+----------+
| Accounting | J. White    | F       | 45000.0  |
| Accounting | K. Holmes   | M       | 39000.0  |
| Accounting | M. Scholl   | M       | 45500.0  |
| HR         | D. Johns    | F       | 48000.0  |
| HR         | N. Lovejoy  | F       | 50000.0  |
| Sales      | E. Malon    | F       | 50000.0  |
| Sales      | J. Cioffy   | M       | 60000.0  |
+------------+-------------+---------+----------+
relation Company has 7 tuples
```

Figure 3.19: Relation `Company` Used in the Examples for Vertical Operations

Reduction

The following statement creates a sum of all salaries by using the reduction operation:

```
let TotSalary be red + of Salary;
```

If we replace the "+" with "min", we get the minimum salary. An aggregate virtual domain (such as `TotSalary`) obtained by reduction is a constant in that it will have the same value for all tuples of the relation in which it is actualized. Consider the example in Figure 3.20. Note that we can combine two aggregate virtual attributes by means of a horizontal operator, *division*. Indeed, complex domain expressions can be built due to the fact that domains are closed under the domain algebra.

Example 1. *Reduction operation* (see Figure 3.20).

```
>let TotSalary be red + of Salary;
>let TotHeadCnt be red + of 1;
>let AvgSalary be TotSalary / TotHeadCnt;
>CompSalaryStats <- [TotSalary,TotHeadCnt,AvgSalary] in Company;
>pr CompSalaryStats;
+---------------+-------------+---------------+
| TotSalary     | TotHeadCnt  | AvgSalary     |
+---------------+-------------+---------------+
| 337500.0      | 7           | 48214.285     |
+---------------+-------------+---------------+
relation CompSalaryStats has 1 tuple
```

Figure 3.20: Example: Reduction Operation

There are ten built-in operators that may follow the **red** keyword in a reduction or partial reduction operation: $+$, $*$, **min**, **max**, **and**, **or**, **nop**, **ijoin**, **ujoin**, **sjoin**. The first six are for primitive typed domains. The last three are for relation typed domains. The operator **nop** works with both types. To qualify in reduction operations, an operator must be associative and commutative. This requirement stems from the orderlessness of tuples in a relation.

A new mechanism has been introduced to allow user-defined computations to be used in vertical operations. One simply needs to place a computation call after the **red** keyword. Details on this feature will be given in Chapter 4.

Reduction and Level Lifting

If we define a relation typed domain `Employee` and include `Name`,`Gender` and `Salary` as its attributes, the `Company` relation becomes the nested relation, `NewCompany`, as shown in Figure 3.21. Notice now there are only three tuples, with one tuple each for the three departments. The staff in each department is represented by an `Employee` sub-relation.

Dept	Employee		
	Name	Gender	Salary
Accounting	J. White	F	45000.0
	K. Holmes	M	39000.0
	M. Scholl	M	45500.0
HR	D. Johns	F	48000.0
	N. Lovejoy	F	50000.0
Sales	E. Malon	F	50000.0
	J. Cioffy	M	60000.0

Figure 3.21: Relation `NewCompany`

To find out about *all* the employees in the company, we could use the code in Figure 3.22. However, since `allEmployees` is itself a relation nested inside the result, we have to refer to the associated dot relation for final answers. This inconvenience can be eliminated by means of "level lifting".

```
>let allEmployees be red ujoin of Employee;
>Result <- [allEmployees] in NewCompany;
>pr Result;
+----------------------+
| allEmployees         |
+----------------------+
| 12                   |
+----------------------+
relation Result has 1 tuple
>pr .allEmployees;
+-----+----------------------+----------------------+---------------+
| .id | Name                 | Gender               | Salary        |
+-----+----------------------+----------------------+---------------+
| 12  | D. Johns             | F                    | 48000.0       |
| 12  | E. Malon             | F                    | 50000.0       |
| 12  | J. Cioffy            | M                    | 60000.0       |
| 12  | J. White             | F                    | 45000.0       |
| 12  | K. Holmes            | M                    | 39000.0       |
| 12  | M. Scholl            | M                    | 45500.0       |
| 12  | N. Lovejoy           | F                    | 50000.0       |
+-----+----------------------+----------------------+---------------+
relation .allEmployees has 7 tuples
```

Figure 3.22: All Employees: Version 1

Level lifting is a jRelix technique to bring subrelations/attribute values one level up the nesting hierarchy via anonymity. As illustrated in Figure 3.23, a `red ujoin` expression replaces the `allEmployees` virtual domain in the previous example. Because no name has been given to the attribute of the result relation, the attributes of `Employee` are used directly, thus achieving level lifting. It is important to note that level lifting

only applies to a subrelation/attribute that has the same value for all tuples (singleton condition). In the example, `allEmployees` satisfies the singleton condition due to the use of a reduction operation. In general, reduction combined with anonymity provides a useful implementation of level lifting in certain nested relations. We have seen the use of "red ujoin"to lift the level of a relation typed domain. Similarly "red max" or "red min" can be used for level lifting on a numeric domain. A more general technique that applies to any type of domain is through the combination of reduction and the `nop` operator. This technique is especially useful when the intention is to lift the level of one particular instance of a subrelation or one particular value of an attribute, with the instance or value picked indeterministically. When a relation contains only one tuple, "red nop" has the same effect as other level lifting means. Chapter 4 contains many examples of level lifting using "red nop".

```
>Result <- [red ujoin of Employee] in NewCompany;
>pr Result;
+----------------------+----------------------+---------------+
| Name                 | Gender               | Salary        |
+----------------------+----------------------+---------------+
| D. Johns             | F                    | 48000.0       |
| E. Malon             | F                    | 50000.0       |
| J. Cioffy            | M                    | 60000.0       |
| J. White             | F                    | 45000.0       |
| K. Holmes            | M                    | 39000.0       |
| M. Scholl            | M                    | 45500.0       |
| N. Lovejoy           | F                    | 50000.0       |
+----------------------+----------------------+---------------+
relation Result has 7 tuples
```

Figure 3.23: All Employees: Version 2

Equivalence Reduction

`Equivalence reduction` allows reduction to be applied to groups of tuples within a relation. These groups are *equivalent* based on having the same value for a specified set of domains. For example, we can continue from the previous example and ask for the average salary in each department.

Example 2. *Equivalence reduction operation* (see Figure 3.24).

The ten operators for reduction also apply to equivalence reduction. User-defined computations are applicable as well.

```
>let DeptTotSal be equiv + of Salary by Dept;
>let DeptHeadCnt be equiv + of 1 by Dept;
>let DeptAvg be DeptTotSal / DeptHeadCnt;
>DeptSalStats <- [Dept, DeptAvg] in Company;
>pr DeptSalStats;
+----------------------+---------------+
| Dept                 | DeptAvg       |
+----------------------+---------------+
| Accounting           | 43166.668     |
| HR                   | 49000.0       |
| Sales                | 55000.0       |
+----------------------+---------------+
relation DeptSalStats has 3 tuples
```

Figure 3.24: Example: Equivalence Reduction Operation

Functional Mapping

Reductions are not sufficient in occasions such as calculating the cumulative sum. To introduce order into vertical operations, we need *functional mapping*. Consider Example 3 in which the salaries are ordered and each employee is given a pay rank. We would like to have the highest paid employee to be ranked number one, therefore the `order` clause uses the negation of Salary.

Example 3. *Functional mapping operation* (see Figure 3.25).

```
>let PayRank be fun + of 1 order -Salary;
>CompRanks <- [Dept,Name,Salary,PayRank] in Company;
>pr CompRanks;
+--------------+-------------+----------+---------+
| Dept         | Name        | Salary   | PayRank |
+--------------+-------------+----------+---------+
| Accounting   | J. White    | 45000.0  | 5       |
| Accounting   | K. Holmes   | 39000.0  | 6       |
| Accounting   | M. Scholl   | 45500.0  | 4       |
| HR           | D. Johns    | 48000.0  | 3       |
| HR           | N. Lovejoy  | 50000.0  | 2       |
| Sales        | E. Malon    | 50000.0  | 2       |
| Sales        | J. Cioffy   | 60000.0  | 1       |
+--------------+-------------+----------+---------+
relation CompRanks has 7 tuples
```

Figure 3.25: Example: Functional Mapping Operation

In addition to the associative and commutative operators discussed in the previous sub-sections, funtional mapping allows the following as well: **cat** (string concatenation), $-$, /, **mod** (modulo), $**$ (power), **pred** (cyclic predecessor), **succ** (cyclic successor). The last two operators are yet to be implemented in jRelix.

Partial Functional Mapping

Partial functional mapping adds a grouping facility to functional mapping in the same way that equivalence reduction does for reduction. Example 4 shows the use of partial functional mapping to calculate the pay rank within each department.

Example 4. *Partial functional mapping operation* (see Figure 3.26).

```
>let DeptRank be par + of 1 order -Salary by Dept;
>DeptRanks <- [Dept,Name,Salary,DeptRank] in Company;
>pr DeptRanks;
+----------------------+----------------------+---------------+-------------+
| Dept                 | Name                 | Salary        | DeptRank    |
+----------------------+----------------------+---------------+-------------+
| Accounting           | J. White             | 45000.0       | 2           |
| Accounting           | K. Holmes            | 39000.0       | 3           |
| Accounting           | M. Scholl            | 45500.0       | 1           |
| HR                   | D. Johns             | 48000.0       | 2           |
| HR                   | N. Lovejoy           | 50000.0       | 1           |
| Sales                | E. Malon             | 50000.0       | 2           |
| Sales                | J. Cioffy            | 60000.0       | 1           |
+----------------------+----------------------+---------------+-------------+
relation DeptRanks has 7 tuples
```

Figure 3.26: Example: Partial Functional Mapping Operation

3.5 Update

Update provides the mechanism for changing a relation. There are three basic update operations on relation: *add*, *delete* and *change*. The syntax for update is:

```
        "update" Identifier ("add"|"delete") Expression ";"
                                    |
"update" Identifier "change" StatementList (UsingClause)?  ";"
```

where `UsingClause` is defined as:

```
                "using" JoinOperator Expression
                                    |
"using" "["IDList":"JoinOperator(":")?ExpressionList"]" Expression
```

The semantics of `add` is the same as that of the `incremental assignment`. The semantics of `delete` is the same as that of the `djoin`. A relation can also be updated in part by the `change` operation. Changes to perform are specified in `StatementList`. Assignment is the most common type of change statement. In later chapters, we will see

examples of issuing computation calls in an update statement. The part of the relation to be updated is the result of join (specified by `JoinOperator`) between the relation being updated and the relation resulting from `UsingClause`. If the `UsingClause` is absent, the whole relation is updated. The default `JoinOperator` is ijoin.

Example 1. *Updating flat relations* (see Figure 3.27).

```
>domain Item, Type strg;
>relation Class(Item,Type) <-  {("Yarn", "a"),
                                ("String", "a"),
                                ("Ball", "b"),
                                ("Sandal", "c")};
>relation ReClass(Item,Type) <- {("Yarn", "a"),
                                 ("String", "b"),
                                 ("Top", "a")};
>update Class add ReClass;
>pr Class;
+---------------------+---------------------+
| Item                | Type                |
+---------------------+---------------------+
| Ball                | b                   |
| Sandal              | c                   |
| String              | a                   |
| String              | b                   |
| Top                 | a                   |
| Yarn                | a                   |
+---------------------+---------------------+
relation Class has 6 tuples

>update Class change Type <- "B"
 using ijoin ReClass;
>pr Class;
+---------------------+---------------------+
| Item                | Type                |
+---------------------+---------------------+
| Ball                | b                   |
| Sandal              | c                   |
| String              | B                   |
| String              | a                   |
| Top                 | B                   |
| Yarn                | B                   |
+---------------------+---------------------+
relation Class has 6 tuples
```

Figure 3.27: Example: Updating Flat Relations

Example 2. *Updating nested relations* (see Figure 3.28 and Figure 3.29).

3.6 Computation

Computations [Mer93, Bak98] are the procedural abstraction mechanism provided by jRelix. They can be regarded as potentially infinite relations containing tuples which

```
>domain Major, Name strg;
>domain Mark intg;
>domain Student(Name, Mark);
>relation Records(Major,Student) <- {("CS", {("J. Doe", 80),
                                             ("G. Ford", 56),
                                             ("H. Canning", 88)}),
                                    ("EE", {("B. Martin", 99),
                                             ("P. Fisher", 45)})};
>relation NewCSRec(Name, Mark) <- {("A. Wood", 77)};

<<add the new student to CS major
>update Records change
  (update Student add NewCSRec)
 using ijoin where Major = "CS" in Records;
>pr Records;
+---------------------+---------------------+
| Major               | Student             |
+---------------------+---------------------+
| CS                  | 1                   |
| EE                  | 2                   |
+---------------------+---------------------+
relation Records has 2 tuples
>pr .Student;
+-------------+---------------------+-------------+
| .id         | Name                | Mark        |
+-------------+---------------------+-------------+
| 1           | A. Wood             | 77          |
| 1           | G. Ford             | 56          |
| 1           | H. Canning          | 88          |
| 1           | J. Doe              | 80          |
| 2           | B. Martin           | 99          |
| 2           | P. Fisher           | 45          |
+-------------+---------------------+-------------+
relation .Student has 6 tuples
```

Figure 3.28: Example: Updating Nested Relations, part 1

```
<<reduce mark by 5 for EE students
>update Records change
   (update Student change Mark <- Mark - 5)
 using ijoin where Major = "EE" in Records;
>pr Records;
+---------------------+---------------------+
| Major               | Student             |
+---------------------+---------------------+
| CS                  | 1                   |
| EE                  | 2                   |
+---------------------+---------------------+
relation Records has 2 tuples
>pr .Student;
+-------------+---------------------+-------------+
| .id         | Name                | Mark        |
+-------------+---------------------+-------------+
| 1           | A. Wood             | 72          |
| 1           | G. Ford             | 51          |
| 1           | H. Canning          | 83          |
| 1           | J. Doe              | 75          |
| 2           | B. Martin           | 94          |
| 2           | P. Fisher           | 40          |
+-------------+---------------------+-------------+
relation .Student has 6 tuples
```

Figure 3.29: Example: Updating Nested Relations, part 2

satisfy the semantic constraints embodied by the code in the computations. Computation domains are the elements exported through the parameter list. A parameter can serve either as input or output, depending on how the computation is called. A computation can have more than one alternative blocks, each representing one of the views of the same constraint. However, it is the programmer's responsibility to ensure the consistency among the blocks. Computations employ the pass-by-name parameter passing mechanism [Seb96], which results from the design choice of modelling them after relations.

Advanced uses of computations include stateful computations, packages, abstract data types, constraint verification, and recursive computations. The first three have been made available by this implementation[2]. Examples of using packages and stateful computations will be given shortly. Abstract data types are treated thoroughly in Chapters 4 and 5. [Bak98] shows examples of constraint verification and recursive computation in Chapter 3.

At the end of this section, we present some commands commonly used with computations. For details please refer to [Bak98].

3.6.1 Defining and Invoking a Computation

The syntax for declaring a computation can be found in Appendix A. It is possible to declare computations at two levels: *top level* or *nested level*. Top level computations are not declared inside the code block of any other computations whereas nested ones are. Top level computations can be invoked from anywhere in the code after its definition. On the other hand, nested level computations are only available in two places:

- **In the computation code block** where the nested level computation is defined, *or*
- **In a relation resulting from instantiating an Abstract Data Type** which exports the nested level computation as an ADT method.

[2]Stateful computations were mentioned in [Bak98]; however, a working model was not available then.

In addition, a nested level computation becomes top-level if it is exported from a package. We'll see an example in Section 3.6.3.

Apart from this distinction in scope, the calling syntax used with the two types of computations differs slightly. There are four forms of invocation syntax for computations in jRelix:

1. **Stand-alone invocation**: invoking a computation by means of a top-level call statement, with specified input and output arguments.
2. **Select/array syntax**: invoking a computation by means of a select expression. The select predicate provides the values of the input parameters, whereas the result relation contains the outputs of a computation as its attributes values.An array syntax may be used instead as a syntactic sugar.
3. **Natural join syntax**: joining a computation with a relation. Each tuple of the relation provides the input value for the computation and the result relation will contain both the input and the output values for all tuples that satisfy the constraint represented by the computation.In the implementation given by [Bak98], the name of the formal parameter must match that of the actual parameter. This restriction has been eliminated. More on this in 5.2.1.
4. **Vertical syntax**: using a user-defined computation to systematically process the values of a domain in a relation.

The last syntax will be discussed in Chapters 4 and 6. Top level computations may be invoked using any syntax form. If a nested level computation is invoked from withing the code block in which it is defined, all forms of syntax apply. Computations intended as accessors[3] of an ADT normally use syntax 2, while those intended as modifiers[4] use syntax 1 within an update statement.

Examples showing the use of computations in ADTs will be given in Chapter 4. The following examples illustrate the use of stateless computations.

[3]Accesor methods are functional.

[4]Modifier methods change the state.

Example 1. *Computation declaration*

Figure 3.30 shows the declaration of a computation named `CircArea`. The first block of code calculates the `area` of a circle given the radius `r`. The second block does the inverse. A constant `pi` is defined using the domain algebra before the computation declaration.

```
>let pi be 3.14;
>domain r, area float;

>comp CircArea(area, r) is
{ area <- pi * r * r;}
alt
{ r <- sqrt(area / pi);};
```

Figure 3.30: Declaration of a Simple Computation

Note that this computation can be thought of as the infinite relation shown in Figure 3.31.

CircArea	
area	r
3.14	1
12.56	2
⋮	⋮
38.5	3.5
⋮	⋮

Figure 3.31: Relation Associated with Computation `CircArea`

Every tuple of this relation satisfies the constraint $area = pi * r * r$. Further more, all tuples satisfying this constraint are included in the relation. The parameters of the computation become the domains of its associated relation.

Full support for nested relations in jRelix means that an attribute of a relation can itself be a relation. Thus a computation can take on relational parameter as well.

Example 2. *Invoking a computation using the select/array syntax*

In Figure 3.32, the first `CircleA` stores the result of the area calculation using the array syntax. The second uses a *select* relational expression instead. The semantics of both

is to calculate the area of a circle given the radius. As the array syntax is meant to be syntactic sugar for the relational *select*, the two `CircleA`s contain the same answer, as expected. Note the use of a comma in the first invocation of `CircArea`. This comma is necessary as it indicates that the `1` after it is the second parameter value for the computation. Thus the system will run the second block which requires the second parameter as input. Without the comma, the *select* expression equivalent would be: `[r] where area = 1 in CircArea`.

```
>CircleA <- CircArea[,1];
>pr CircleA;
+---------------+
| area          |
+---------------+
| 3.14          |
+---------------+
relation CircleA has 1 tuple

>CircleA <- [area] where r = 1 in CircArea;
>pr CircleA;
+---------------+
| area          |
+---------------+
| 3.14          |
+---------------+
relation CircleA has 1 tuple
```

Figure 3.32: Computation Invocation: Select/array Syntax

Example 3. *Invoking a computation using the natural join syntax*

Computations are a special kind of relation. Therefore computations can be used in join expressions. In Figure 3.33, we first declare and initialize a relation named `MoreCircs`. When natual joined (i.e. `ijoin`) with the `CircArea` computation, this relation provides the input values of `r`, and the computation returns both the calculated `area` and the input values. The system considers `r` to be the input parameter since it is the only common domain between `MoreCircs` and `CircArea`. Whatever parameter is left in the computation is regarded as output from the computation. Even if `MoreCircs` is defined neither on `r` nor on `area`, we can still explicitly name the join domain (see Figure 3.34).

Example 4. *Invoking a computation using a stand-alone call*

The computation used in the previous examples has primitive typed parameters. In this example we will look at a computation whose parameters are relations. As a matter of

```
>relation MoreCircs(r) <- {(1),(2),(3),(4)};
>MoreArea <- MoreCircs ijoin CircArea;
>pr MoreArea;
+---------------+---------------+
| area          | r             |
+---------------+---------------+
| 3.14          | 1.0           |
| 12.56         | 2.0           |
| 28.26         | 3.0           |
| 50.24         | 4.0           |
+---------------+---------------+
relation MoreArea has 4 tuples
```

Figure 3.33: Computation Invocation: Natural-Join Syntax

```
>domain R float;
>relation MoreCircs(R) <- {(1),(2),(3),(4)};
>MoreArea <- MoreCircs [R:ijoin:r] CircArea;
>pr MoreArea;
+---------------+---------------+
| area          | R             |
+---------------+---------------+
| 3.14          | 1.0           |
| 12.56         | 2.0           |
| 28.26         | 3.0           |
| 50.24         | 4.0           |
+---------------+---------------+
relation MoreArea has 4 tuples
```

Figure 3.34: Computation Invocation: Natural-Join with Named Join Domain

fact, a parameter can be of any type, even computation. Computation `SuperSet`, as defined in Figure 3.35, contains only one block which assigns relation `group1` or `group2` or nothing to `super` according to the following rules:

- If `group1` contains every `name` of `group2`, then assign `group1` to `super`;
- If `group2` contains every `name` of `group1`, then assign `group2` to `super`;
- If the above two conditions both fail, don't assign anything to `super`.

Figure 3.36 shows three stand-alone instances of invocation of the computation each followed by its results. The direction of the arguments being passed are indicated by `in` for input or `out` for output.

3.6.2 Stateful Computations: a Simple Example

Computations come with a facility to create objects with state. It is also possible to define accessor and modifier methods on the state. Figure 3.37 shows a computation, `Counter`. The state is `_curVal`, of type integer. This computation has one parameter

```
>domain name string;
>domain group1, group2(name);
>domain super(name);

>comp SuperSet(group1,group2,super) is
{if (group1 sup group2)
 then
        super <- group1
 else
        if (group2 sup group1)
        then
                super <- group2;
};
```

Figure 3.35: Declaration of a Computation with Relation Typed Parameters

```
>relation G1(name) <- {("Andy"), ("George"), ("Hans")};
>relation G2(name) <- {("Andy"), ("Hans")};
>relation G3(name) <- {("Andy"), ("Chuck")};
>relation super(name);
>relation super1(name);

>SuperSet(in G1, in G2, out super);
>pr super;
+----------------------+
| name                 |
+----------------------+
| Andy                 |
| George               |
| Hans                 |
+----------------------+
relation super has 3 tuples

>SuperSet(in G2, in G1, out super);
>pr super;
+----------------------+
| name                 |
+----------------------+
| Andy                 |
| George               |
| Hans                 |
+----------------------+
relation super has 3 tuples

>SuperSet(in G2, in G3, out super1);
>pr super1;
+----------------------+
| name                 |
+----------------------+
+----------------------+
relation super1 has 0 tuple
```

Figure 3.36: Computation Invocation: Stand-alone Calls

`curVal`, which is used as input in the first block and as output in the second block. When the first block is invoked, the input parameter `curVal` passes its value on to the **state** `_curVal`, thus resetting the value of the state. When the second block is invoked, the state `_curVal` first increments its own value by one and then assigns the value to the output parameter `curVal`. Figure 3.38 demonstrates the use of this computation. In this case only one object is created.

```
>domain curVal intg;

>comp Counter(curVal) is
state _curVal intg;
{ _curVal <- curVal;}
alt
{ _curVal <- _curVal + 1;
  curVal <- _curVal;
};
```

Figure 3.37: Declaration of a Computation with State

```
>ACounter <- Counter[0];
>ACounter <- Counter[];
>pr ACounter;
+-------------+
| curVal      |
+-------------+
| 1           |
+-------------+
relation ACounter has 1 tuple

>BCounter <- Counter[];
>pr BCounter;
+-------------+
| curVal      |
+-------------+
| 2           |
+-------------+
relation BCounter has 1 tuple

>CCounter <- Counter[9];
>CCounter <- Counter[];
>pr CCounter;
+-------------+
| curVal      |
+-------------+
| 10          |
+-------------+
relation CCounter has 1 tuple
```

Figure 3.38: Using a Computation with State

In order to initialize a `Counter` object, the computation should be invoked with a single input parameter. Invoking it without a parameter will fire the second block and

return the value of the incremented state.

3.6.3 Packages

JRelix supports the notion of packages with no new syntax. A package is a computation designed to export other computation(s) defined inside its body. This makes it possible to switch between different implementations of the same computation on the fly, as long as each implementation of the computation is included in a different package **and** the signature of the computation remains the same[5] across all packages in which it is defined. In the example we give in Figures 3.39 and 3.40, two packages are defined, each exporting computation `CalcArea`. `CalcArea` in `package1` calculates the area of a circle given radius, while the one in `package2` gives the area of a square. Since `CalcArea` is a parameter and domain of the package computations, it has to be declared beforehand as a computation-typed domain.

```
domain CalcArea comp(area, r);
```

`Area` is declared to be a *view* on `CalcArea` when the second parameter's value is `2`. The view materializes when used in a relational expression or or as a command argument, such as in

```
pr Area;
```

To export a computation from a package, simply invoke the package with a stand-alone call and prefix the name of the computation being exported with keyword `out`. The exported computation is then ready for use as any other top level computation.In our example, the `pr` command forces the view `Area` to be evaluated which in turn calls `CalcArea` using the array syntax (see 3.6.1 for calling syntax for computations). The relation resulting from the evaluation has one domain `area`, the value of which reflects the package being used.

Since packages are top level compuations that are persistent on secondary storage, once defined, packages can be used over and over again without the need for

[5]This means same name and same parameters. Parameter type counts.

```
>let pi be 3.14;
>domain r, area float;
>domain CalcArea comp(area, r);
>Area is CalcArea[,2];

>comp package1 (CalcArea) is
{comp CalcArea(area,r) is
 {
  area <- pi * r * r;
 };
};

>comp package2 (CalcArea) is
{comp CalcArea(area,r) is
 {
  area <- r * r;
 };
};
```

Figure 3.39: Declaration of Packages

```
>package1(out CalcArea);
>pr Area;
+---------------+
| area          |
+---------------+
| 12.56         |
+---------------+
expression has 1 tuple

>package2(out CalcArea);
>pr Area;
+---------------+
| area          |
+---------------+
| 4.0           |
+---------------+
expression has 1 tuple
```

Figure 3.40: Use of Packages

re-declaration. This opens up the opportunity for packages to be used as libraries. Packages are more useful when they contain commonly used utility computations, such as the ones shown in the example.

3.6.4 Commands

JRelix commands that are useful for computations are summarized as follows:

pr prints the source code of the computation.

dr deletes a relation, view or computation.

sc displays information about the 'alt' blocks of a computation. The format of the output from this command is

$$[\textit{Input Parameter List}] \text{ -> } [\textit{Output Parameter List}]$$

All the above mentioned commands take the name of a computation as the sole argument.

Chapter 4

User's Manual

Computations not only provide support for *procedural abstraction*, as demonstrated in the previous chapter, but they also can be used to implement *data abstraction*. The key ideas of an abstract data type (ADT) are encapsulation and information hiding. In Section 4.1 we will discuss the declaration and use of abstract data types. The reader is encouraged to review the basic concepts of computations presented in Section 3.6. The domain algebra (see Section 3.4) complements the relational algebra by providing much expressive power to jRelix. However, such power is limited to a predefined set of operations, be it arithmetic or relational joins. To allow for user-defined operations on arbitrary data types, an extension to the domain algebra is necessary. The power of such an extension is illustrated in Section 4.2.

4.1 User's Manual on ADT

4.1.1 Introduction

An abstract data type (ADT) in jRelix is declared using the same syntax as a computation. The reader is referred to Appendix A for the BNF syntax of a complete computation declaration. The structure of a computation is show in Figure 4.1. Square brackets include optional elements, such as ParameterList. Items that may appear zero

or more times are delimited by ()*. Key words and all mandatory delimiters are shown in bold face. A vertical bar separates alternative items.

```
comp  CompName ([ParameterList ]) is
[ComputationVariableDeclaration  ]
 [redop|funop]
 {
  (Statement |Command )*
  }
( alt
  [redop|funop]
  {
   (Statement |Command )*
  }
 )*
```

Figure 4.1: The Structure of a Computation

An ADT has hidden information[1] that is declared as **state** variables. Operations on the states are given by the nested-level compuations inside the ADT, thus the support for encapsulation. Two types of operations are normally supplied for an ADT: accessor methods and modifier methods. In most cases, an accessor method reveals the value of the hidden **state** while a modifier method provides a means of changing its value. It is however also possible for an accessor method to be defined otherwise, as long as it is purely *functional* (i.e. not modifying the state). Methods whose name appears in the parameter list of the ADT become public methods.

Once an ADT has been declared, objects of that type can be instantiated via an intersection join (**ijoin**). The relation that joins with the ADT may contain initial values for the hidden state(s). The result of the join is a relation, which contains the hidden state(s). Its domains include the associated public methods. The hidden state can only be manipulated through the methods defined in the ADT. Each tuple of the result relation constitutes an object of the ADT. Since the result relation is by all means a normal relation, the relational algebra and domain algebra operations introduced in

[1] The original concept of information hiding applies to hiding the method implementation as well. This is not supported by jRelix.

Chapter 3 still apply.

In the following sub-sections, we will see two examples of using ADTs. The first one illustrates object instantiation, and the use of accessor/modifier methods on a `float` state variable. The second is a more realistic application that involves almost all aspects of the jRelix language we have seen so far.

4.1.2 Example 1: Car Racing

Declaration and Instantiation

In this example, we define a `RaceCar` ADT as shown in Figure 4.2. There are three **state** variables defined in the ADT: `_v0`, `_a`, and `_v`. In view of the motion of uniform acceleration, these represent the initial velocity, the acceleration, and the current velocity, in their standard units respectively. The two modifier methods, `ACCELERATE` and `STOP`, apply the rules of uniform acceleration to the state variables. `ACCELERATE` accelerates the race car for `t` seconds whereas `STOP` brings the car to a total stop. The only accessor method for this ADT is `GETSPEED` which is meant to reveal the current velocity of a car. All three methods are public, as they are present in the parameter list. The other two variables in the parameter list are `v0` and `a`. They specifiy the initial values for states `_v0` and `_a`. Obviously their values need to be provided when the `RaceCar` ADT is instantiated. This example also shows how domains of type `computation` are declared. It is important to declare all domains in the parameter list of an ADT before the declaration of the ADT itself.

The input and output relationship among the parameters can be displayed by an `sc` command, as in Figure 4.3. The result of this command provides clue as to what inputs are expected by the ADT and what outputs are available.

Figure 4.4 declares a relation `Racers` which contains two racers each with his own name, initial velocity, and acceleration.

To instantiate two `RaceCar` objects, we simply `ijoin` the relation `Racers` and the ADT `RaceCar` (see Figure 4.5). The `Racers` relation could contain thousands of tuples

```
>domain v0, vt, a, t, curSpeed float;
>domain ACCELERATE comp(t);
>domain STOP comp();
>domain GETSPEED comp(curSpeed);
>comp RaceCar(v0,a,ACCELERATE,STOP,GETSPEED) is
state _v0, _a, _v float;
{_v0 <- v0; _a <- a; _v <- v0;

 comp ACCELERATE(t) is
 { _v <- _v0 + _a * t; _v0 <- _v;};

 comp STOP() is
 { _v <- 0; _v0 <- _v;};

 comp GETSPEED(curSpeed) is
 { curSpeed <- _v; };
};
```

Figure 4.2: Example ADT: `RaceCar`

```
>sc RaceCar;
RaceCar (v0, a, ACCELERATE, STOP, GETSPEED)
  [ v0 a ] -> [ ACCELERATE STOP GETSPEED ]
```

Figure 4.3: `RaceCar` Input and Output

```
>domain name string;
>domain v0', a' float;
>relation Racers(name, v0', a') <- {("James Bond", 0, 6000),
                                   ("Michael Schumacher", 0, 5500)};
```

Figure 4.4: The `Racers`

instead, and the instantiation would still require just one join operation. This way of object instantiation is easier than in other languages such as C++ or Java, where only one new object at a time can be instantiated. This form of instantiation is thus appropriate for databases which normally contain large amounts of data.

```
>TopRacers <- [name,ACCELERATE,STOP,GETSPEED] in (RaceCar[v0,a:ijoin:v0',a']Racers);
>pr TopRacers;
+---------------------+-------------+-------+-----------+
| name                | ACCELERATE  | STOP  | GETSPEED  |
+---------------------+-------------+-------+-----------+
| James Bond          | 1           | 2     | 3         |
| Michael Schumacher  | 1           | 2     | 3         |
+---------------------+-------------+-------+-----------+
relation TopRacers has 2 tuples
```

Figure 4.5: Instantiating the `RaceCar` ADT

Consider the join syntax used in Figure 4.5. Were the relation `Racers` defined on `name`, `v0`, and `a`, the following join syntax could be used:

```
TopRacers <- [name,ACCELERATE,STOP,GETSPEED] in (RaceCar ijoin Racers);
```

In this case, the common domains of `RaceCar` and `Racers`, i.e. `v0` and `a`, become the join domains which, in turn become the input parameters of the `RaceCar` ADT. The remaining parameters become outputs. However, in our example, the relation `Racers` are defined on domains not in the parameter list of the ADT. Therefore the extended join syntax is used to explicitly name the join domains. This may seem a bit lengthy, but in return we gain the freedom of defining `Racers` on any domains as long as they are compatible with the types of the ADT's input parameters.

The projector list in the T-Selection in Figure 4.5 contains the domains we want in the result, `TopRacers`. Since the initializers (`v0` and `a`) are no longer needed, they are omitted from the list. Though not shown, there are yet three hidden variables in `TopRacers`: the three states in the `RaceCar` ADT. Thus the result of an ijoin between an ADT and a relation is defined on a set of domains consisting of the union of the domains of its two operands **and** the hidden states of the ADT. To show these hidden states, we can turn on the `debug` switch as in Figure 4.6. The names of the hidden states are quoted in '*'. Although hidden states can be displayed this way, they cannot be

used anywhere normal domains are expected. This restriction ensures the information hiding aspect of an abstract data type.

```
>debug;
Note: debug mode is on
>pr TopRacers;
+--------------------+------------+------+----------+--------+-------+--------+
| name               | ACCELERATE | STOP | GETSPEED | *_a*   | *_v*  | *_v0*  |
+--------------------+------------+------+----------+--------+-------+--------+
| James Bond         | 1          | 2    | 3        | 6000.0 | 0.0   | 0.0    |
| Michael Schumacher | 1          | 2    | 3        | 5500.0 | 0.0   | 0.0    |
+--------------------+------------+------+----------+--------+-------+--------+
relation TopRacers has 2 tuples
```

Figure 4.6: Showing Hidden Attributes

Figure 4.6 also proves that the assignment statements in the beginning of the computation block of `RaceCar` have been executed correctly. It is only in a computation block that assignment to a scalar typed variable is allowed. The numbers in the columns titled `ACCELERATE`, `STOP` and `GETSPEED` are what we call "surrogates". They function like pointers to the three public methods exported by the ADT. Although each tuple[2] in `TopRacers` has its own states, the methods are the same for all tuples.

Method Invocation

Now we discuss how the accessor methods are used. The following statement cannot be used to examine the current velocity of the car because `_v` is a hidden state:

```
pr ([name, _v] in TopRacers);
```

Instead we must use the accessor method `GETSPEED`. Figure 4.7 shows how.

First we declare a virtual attribute `speed'` to hold the result of the computation call `GETSPEED[]`. This invocation uses the array syntax (see Section 3.6.1 for a discussion of calling syntax) and produces a relation defined on `curSpeed`. Thus it is **unary**. In addition, the relation `speed'` has only one tuple, i.e., the value of `_v`. So `speed'` is also **singleton**. However, it is cumbersome to have a nested relation in the result when all we need is the value of the tuple in the unary singleton relation. Thus the second `let`

[2]A tuple here can be thought of as an object, but we prefer the term tuple in this relational database context

```
>let speed' be GETSPEED[];
>let speed be [red nop of curSpeed] in speed';
>AllSpeeds is [name, speed] in TopRacers;
>pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 0.0           |
| Michael Schumacher   | 0.0           |
+----------------------+---------------+
expression has 2 tuples
```

Figure 4.7: Using the Accessor Method

statement is employed to give a float value: `speed`. The expression

```
[red nop of curSpeed] in speed'
```

produces the value of `curSpeed` in relation `speed'`[3]. And this value will be assigned to `speed` when `speed` is actualized. The `pr` command in Figure 4.7 triggers such an actualization. (We define `AllSpeeds` as a view in anticipation of its repeated use in later code. More details on views, computed relations, can be found in [Hao98]).

A more succinct way of defining `speed` is:

```
let speed be [red nop of curSpeed] in GETSPEED[];
```

To start the cars racing, the modifier method `ACCELERATE` comes into play. As it actually changes the state of the "race cars", an `update` statement is involved, as illustrated in Figure 4.8. The method `ACCELERATE` works on each tuple in `TopRacers` in turn. As a result, each racer has been accelerating for 3 seconds.

```
>update TopRacers change ACCELERATE(in 3);
>pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 18000.0       |
| Michael Schumacher   | 16500.0       |
+----------------------+---------------+
expression has 2 tuples
```

Figure 4.8: Using the Modifier Method: Example 1

Of course we can choose to accelerate `James Bond`'s car only, and even stop `Michael Schumacher` all together (see Figure 4.9).

[3]see Section 3.3.2 for a discussion on the operator `nop` and Section 3.4.2 on its role with reduction

```
>update TopRacers change ACCELERATE(in 2)
 using ijoin where name = "James Bond" in TopRacers;
>pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 30000.0       |
| Michael Schumacher   | 16500.0       |
+----------------------+---------------+
expression has 2 tuples

>update TopRacers change STOP()
 using ijoin where name = "Michael Schumacher" in TopRacers;
>pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 30000.0       |
| Michael Schumacher   | 0.0           |
+----------------------+---------------+
expression has 2 tuples
```

Figure 4.9: Using the Modifier Method: Example 2

States in jRelix are persistent on secondary storage. This means we can quit the current session, restart in the same database directory, and continue with the car race example as if no interruption has occurred. Figure 4.10 illustrates this point. Note in particular the value of the state variables for `James Bond` and `Michael Schumacher`. The surrogates for the three public methods remain the same as in the previous run as well.

The mechanism of instantiation and method invocation introduced in this sub-section is generally applicable to all ADTs. In the next example, we will focus on the integration of various database operations into an ADT's methods in a more realisitc application.

4.1.3 Example 2: A Banking Application

Declaration and Instantiation

Consider a banking application that involves multiple banks, each with its own customers. Here is a list of possible requirements for such an application:

1. The banks are distinguished from each other by their name.
2. Each bank has a number of customers identified by their account number.
3. There is a non-negative balance in each account.

```
>quit;
[start a new session]
>relation NewRacers(name, v0, a) <- {("Spiderman", 0, 4000)};
>NewTopRacer <- [name,ACCELERATE,STOP,GETSPEED] in (RaceCar ijoin NewRacers);
>update TopRacers add NewTopRacer;
>pr TopRacers;
+----------------------+-------------+-------+-----------+
| name                 | ACCELERATE  | STOP  | GETSPEED  |
+----------------------+-------------+-------+-----------+
| James Bond           | 1           | 2     | 3         |
| Michael Schumacher   | 1           | 2     | 3         |
| Spiderman            | 1           | 2     | 3         |
+----------------------+-------------+-------+-----------+
relation TopRacers has 3 tuples

>update TopRacers change ACCELERATE(in 4)
 using ijoin where name = "Spiderman" in TopRacers;
>pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 30000.0       |
| Michael Schumacher   | 0.0           |
| Spiderman            | 16000.0       |
+----------------------+---------------+
expression has 3 tuples
```

Figure 4.10: Persistent State

4. The customer can see his/her balance and make deposits or withdrawals as long as the account is in good standing (not overdrawn).
5. The bank needs to tally the sum of the balances in all acounts. It also needs to know how many open accounts it has.
6. The bank should be able to open and close accounts.
7. The bank can transfer money from one account to another.

We devise two ADTs: one for the bank account (called `BA`) and the other for the bank (named `BANK`). The `BA` ADT stores the current balance of an account in (`bal`) and also remembers the last balance (`oldbal`). It provides one accessor method (`BALANCE`) and one modifier method (`DEPOSIT`). The method `BALANCE` consists of two 'alt' blocks: the first returns the current balance while the second gives the amount of the most recent deposit. The modifier method can actually be used to withdraw money when the parameter value is negative. It also prevents any attempt to overdraw an account. The complete definition of the `BA` ADT is shown in Figure 4.11.

```
>domain Dep,Bal,InitBal intg;
>domain DEPOSIT comp(Dep);
>domain BALANCE comp(Bal);
>comp BA(InitBal,DEPOSIT,BALANCE) is
 state bal,oldbal intg;
 { bal <- InitBal;

   comp DEPOSIT(Dep) is
   { oldbal <- bal; bal <- bal + Dep;}
   alt
   { Dep <- bal - oldbal;};

   comp BALANCE(Bal) is
   { Bal <- bal;};
 };
```

Figure 4.11: The BA ADT

To satisfy requirements 5 to 7, we define the BANK ADT as in Figures 4.12 and 4.13. Since this is a lengthy definition, we break it down into two pieces and number the lines continuously (Figures 4.13 gives the complete definition of the ADT). The line numbers will be referred to when we explain this example in detail.

```
1 >let BA' be BA;
2 >domain Acct strg;
3 >domain CUST(Acct,InitBal);
4 >domain TotBal intg;
5 >domain Amount intg;
6 >let FromAcct be Acct;
7 >let ToAcct be Acct;
8 >domain TOCLOSE(Acct);
9 >domain Cnt intg;
10
11 >domain SUM comp(TotBal);
12 >domain TRANSFER comp(Amount,FromAcct,ToAcct);
13 >domain OPENACCT comp(CUST,BA');
14 >domain CLOSEACCT comp(TOCLOSE);
15 >domain ACCTCNT comp(Cnt);
```

Figure 4.12: The BANK ADT: Part 1

Let's first see how we can instantiate five BANK objects. The names and initial customer accounts of our banks are given by relation BIG5_CUST in Figure 4.14. The domain corp serves as the primary key while CUST is a relation typed domain defined on Acct, the account number, and InitBal, the initial balance, as shown on line 3 of Figure 4.12.

The instantiation of the BANK ADT involves an ijoin between the ADT and an initializing relation. In order to initialize the state variable customers upon instantiation (line 18 in Figure 4.13), the initializing relation must contain at least two domains com-

```
16 >comp BANK(CUST,BA',SUM,TRANSFER,OPENACCT,CLOSEACCT,ACCTCNT) is
17  state customers(Acct,DEPOSIT,BALANCE);
18  {customers <- [Acct,DEPOSIT,BALANCE] in (CUST ijoin BA');
19
20   comp SUM(TotBal) is
21   {let CustBal be [red nop of Bal] in BALANCE[];
22    TotBal <- [red + of CustBal] in customers;
23   };
24
25   comp TRANSFER(Amount,FromAcct,ToAcct) is
26   {let FBal2 be [red nop of Bal] in BALANCE[];
27    let FBal be FBal2-Amount;
28    FA <- where Acct=FromAcct in customers;
29    TA <- where Acct=ToAcct in customers;
30    FAS <- where FBal>=0 in FA;
31    if ([ ] in FAS) and ([ ] in TA)
32    then
33    {let nAmount be 0-Amount;
34     update customers change DEPOSIT (in nAmount)
35      using ijoin where Acct=FromAcct in customers;
36     update customers change DEPOSIT(in Amount)
37      using ijoin where Acct=ToAcct in customers;
38    }
39    else
40     print "Error in TRANSFER"
41   };
42
43   comp OPENACCT(CUST, BA') is
44   {new_customers <- [Acct,DEPOSIT,BALANCE] in (CUST ijoin BA');
45    update customers add new_customers;
46   };
47
48   comp CLOSEACCT(TOCLOSE) is
49   {update customers delete TOCLOSE;};
50
51   comp ACCTCNT(Cnt) is
52   {Cnt <- [red + of 1] in customers;};
53  };
```

Figure 4.13: The BANK ADT: Part 2

```
>domain corp strg;
>relation BIG5_CUST(corp, CUST) <- {("BMO",{("S001", 100),
                                            ("C002", 20),
                                            ("S003", 300)}),
                                   ("RBC",{("S001", 30000),
                                            ("S002", 2000)}),
                                   ("TD",{("C003", 30)}),
                                   ("CIBC",{("S001", 30000),
                                             ("C002", 100),
                                             ("C003", 20000),
                                             ("C004", 90)}),
                                   ("Scotia",{("C001", 40),
                                              ("C002", 500),
                                              ("S003", 90000)})
                                  };
```

Figure 4.14: The Big 5 Banks and Their Customers

patible with the input parameters of BANK: CUST and BA'. The latter is a virtual domain defined on the top-level computation BA (see line 1). The relation BIG5_CUST contains CUST already, but it does not have any computation typed domain. Since BA' is a virtual domain, we include it by the following T-selection:

```
BIG5_BANK <- [corp,CUST,BA'] in BIG5_CUST;
```

The relation BIG5_BANK can now be used to instantiate the bank objects, as shown in Figure 4.15. The ijoin between BANK and BIG5_BANK on common domains leads the system to run the only 'alt' block of BANK for each tuple of BIG5_BANK. The first statement (line 18) in the block instantiates BA objects and stores them in customers. The rest of the statements are computation declarations which are simply parsed and kept in memory for future use. Next, the state customers becomes the hidden variable of the relation BIG5 while its accessor and modifier methods are exported as visible domains of the same relation. Note the debug mode is on so that we can observe the internal states (quoted in '*') directly. The "dot relation" that holds the contents of customers shows the two states of the bank account (bal and oldbal) and their associated methods.

Method Invocation

In this sub-section, we will carry out a series of transactions[4] on behalf of the banks. The examples are cumulative and therefore ought to be followed in sequence. Line numbers refer to those that appear in Figures 4.12 and 4.13 in the previous sub-section. Surrogate values can be linked to bank names via Figure 4.15.

Example 1. *Transfer 40 dollars from account "C001" to "C002" of the Scotia Bank.* This operation will change the state of the bank objects, therefore an update statement is used. Figure 4.16 illustrates the invocation of the method TRANSFER on the tuple that corresponds to the Scotia bank. Three input parameters are provided, as indicated by the **in** keyword before each actual parameter.

[4] This has nothing to do with database transactions which are concerned about maintaining consistency and integrity during updates.

```
>BIG5 <-  [corp,SUM,TRANSFER,OPENACCT,CLOSEACCT,ACCTCNT] in (BIG5_BANK ijoin BANK);
>debug;
Note: debug mode is on
>pr BIG5;
+--------+------+----------+----------+-----------+---------+-------------+
| corp   | SUM  | TRANSFER | OPENACCT | CLOSEACCT | ACCTCNT | *customers* |
+--------+------+----------+----------+-----------+---------+-------------+
| BMO    | 10   | 11       | 12       | 13        | 14      | 9           |
| CIBC   | 10   | 11       | 12       | 13        | 14      | 15          |
| RBC    | 10   | 11       | 12       | 13        | 14      | 16          |
| Scotia | 10   | 11       | 12       | 13        | 14      | 17          |
| TD     | 10   | 11       | 12       | 13        | 14      | 18          |
+--------+------+----------+----------+-----------+---------+-------------+
relation BIG5 has 5 tuples
>pr .customers;
+-----+------+---------+---------+-------------+-------------+
| .id | Acct | DEPOSIT | BALANCE | *bal*       | *oldbal*    |
+-----+------+---------+---------+-------------+-------------+
| 9   | C002 | 7       | 8       | 20          | 0           |
| 9   | S001 | 7       | 8       | 100         | 0           |
| 9   | S003 | 7       | 8       | 300         | 0           |
| 15  | C002 | 7       | 8       | 100         | 0           |
| 15  | C003 | 7       | 8       | 20000       | 0           |
| 15  | C004 | 7       | 8       | 90          | 0           |
| 15  | S001 | 7       | 8       | 30000       | 0           |
| 16  | S001 | 7       | 8       | 30000       | 0           |
| 16  | S002 | 7       | 8       | 2000        | 0           |
| 17  | C001 | 7       | 8       | 40          | 0           |
| 17  | C002 | 7       | 8       | 500         | 0           |
| 17  | S003 | 7       | 8       | 90000       | 0           |
| 18  | C003 | 7       | 8       | 30          | 0           |
+-----+------+---------+---------+-------------+-------------+
relation .customers has 13 tuples
```

Figure 4.15: Instantiating 5 BANK objects

```
>update BIG5
  change TRANSFER(in 40,in "C001",in "C002")
 using ijoin where corp="Scotia" in BIG5;
>pr .customers;
+-----+------+---------+---------+-------------+-------------+
| .id | Acct | DEPOSIT | BALANCE | *bal*       | *oldbal*    |
+-----+------+---------+---------+-------------+-------------+
| ... | ...  | ...     | ...     | ...         | ...         |
| 17  | C001 | 7       | 8       | 0           | 40          |
| 17  | C002 | 7       | 8       | 540         | 500         |
| 17  | S003 | 7       | 8       | 90000       | 0           |
| ... | ...  | ...     | ...     | ...         | ...         |
+-----+------+---------+---------+-------------+-------------+
relation .customers has 13 tuples
```

Figure 4.16: Transfer Money between Two Accounts

The statements in the computation block of `TRANSFER` are executed in sequence. Line 21 defines a virtual domain `FBal2` to hold the return value from the computation call `BALANCE[]`. The use of reduction and the `nop` operator has been discussed in Section 4.1.2 and will not be repeated here. `FBal` holds the difference between the current balance and the amount to transfer. Lines 28 to 30 define three relations. `FA` holds the tuple of the account to be transferred from; `TA` holds that of the account to be deposited into; and finally `FAS` is the same as `FA` if there is sufficient funds for withdrawal, otherwise it is empty. If the conditions in line 31 hold, that is, if both accounts exist and the source account has enough money, the code in the 'then' block is executed. Otherwise an error message is printed on the screen. The actual transfer involves withdrawing money from the source account and depositing the same amount in the destination account. This is achieved by using the DEPOSIT method exported by the `BA` ADT. Note here we are strictly observing the rule of information hiding by not revealing the states (`bal` and `oldbal`) of the bank account.

The result shows only the tuples of customers related to the Scotia bank (surrogate value 17). As expected, the execution of the DEPOSIT method results in the `oldbals` of the affected accounts taking on the previous values of `bal` and the `bals` of accounts "C001" and "C002" have been updated according to the amount of transfer.

Example 2. *Add new accounts to bank BMO.*

An update statement is needed in this case. Figure 4.17 shows the details of the new accounts and illustrates the result of invoking the method `OPENACCT`. This time, the computation `BA` is supplied as an input parameter. The code (line 44) in the method `OPENACCT` resembles that of the first statement of `BANK`. Line 45 actually adds the new accounts by an `update add` operation.

Example 3. *Delete some accounts from BMO.*

Again, since this operation will change the state of the bank objects, an `update` statement is used. Figure 4.18 shows the statement and the result. The relation `CLOSE` designates the accounts to be closed. An `update delete` operation is all that is needed in the method `CLOSEACCT`.

```
>relation NEW_BMO_CUST(Acct,InitBal) <- {("C004", 900), ("S005", 0)};
>update BIG5
  change OPENACCT(in NEW_BMO_CUST,in BA)
 using ijoin where corp="BMO" in BIG5;
>pr .customers;
+------+------+---------+---------+-------------+-------------+
| .id  | Acct | DEPOSIT | BALANCE | *bal*       | *oldbal*    |
+------+------+---------+---------+-------------+-------------+
| 9    | C002 | 7       | 8       | 20          | 0           |
| 9    | C004 | 7       | 8       | 900         | 0           |
| 9    | S001 | 7       | 8       | 100         | 0           |
| 9    | S003 | 7       | 8       | 300         | 0           |
| 9    | S005 | 7       | 8       | 0           | 0           |
| ...  | ...  | ...     | ...     | ...         | ...         |
+------+------+---------+---------+-------------+-------------+
relation .customers has 15 tuples
```

Figure 4.17: Open New Accounts

```
>relation CLOSE(Acct) <- {("C004"), ("S001")};
>update BIG5
  change CLOSEACCT(in CLOSE)
 using ijoin where corp="BMO" in BIG5;
>pr .customers;
+-----+------+---------+---------+-------------+-------------+
| .id | Acct | DEPOSIT | BALANCE | *bal*       | *oldbal*    |
+-----+------+---------+---------+-------------+-------------+
| 9   | C002 | 7       | 8       | 20          | 0           |
| 9   | S003 | 7       | 8       | 300         | 0           |
| 9   | S005 | 7       | 8       | 0           | 0           |
| ... | ...  | ...     | ...     | ...         | ...         |
+-----+------+---------+---------+-------------+-------------+
relation .customers has 13 tuples
```

Figure 4.18: Close Accounts

Example 4. *Tally the sum of balances and total number of accounts for every bank.* This transaction does not change the state of the banks. Figures 4.19 and 4.20 show the statements and the results. The use of reduction with the `nop` operator was covered in Section 4.1.2.

```
>let banksum be [red nop of TotBal] in SUM[];
>Big5Sum <- [corp, banksum] in BIG5;
>pr Big5Sum;
+----------------------+-------------+
| corp                 | banksum     |
+----------------------+-------------+
| BMO                  | 320         |
| CIBC                 | 50190       |
| RBC                  | 32000       |
| Scotia               | 90540       |
| TD                   | 30          |
+----------------------+-------------+
relation Big5Sum has 5 tuples
```

Figure 4.19: Tally Sum of Balances

```
>let AcctCnt be  [red nop of Cnt] in ACCTCNT[];
>Big5Cnt <- [corp, AcctCnt] in BIG5;
>pr Big5Cnt;
+----------------------+-------------+
| corp                 | AcctCnt     |
+----------------------+-------------+
| BMO                  | 3           |
| CIBC                 | 4           |
| RBC                  | 2           |
| Scotia               | 3           |
| TD                   | 1           |
+----------------------+-------------+
relation Big5Cnt has 5 tuples
```

Figure 4.20: Tally Total Counts of Accounts

4.1.4 Summary

The properties of ADTs in jRelix are summarized as follows:

- An ADT is declared as a computation with state(s).
- An ADT contains the accessor or modifier methods for the state(s).
- The state variable in an ADT can be of any domain type except computation.
- Objects of an ADT are instantiated by an ijoin between the ADT and an initializing relation. It is customary for the initializing relation to supply intial values for the state(s) via the join domains. It may also contain object identifiers.

For example, `corp` *may be regarded as an object identifier for the bank objects in the bank account example.*

- The result of an instantiation is a normal relation containing the hidden state(s) of the ADT. It also contains the methods exported through the parameter list of the ADT.
- The hidden state(s) in the result of an instantiation may not be used as ordinary domains. But they exist in relations derived from the result by means of a selection, projection, or T-selection.

  For example, after executing `Corps <- [corp] in BIG5`*, relation* `Corps` *also has the hidden state* `customers`*. This is merely a design choice, as one cannot specify the state in the projection list; so if the state is not retained, it will never be usable beyond the initial result of an instantiation.*
- Modifier methods must be invoked with an `update change` statement.
- Accessor methods are used in the domain algebra. Since they produce relation typed results, it is recommended to use reduction and the `nop` operator to unnest a unary singleton relation.

4.2 User's Manual on Extended Domain Algebra

4.2.1 Introduction

This section discusses an extension to the vertical domain algebra. The four classes of vertical operations (reduction, equivalence reduction, functional mapping, and partial functional mapping) can be used to combine values of a domain across tuples in a systematic way. The typical applications are to calculate the grand total, sub-totals, or generate ranks of certain numerical domains. They are more powerful than the common aggregate functions in SQL and more intuitive to use as well.

Nevertheless, the capability of these vertical operations is limited to the few system defined operators compatible in such operations. Take reduction as an example: only

seven operators are allowed on scalar operands and four for relation typed operands (see Section 3.4.2). Should there arise a need to include in reduction another associative commutatitve operator, the implementation of the domain algebra would have to be changed and the whole system affected. To alleviate this restriction, we introduce a mechanism to embed computation calls in the vertical operations. Such computations may be defined to carry out any legitimate operations not pre-defined by the system.

4.2.2 New Syntax

The syntax change to support extended vertical domain operations comes in two parts. First, the sytax for computation declaration is enhanced with the keywords, "redop" and "funop". An 'alt' block intended for use in reduction must be prefixed with "redop". However, this does not prevent it from being used in functional mapping. On the other hand, "funop" is used to decorate blocks that are available for functional mapping only. It is common for a computation to have one "redop" block and several "funop" blocks, but this is not mandatory. For example, a computation describing the behavior of the logic operator "xor" may have three "redop" blocks. It is also possible for a computation to have, at the same time, non-decorated blocks and decorated blocks. Invoking a block not decorated with "redop" or "funop" in a vertical operation will generate an error.

The only change in syntax for vertical domain declaration is that the operator `OP` may now be replaced with a computation call in the following format:

CompName ["(" InOutList ")"]

where `CompName` is the name of a computation, and `InOutList` consists of a sequence of two "in" and one "out", separated from each other by a comma. The parenthesized list may be omitted, in which case it is syntactic sugar for:

CompName "(" "in" "," "in" "," "out" ")"

The `InOutList` specifies the position of the input and output parameters. It is used to determine which block of the computation is targeted. For example, the following are legal invocations of the `cplx1` computation defined in Figure 4.23 (equivalence reduction and partial functional mapping now shown):

- `red cplx1(in,in,out) of R`
- `red cplx1 of R` (syntactic sugar for the previous one)
- `fun cplx1(in,out,in) of R order id`
- `fun cplx1 of R order id`

The first two are used for calculating the sum of complex numbers stored in `R`; the last two are for the calculation of the alternating sum and the cumulative sum, respectively. Note that examples 1, 2 and 4 invoke the same block of the `cplx1` computation, namely the one decorated with "redop". The third example invokes the second "funop" block. Although all "redop" blocks are available for use with functional mapping, none of the "funop" blocks can be invoked in reduction. This is because the "redop" blocks are intended for operations that are commutative and associative, whereas there is no such restriction on the "funop" blocks. This being said, it is still the programmer's responsibility to obey these rules while writing the computation. It is impossible for the system to deduce or verify whether the operation described by a "redop" block is indeed commutative and associative.

4.2.3 Example 1: Vertical String Concatenation

Consider the relation in Figure 4.21. Each word is followed by a space and is numbered according to its position in the text.

```
>pr Text;
+-------------+----------------------+
| index       | word                 |
+-------------+----------------------+
| 1           | Aldat                |
| 2           | is                   |
| 3           | fun                  |
+-------------+----------------------+
relation Text has 3 tuples
```

Figure 4.21: A Relation Containing Ordered Strings

We would like to have a string to represent the text as a whole: "Aldat is fun". jRelix has a binary string operator `cat`, but unfortunately it currently cannot be used

in functional mapping[5]. So we need something to mimic the following statement:

```
let WholeText be fun cat of word order index;
```

With extended domain algebra, we have a solution as shown in Figure 4.22.

```
>domain s1,s2,s3 strg;
>comp STRCAT(s1,s2,s3) is
funop {s3 <- s1 cat s2;};
>let WholeText' be fun STRCAT(in, in, out) of word order index;
>let WholeText be red max of WholeText';
>Temp <- [index,word,WholeText',WholeText] in Text;
>pr Temp;
+-------+--------+---------------+--------------+
| index | word   | WholeText'    | WholeText    |
+-------+--------+---------------+--------------+
| 1     | Aldat  | Aldat         | Aldat is fun |
| 2     | is     | Aldat is      | Aldat is fun |
| 3     | fun    | Aldat is fun  | Aldat is fun |
+-------+--------+---------------+--------------+
relation Temp has 3 tuples
>R <- [WholeText] in Text;
>pr R;
+----------------------+
| WholeText            |
+----------------------+
| Aldat is fun         |
+----------------------+
relation R has 1 tuple
```

Figure 4.22: Extended Vertical String Concatenation

First of all, we define a computation with three string parameters. The block of the computation is decorated with the keyword **funop**, indicating its candidacy for use with functional mapping or partial functional mapping. The only statement in the block assigns the concatenation of the first two parameter values to the third. Next, a virtual domain `WholeText'` is defined, in much the same way as we would in any ordinary functional mapping. The difference is that the keyword **fun** is now followed by a computation call. This call identifies the computation (`STRCAT`) and the block within by giving the directions of its parameters. The results of actualizing this virtual attribute are shown in the relation `Temp`. As we are only interested in the longest of the accumulated texts, a second virtual domain `WholeText` is defined. Finally, relation `R` contains the result we want.

[5]This is an omission in the implementation

4.2.4 Example 2: Sum of Complex Numbers

The previous example illustrates an extension to the scalar vertical operation. In the next, we show how to calculate the sum of some complex numbers represented by nested relations. This can be regarded as extending

```
let Total be red + of Part;
```

Only that in our case `Part` is a relation typed domain. See Figures 4.23 and 4.24 for two alternative ways of declaring the computation for complex numbers. The reason for having alternatives will be explained shortly.

```
>domain r,i intg;
>domain R1,R2,R3(r,i);
>comp cplx1(R1,R2,R3) is
 redop { let r' be r; let i' be i;
         let r'' be r+r'; let i'' be i+i';
         R3' <- [r'',i''] in (R1 ijoin ([r',i'] in R2));
         let r be r''; let i be i'';
         R3 <- [r,i] in R3';
       }alt
funop  { let r' be r; let i' be i;
         let r'' be r-r'; let i'' be i-i';
         R1' <- [r'',i''] in (R3 ijoin ([r',i'] in R2));
         let r be r''; let i be i'';
         R1 <- [r,i] in R1';
       }alt
funop  { let r' be r; let i' be i;
         let r'' be r-r'; let i'' be i-i';
         R2' <- [r'',i''] in (R3 ijoin ([r',i'] in R1));
         let r be r''; let i be i'';
         R2 <- [r,i] in R2';
       };
```

Figure 4.23: The Computation for Complex Numbers: Alternative 1

Both of the two alternatives take three relation typed parameters, each defined on `r` and `i`. Think of `r` as the real part of a complex number and `i` the imaginary part. We assume that each parameter stands for one complex number only, i.e., `R1`, `R2` and `R3` are all singleton relations. There are three 'alt' blocks to each alternative computation. The first is decorated with the keyword **redop** and the rest with **funop**. Thus the first block is designed to be used as an associative and commutative operator while the other two are to be used in order-dependent operations. The code in the first block calculates the sum of the complex numbers represented by two of the parameters and assigns the result

```
>comp cplx2(R1,R2,R3) is
redop { let r' be r; let i' be i;
        let r'' be r+r'; let i'' be i+i';
        R2' <- [r'',i''] in (R1 ijoin ([r',i'] in R3));
        let r be r''; let i be i'';
        R2 <- [r,i] in R2';
      }alt
funop { let r' be r; let i' be i;
        let r'' be r-r'; let i'' be i-i';
        R1' <- [r'',i''] in (R2 ijoin ([r',i'] in R3));
        let r be r''; let i be i'';
        R1 <- [r,i] in R1';
      }alt
funop { let r' be r; let i' be i;
        let r'' be r-r'; let i'' be i-i';
        R3' <- [r'',i''] in (R2 ijoin ([r',i'] in R1));
        let r be r''; let i be i'';
        R3 <- [r,i] in R3';
      };
```

Figure 4.24: The Computation for Complex Numbers: Alternative 2

to the third parameter. The second and third blocks subtract one complex number from another and assign the difference to the third parameter.

The only difference between the computation `cplx1` and `cplx2` is that in the former, `R3` represents the sum while in the latter, `R2` takes its place. This difference affects the way the computation is invoked in vertical operations.The example in Figure 4.25 shows that `Sum1` and `Sum2` are exactly the same, but they are each produced by a different computation call in a reduction. The call that generates `Sum1` specifies the third parameter as output, corresponding to `R3`, whereas in the case of `Sum2` the second parameter is designated as output. Semantically the two computations are exactly the same, but this example illustrates the following points:

- For a computation to be used in vertical operations, it must have exactly three parameters, all of the same type.
- Each 'alt' block must have two input parameters and one output parameter, as the operators allowed in vertical operations must be binary. But the order of the three parameters is up to the programmer defining the computation.
- The order of the keywords **in** and **out** in the computation call of the vertical domain operation must match that of the parameters' direction of the intended 'alt' block.

Figure 4.25 also shows the the reduction operation in the projector list of a T-Selection. Since the reduction operation always produces a unary singleton relation, enclosing it directly in the square brackets has the effect of lifting the value of its tuple to the top level. Had we defined a virtual domain on the same operation and used that attribute in the T-selection, we would have ended up with `Sum1` (or `Sum2`) being a nested relation.

```
>domain id intg;
>domain R(r,i);
>relation CNumbers(id,R) <-  {(1,{(2,-2)}),
                             (2,{(3,2)}),
                             (3,{(1,0)}),
                             (4,{(-1,1)})};
>Sum1 <- [red cplx1 of R] in CNumbers;
>pr Sum1;
+---------------+---------------+
| r             | i             |
+---------------+---------------+
| 5             | 1             |
+---------------+---------------+
relation Sum1 has 1 tuple

>Sum2 <- [red cplx2(in,out,in) of R] in CNumbers;
>pr Sum2;
+---------------+---------------+
| r             | i             |
+---------------+---------------+
| 5             | 1             |
+---------------+---------------+
relation Sum2 has 1 tuple
```

Figure 4.25: The Sum of Complex Numbers

The `funop` blocks of `cplx1` or `cplx2` can be used to calculate the alternating sum of complex numbers. Figure 4.26 shows an alternating sum ordered by `id`.

The values of `aSum` are the sequence: $2-2i$, $(3+2i)-(2-2i)$, $1-(3+2i)+(2-2i)$, and $(-1+i)-1+(3+2i)-(2-2i)$. The usual alternating sum would be $2-2i$, $(2-2i)-(3+2i)$, $(2-2i)-(3+2i)+1$, and $(2-2i)-(3+2i)+1-(-1+i)$. But it is easy to convert the former to the latter using domain algebra.

Computations usable in vertical operations are still normal computations that can be called using the syntax introduced in Section 3.6. Some examples are given in Figure 4.27.

```
>let aSum be fun cplx1(out,in,in) of R order id;
>aSum1 <- [id,aSum] in CNumbers;
>pr aSum1;
+-------------+----------------------+
| id          | aSum                 |
+-------------+----------------------+
| 1           | 27                   |
| 2           | 31                   |
| 3           | 35                   |
| 4           | 39                   |
+-------------+----------------------+
relation aSum1 has 4 tuples
>pr .aSum;
+--------+---------------+---------------+
| .id    | r             | i             |
+--------+---------------+---------------+
| 27     |  2            | -2            |
| 31     |  1            |  4            |
| 35     |  0            | -4            |
| 39     | -1            |  5            |
+--------+---------------+---------------+
relation .aSum has 4 tuples
```

Figure 4.26: The Alternating Sum of Complex Numbers

4.2.5 Summary

Computations have been adapted to enhance the vertical domain algebra. JRelix programmers can now add appropriate user-defined vertical operators as long as the following rules are observed:

- The computation that defines the operation must have exactly three parameters of the same type: two for input and one for output.
- An 'alt' block may be decorated with **redop** if it is to be used in reduction or equivalence reduction. Blocks intended for functional mapping or partial functional mapping must be labelled **funop**. It is the programmer's responsibility to ensure the semantic consistency among blocks and to verify that a **redop** block is truly commutative and associative.
- To call the computation in a vertical operation, simply place the computation's signature after the relevant keyword (**red**, **fun**, **equiv**, or **par**). The signature is the computation's name optionally followed by a comma-delimited, parenthesized list of "**in**" and "**out**". The order of these directional keywords must match that of the in/out parameters of the intended block. By default, the first two parameters are **in**, and the third **out**.

```
>relation cn1(r,i) <- {(1,2)};
>relation cn2(r,i) <- {(1,-2)};
>cplx1(in cn1,in cn2,out cn3);
>pr cn3;
+---------------+---------------+
| r             | i             |
+---------------+---------------+
| 2             | 0             |
+---------------+---------------+
relation cn3 has 1 tuple

>cplx1(in cn1,out cn2',in cn3);
>pr cn2';
+---------------+---------------+
| r             | i             |
+---------------+---------------+
| 1             | -2            |
+---------------+---------------+
relation cn2' has 1 tuple

>relation BigR(R1,R2) <- {({(1,2)},{(1,-2)}),
                          ({(2,4)},{(-2,4)})};
>BigR3 <- [R3] in (BigR ijoin cplx1);
>pr BigR3;
+----------------------+
| R3                   |
+----------------------+
| 58                   |
| 57                   |
+----------------------+
relation BigR3 has 2 tuples
>pr .R3;
+----------+---------------+---------------+
| .id      | r             | i             |
+----------+---------------+---------------+
| ...      | ...           | ...           |
| 57       | 2             | 0             |
| 58       | 0             | 8             |
+----------+---------------+---------------+
relation .R3 has 14 tuples
```

Figure 4.27: Other Uses of the Complex Number Computation

Chapter 5

Implementation of Abstract Data Type

In this and the next chapter, we present the implementation details of the new features as seen in *User's Manual*. Abstract data type forms the main topic of this chapter. We start with a brief overview of the system architecture and the development environment of jRelix, followed by descriptions of components relevant to our implementation. With the background laid out, we discuss in Section 5.2 the implementation of the three aspects of ADT: state, accessor method and modifier method.

5.1 System Overview

5.1.1 Development Environment

JRelix is implemented in Java (JDK 1.2.2 and up). It runs on Windows platform as well as on Unix, as long as a compatible version of the Java run-time environment is installed.

The parser for the language is generated by two tools: JJTree and JavaCC. JavaCC, a java compiler compiler, is a utility written in Java which automatically generates a parser by compiling a high-level grammar specification stored in a text file and converts

it to a Java program which can recognize matches in the grammar. JJTree, on the other hand, is a preprocessor for JavaCC utility that inserts parse tree building actions at various places in the JavaCC source. Figure 5.1 depicts the process of generating the parser file (Parser.java). Please refer to the JavaCC and JJTree documentation in [SDV01] for more details.

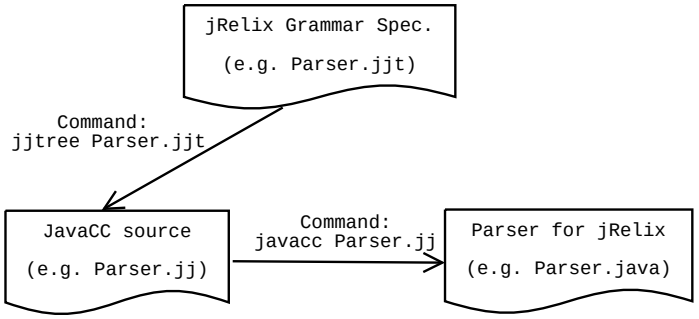

Figure 5.1: Generating the Parser Using JJTree and JavaCC

5.1.2 JRelix Storage Format and Architecture

JRelix Storage Format

A relation is stored as a file with the name of the relation on the hard disk. In the case of a computation, the syntax tree of the computation is saved under a name of the form *CompName.comp* where *CompName* is the computation's name. Event handlers are also stored by their names[1].

The information about all relations and domains (which we call *database metadata*) in a jRelix database is maintained in memory while a user session is in progress and dumped onto the hard disk when the user logs out. On hard disk, the metadata is split into several system files, including *.rel*, *.dom*, *.rd*, and *.expr*. *.expr* is used to store a serialized form of the definition of views and virtual domains in the system. The other three are themselves relations. Table 5.1 describes these system relations. For

[1] An event handler's name contains ':', which causes problems on the Windows platform.

more information on the storage and maintenance of database entities, please refer to [Yua98].

Relation	Domain	Description
.rel	.rel_name	name of a relation
.rel	.tuples	number of tuples in the relation
.rel	.attributes	number of attributes in the relation
.rel	.rvc	is a relation, view or computation
.rel	.sort	number of attributes the relation is sorted on
.dom	.dom_name	name of an attribute
.dom	.type	type of the attribute (see Table 3.1)
.dom	.count	reference count for attribute
.dom	.isState	is a hidden state (*new to this implementation*)
.rd	.rel_name	name of a relation
.rd	.dom_name	name of an attribute
.rd	.position	index of the attribute in the relation

Table 5.1: System Relations

JRelix Architecture

The jRelix database system is composed of three conceptual modules: a *front-end interface*, a *database engine* and a *database maintainer*. The overall architecture of the jRelix system is shown in Figure 5.2 . As its name implies, the front-end interface module serves as an interface between the user and the database engine. At the beginning of each interpretation cycle, it accepts a user command and performs syntax analysis. It then converts these commands into a tree-like structure called a syntax tree, performs error checking and finally invokes appropriate functions in the database engine to execute the necessary operations. At the heart of jRelix, the database engine implements the relational and domain algebras as well as computation. The intermediate and final outputs of the database engine (in the form of relations) is then passed to the database maintainer which updates the database system on the hard-disk as appropriate. The database maintainer is also responsible for loading relations into memory when necessary.

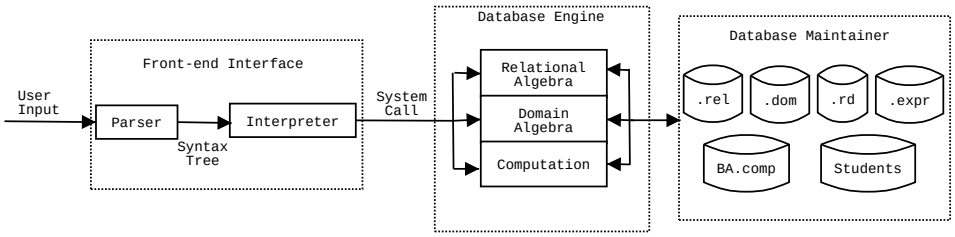

Figure 5.2: JRelix System Architecture

Each of the conceptual modules is made up of several components, or *classes* in Java parlance. Appendix B shows all the classes in the system and their categorization. The following sub-section highlights some of them that are most relevant to our implementation.

5.1.3 Synopses of Selected Components

The implementation of abstract data type and extended domain operations has been built on an existing system. It follows the philosophy of maximizing code reuse. As a result, only one new class has been created. Most new functionalities have been achieved through augmenting and glueing code in the classes to be described shortly. The interested reader may find a summary of all the enhancements by this implementation in Appendix C.

The SimpleNode Class

Every jRelix statement or command input by the user is parsed and transformed into a syntax tree. The syntax tree can be decomposed recursively from top down into a number of sub-trees, each of which represents a jRelix expression, statement or operator. At the extreme end of such decomposition are **node**s . Figure 5.3 shows the syntax tree of the following virtual domain declaration found in Figure 4.7 of the Car Racing example (each circle is a node):

```
let speed be [red nop of curSpeed] in speed';
```

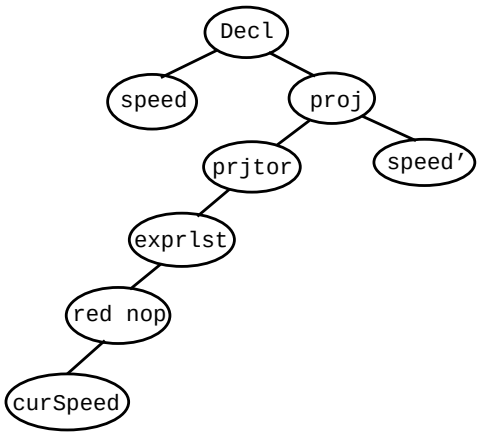

Figure 5.3: Syntax Tree Example

The SimpleNode class desribes a node in the syntax tree. Since a node contains references to both its children and its parent, it is possible to traverse a syntax tree (or part of it) in both ways. The information kept with a node object is frequently used by the **Interpreter** class and other classes. The SimpleNode class is therefore one of the most important classes of jRelix. Table 5.2 gives the most important members and methods of this class.

The Interpreter Class

A single Interpreter object is instantiated and used throughout a user session of jRelix. The role of this object is to take over the syntax tree from the parser, descend the tree in a top-down fashion and dispatch function calls into the database engine based on the type and opcode information of the node at hand. This class also provides methods that perform basic validity check, such as *traverseNode* and *traverseType*. Several methods in this class have been augmented to support ADT and extended domain operations.

The Relation Class

This class implements the relational algebra [Hao98]. Its most frequently accessed public members are (Java type in parentheses):

- *name (String)* The name of the relation.

SimpleNode Members	
Name	*Description*
type*	Code of operation represented by the node
opcode	Sub code of the operation
name	Name of a node of type OP_IDENTIFIER (null otherwise)
info	Value of a node of type OP_CONSTANT; placeholder for an accumulator in vertical operations (null otherwise)
bits	Special information on a node
SimpleNode Methods	
Name	*Description*
jjtCreateNode	Creates a SimpleNode object
jjtGetParent	Returns the reference to the parent node
jjtSetParent	Sets the reference the parent node
jjtGetChild	Returns the child node with a given index
jjtGetNumChildren	Returns the child node count
jjtAddChild	Appends a child node to the parent
jjtReplace	Replaces this node with another
jjtRemoveChild	Removes a child with a given index
setBit	Set the bits field to a specific value
isBitSet	Check if a certain bit is set
*Type and opcode are defined in Constants.java	

Table 5.2: The SimpleNode Class

- *rvc (int)* RELATION, VIEW or COMPUTATION.
- *numtuples (int)* The tuple count.
- *numattrs (int)* The attribute count.
- *tree (SimpleNode)* The syntax tree for a view or computation.
- *myEnv (Environment)* The environment the relation is declared in.
- *domains (Domains[])* The array of Domain objects; each element holds the information about a domain[2].
- *data (Object[])* The data held in a relation, one column per element; each element is itself an array.

The public methods in the Relation class are readily available in the subclass Computation for the purpose of relational evaluations. No change has been made in this implementation.

The Domain Class

An object of the Domain class represents a domain, virtual or not. Its public members are:

- *name (String)* The name of the domain.
- *type (int)* The type code of the domain.
- *numref (int)* The reference count of the domain (i.e. how many relations are defined on this domain).
- *tree (SimpleNode)* The syntax tree of a virtual domain.

New members have been added to the class to implement new features. These will be described in later Section 5.2.

[2]The difference between an attribute and a domain is subtle. Attribute can be thought of the name of a column in a relation, while a domain is a set of values allowed in a column. In this thesis, the two are used interchangeably

The Actualizer Class

The Actualizer class handles the actualization of virtual domains. A virtual domain declaration leads to the creation of a Domain object[3]. To materialize this virtual domain in a relation, an Actualizer object must be created and initialized. The initialization process involves a series of validity checks to ensure that the virtual domain is "actualizable". Specifically, it detects cases where a virtual domain is defined on non-existing actual domain(s), is recursively defined on itself, or is impossible because the syntax tree of its declaration contains semantic errors. Once the validity checks are passed, the programmer needs to explicitly invoke the *actualize()* method of the Actualizer object for the actualization process to be complete. In jRelix, a virtual domain is actualized with a tuple-by-tuple approach.

Here are some important methods which were first implemented by [Yua98] and [Kan01].

- *buildTree()* This is a self-recursive routine for the virtual tree building process. It accepts a SimpleNode object representing the syntax tree of a virtual domain. Besides the validity checks mentioned previously, it also performs tree expansion to make sure that all identifier nodes in the final syntax tree are actual domains from the source relation. For instance, if we have:

  ```
  let x be A + B;   let y be C;   let z be x * y;
  ```

  The virtual domain tree of `z` will be expanded to represent:

  ```
  (A + B) * C;
  ```

  A third task that the *buildTree()* method accomplishes is tree truncation in the case of virtual domains defined on vertical operations such as reduction or functional mapping. For example, given the definition:

  ```
  let A be red + of B * C;
  ```

[3]In the case of a relation typed domain, a Relation object containing the dot relation is created and registered into the environment as well

The subtree corresponding to `B*C` will be truncated and actualized first, before the reduction can be evaluated.

- *the "cell methods"* If we think of a virtual domain in a relation as a column of empty cells, actualizing the virtual domain is like filling in the cells with the values calculated from the data in the tuple according to the rule given by the definition of the domain. There are a number of methods in the Actualizer class for cell data calculation, as listed in Table 5.3. We will refer to them as the "cell methods", a term borrowed from [Yua98]. A cell method may invoke other cell methods or itself to evaluate a subtree. When cell methods are used for horizontal domain operations, the calculated data isdirectly put into the corresponding cell. When they are invoked by methods dealing with vertical operations, the data is combined with an accumulator value first and then stored back to the accumulator.

Method	JRelix Type of Domain to Actualize
actBoolCell	BOOLEAN
actIntCell	SHORT, INTEGER
actLongCell	LONG
actDoubleCell	FLOAT, DOUBLE
actStrCell	STRING
actRelCell	IDLIST

Table 5.3: Cell Methods in the Actualizer Class

- *the vertical actualize methods* These are methods involved in actualizing virtual domains defined on vertical operations. Reduction is actualized in *actualizing()*, equivalence reduction in *actualizeEquiv()*, functional mapping in *actualizeFun()*, and finally, partial functional mapping in *actualizeParFun()*. Algorithms of vertical domain actualization will be given in the next Chapter where we discuss the extension to allow user-defined operations.

The implementation of ADT and extended vertical domain operation has brought quite a few additions to the Actualizer class.

The Environment Class

An environment is an object that holds bindings of variables to their specific information (such as value or type). In jRelix, an object of the Environment class maintains the bindings for various types of variables, including relations, domains, state variables, parameters and local variables. The last three are unique to computations. An Environment object stores these bindings in hashtables for fast lookup. It also exposes public methods to add or delete a binding, or to look up the information of a variable.

The specific information about a state, a parameter, or a local variable is encapsulated in objects of class **StateInfo**, **ParamInfo**, or **LocalInfo**, respectively. These[4] are all subclasses of **IDInfo**, an abstract base class used to name the return types of the general lookup routines. For example, the method *Environnment.lookup()* is declared to return an object of type IDinfo, but it may actually return an instance of StateInfo at run time. Such implementation takes advantage of the late-binding facility of the Java language, thus avoiding repeated coding for different subtypes. The subtypes are distinguished by their *kind* member. They also contain information such as the type, value and position of the variable.

Computations are bound in their declaration environment. Upon invocation, a new temorary environment is created to hold bindings for the parameters, local and state variables. The code of the computation is then evaluated with regard to this new environment and its parent (the declaration environment). Upon exit from the computation, the temporary environment is discarded. The Environment class is therefore particularly relevant to the implementation of ADT, a special form of computation.

The NREnvironment Class

In jRelix, each level of a deeply nested relation may itself consist of nested relations. It is possible to write domain algebra expressions involving domains from different levels of such a nested relation. To relieve the system programmer of the burden of remembering

[4] They were created by [Bak98], but underwent significant changes in this implementation.

these levels while implementing virtual domain actualization, the NREnvironment class was invented.

An NREnvironment object is created for each level of a deeply nested relation as the relation's hierarchy is descended in the actualization process. It holds the bindings for the domains found at its own level. The current row number of the relational domain being scanned is also kept track of. The domain algebra expression to be actualized is then evaluated in the context of the NREnvironments as well as the global Environment that holds the bindings for everything else. The NREnvironment objects are attached to the Environment object and are always checked first for bindings upon a look-up, before any other hash table in the same Environment.

Instantiation of an ADT often results in deeply nested relations. With a little adjustment of the NREnvironment class, implementation of ADT facilities becomes much more manageable.

The Constants Interface

This Java interface class defines all the operation and type codes used by jRelix. A node in a syntax tree always has an associated *type* and an *opcode*. For example, a node representing the identifier **A** (the name of a relation) will have both of these set as OP_IDENTIFIER, which is defined to be the value `230` in the Constants interface. Whenever a new operation is added to the language syntax, the Constants class needs to be updated. We will encounter such an addition in the next chapter.

5.2 Implementation of ADT

An abstract data type features encapsulation and information hiding. In jRelix, encapsulation is achieved by defining an ADT using the same syntax as a computation with state. That is, the information to hide is represented by the state variable(s), while the computations defined inside the ADT implement the operations on objects of the type. JRelix allows the programmer to instantiate by joining the ADT to an appropri-

ate relation. This removes the need for a *new* operator, which would be cumbersome in the database context of possibly millions of instantiations. JRelix employs special implementation to ensure that states are only accessible through the operations of an ADT, thus information hiding.

[Bak98] first described stateful computations and proposed a static environment model in which states and associated methods can be exported to become computation typed (COMP) domains of a relation. However, the issue of information hiding was neglected and no working implementation of stateful computation was available. Later, [Sun00] gave a short section on how computation calls may be invoked in *update change* operations. The example involves a single assignment statement in the computation body and all variables are integers.

To provide a sound foundation for the implementation of ADT, a number of computation related enhancements have been made, including:

- a flexible pass-by-name parameter passing mechanism,
- a broader range of statements that work in a computation block, and
- an updated set of rules for the calculating a block type.

Details of these improvements are given in Section 5.2.1.

The rest of this chapter covers the implementation of support for ADT. Among other things, the system has been modified to handle states of any type (with the exception of computation[5]) and provide appropriate storage for them in memory. For the sake of hiding information, states are made inaccessible to normal relational and domain operations. The algorithms for virtual domain actualization and updates have been augmented to provide controlled access and modification to states.

Although aimed at implementing ADT, the enhancements and modifications mentioned above also lead to non-ADT related features such as packages (see examples given in Section 3.6.3) and level lifting in the case of unary singleton relations (see examples

[5]We have not found a good reason for a state to be a computation; therefore it is not supported as of this writing

in Section 4.1 on the use of ADT). The latter is available for use even outside of the context of computations.

5.2.1 General Enhancements Related to Computation

In this section, we first summarize the implementation of computation given by [Bak98], hereafter referred to as the "previous implementation". Next we describe some enhancements that make computations more flexible and powerful.

Summary of Previous Implementation

The three classes most closely related to computation are Computation, CompBlock and EvalExpr. The Computation class contains the following public methods (parameters are omitted):

- *Computation()* for instantiating a Computation object
- *applyInOut(), applySelect(), applyIjoin()* for invoking a top-level computation using the first three forms of syntax as described in Section 3.6.1
- *sc()* for use with the **sc** command (see Section 3.6.4)
- *print()* for use with the **pr** command (see Section 3.6.4)

A reference to the environment in which a computation is declared is stored in the *declEnv* member. The computation's 'alt' blocks, represented by CompBlock objects, are referenced by the *blocks* array. The *tree* member holds a reference to the computation declaration. All these references are intialized when a Computation object is instantiated.

The 'alt' blocks of a computation are distinguished from each other by their type. When a computation is created, the types of its blocks are calculated and stored with the CompBlock objects. This way, re-calculation of the type upon computation invocation is avoided. The type of a block is calculated based on the input/output status of computation parameters upon their first use, as follows:

If the computation has n parameters, the type of a block is represented by an n-digit base-2 number.The most significant digit of that number corresponds to the last parameter (the rightmost in the parentheses after the computation name), and the least significant digit corresponds to the first. Each digit takes on a value of 1 if the corresponding parameter is input, or 0 if otherwise.

Consider the computation given in Figure 5.4. It has two blocks. The type of the first block is 10_2 (decimal 2), because `r`, the second parameter, is used as input, while `area` is the output. The second block has exactly the opposite situation, therefore its block type is 01_2 (decimal 1).

```
>let pi be 3.14;
>domain r, area float;
>comp CircArea(area, r) is
 { area <- pi * r * r;}
 alt
 { r <- sqrt(area / pi);};
```

Figure 5.4: Computation `CircArea`

The *findBlockType()* method of the CompBlock class implements the rules that determine the input/output status of a parameter.

Expressions encountered while executing a computation block are evaluated by the methods in the EvalExpr class. One method each has been provided for expressions of the following jRelix types: boolean, short, integer, long, float, double, and string. Relational expression evaluation is accomplished via calls to methods implemented in the Relation class.

Other classes that support the implementation of computation include IDInfo, StateInfo, ParamInfo and LocalInfo, as mentioned in Section 5.1.3. They are mainly used by the Environment class to abstract over the details of different variables associated with a computation.

Enhancement 1: Pass-by-name Parameter Passing in Natural Join Syntax

With the previous implementation, if a programmer joins a computation with a relation and expects the relation to pass input values to the computation, he or she has no choice but to name the input domain in the relation according to the input parameter. For example, given the declaration of `CircArea(area, r)` in Figure 5.4, we can use the relation `A(r)` or `B(area)` to invoke it, but not `C(radius)`, nor `D(AREA)`[6].

Such restriction has been eliminated by using the pass-by-name parameter passing method for our implementation. With this method, whenever a computation is invoked, its syntax tree is traversed and the formal parameters are substituted by their corresponding actual parameters in all occurrences. Back to our `CircArea(area, r)` computation: if it is invoked with `C(radius)`, the syntax tree of the computation will be transformed as if the declaration was:

```
>comp CircArea(area, radius) is
 { area <- pi * radius * radius;}
 alt
 { radius <- sqrt(area / pi);};
```

The implementation of the pass-by-name mechnism is given by the modified *applyIjoin* method and a number of newly added methods, as listed in Table 5.4. The step number in the second column refers to the number listed in the pseudo code for *applyIjoin* (see Figure 5.5).

Steps in the pseudo-code without a '*' have been inherited from [Bak98], so we will skip them here. Step 3 in the new algorithm has two possible execution paths. If the ijoin does not involve explicitly named domains (i.e. `domsr` and `domsc` are empty), the common domains between the relation and the computation will be used as the input domains. On the other hand, when there is no common domain between the invoking relation and the computation (such as when `CircArea(area, r)` is joined with `C(radius)`), we can specify the join domains from the computation in `domsc` and those from the relation in `domsr`. In our example, `domsc` contains `r` and `domsr` contains `radius`. Join domains in this case are given by either `domsr` or `domsc`, but we still

[6] JRelix is case sensitive

```
Relation applyIjoin(Relation rel, Domain[] domsc,
   Domain[] domsr, String filename, Environment callEnv)
{
 1. Create an Environment object (env) and set its parent
    to be the declaration environment of this computation.
 2. Back up the original domains of the computation into
    originalDomains.
 3*. Determine join domains (joinAttrs) and extra domains
    (extraAttrs).
 4*. Transform the syntax tree of the computation.
    4.1. Get a copy of the syntax tree (tempTree).
    4.2. if named join domains (domsc, domsr) are specified
         then
         4.2.1. For each domain in domsc (computation domain),
                substitute its occurrences in tempTree for
                the corresponding domain in domsr (relation domain).
         4.2.2. Re-generate the CompBlock array (blocks) from
                tempTree.
 5. Register the parameters, local variables and states
    into the current environment (env).
 6. Find the computation block based on the join domains.
 7. Load the data of the input relation (rel) into memory.
 8. For each tuple of the input relation do
    8.1. Execute the statements in the selected block with
         reference to the current environment (env).
    8.2. Append the output to the relation.
 9. Project rel on all its domains to eliminate duplicates.
 10. Restore the original domains from originalDomains.
 11*. Restore the original CompBlock array.
 12. Detach the Environment (env) for garbage collection.
}
```

Figure 5.5: Pseudo-code for *applyIjoin()*

need to find out what "extra domains" are — these domains do not participate in the computation, but they need to be in the result relation. The method *extraDomains2()* is provided for this purpose. The example we looked at would not produce any extra domains. Had we invoked the computation with `E(radius, index)` instead, `index` would be regarded as extra.

The code that fills in step 4 performs a textual substitution of the names in `domsc` for the names in `domsr` position by position. For instance, all occurrences of **r** in computation `CircArea(area, r)` will be replaced by **radius** in our example. Then the CompBlock objects referenced by the `blocks` member of the Computation object are regenerated based on the transformed syntax tree. After these steps, the input parameters of the computation block to be run exactly match the input domains provided by the invoking relation. Further more, all the existing code for executing the 'alt' block in *applyIjoin* still works, unaware of the use of a different join syntax.

The only extra step left is to revert the input parameter names in the CompBlock objects to their original at the end of the method. This is necessary to ensure the success of subsequent invocations on the same computation.

Method	**Step**	**Description**
extraDomains2	3	returns the union of domains of the computation and the invoking relation except the join domains
renameParamsInTree	4	replaces the input parameter name with the actual argument name in the computation's syntax tree
replaceJoinAttrs	4	renames the join domains according to the actual argument names
reGenerateBlocks	4	generates the CompBlock objects from the renamed syntax tree of the computation; the domain names of the computation are modified accordingly

Table 5.4: New Methods to Support Flexible Pass-by-name Mechanism

Enhancement 2: Statements and Commands in Computation Blocks

In the previous implementation, the only statement type allowed in a computation block was the assignment statement. This is no longer the case. All jRelix statements and commands that have been implemented at the top-level can now also appear inside a computation block.

The central place for handling different types of statements and commands is the *runSingleStmt()* method of the Computation class. A sketch of the code in this method is given in Figure 5.6. (Note that not all statement types are shown. For a complete listing of the types please refer to Appendix A).

The *runSingleStmt()* method is invoked each time a statement of a computation block is to be executed. It accepts two parameters, one is the syntax tree of the statement to be executed (`stmt`), and the other is the Environment object that contains all the bindings known to the computation (`env`). Based on the type of the statement, specific actions are taken subsequently. These actions can be categorized as follows:

- *call runSingleStmt() recursively* The case of OP_CONDITIONAL (if-then-else) is in this category. Whenever a statement contains sub-statements, the sub-statements are evaluated first using the same method.
- *call a delegate method in the Interpreter class* Case OP_COMMAND deals with commands in computations. Instead of re-implementing the logic for executing different types of commands, it simply calls the *executeCompCommand()* method in Interpreter. This method in turn invokes *executeCommand()* which was implemented by [Hao98]. *executeCompCommand()* is just one of several delegate methods added by this implementation. They are simple public methods used to direct calls from methods in the Computation class into the private methods of the Interpreter class. Their names all start with "executeComp".
- *call a public method in the Interpreter class* The purpose of this approach is to promote code-reuse, just as in the case of calling a delegate method. Methods such as *evaluateTLExpress()* in the Interpreter class are public, thus they can be accessed freely from other classes. To obtain the relation resulting from the

```
void runSingleStmt(SimpleNode stmt, Environment env)
{
 ...
 switch (stmt.type)
 {
  case OP_ASSIGNMENT:
   SimpleNode left = (SimpleNode) stmt.jjtGetChild(0);
   SimpleNode expr = (SimpleNode) stmt.jjtGetChild(1);
   IDInfo id = env.lookup(left.name, true);
   //code to find out the jRelix type of id and assign it to stmtType
   switch (stmtType)
   {
    case SHORT:
     short shortResult = EvalExpr.evalShortExpr(expr, env);
     //code to store shortResult in appropriate place
     //based on whether id.kind is PARAMETER, LOCAL, or STATE
    case IDLIST:
     ...
     transformExpr(expr, env);
     ...
     Relation relResult =
            interpreter.evaluateTLExpression(expr, ..., env);
     ...
     //code to store relResult in appropriate place
     //based on id.kind
   }
  case OP_STATEMENT:
   switch (stmt.opcode)
   {
    case OP_CONDITIONAL:
     ...
     if (EvalExpr.evalBooleanExpr(if_node, env))
      runSingleStmt(then_node, env);
     else ...
    case ...
   }
   ...
  case OP_COMMAND:
   interpreter.executeCompCommand(stmt, env);
   ...
  case OP_UPDATE:
   ...
 }
}
```

Figure 5.6: Pseudo-code for *runSingleStmt()*

right hand side of an assignment statement, we simply let *evaluateTLExpression()* handle it. There is a tricky issue here. Since the methods in the Interpreter class have no notion of parameters or local variables that are bound in the environment inside a computation, we cannot simply hand over the expression encountered in a computation as is. Instead, we must first transform the expression so that all references to computation specific variables are replaced with the values of these variables (relational variables are treated differently, for they are still recognized in a non-computation environment). The recursive method *transformExpr()* is provided for this purpose. A similar method, *transformStmt()*, is also available for transforming a statement.

- *call a method in the EvalExpr class* This approach is taken when there is a need to evaluate an expression of atomic type, such as INTEGER or FLOAT. EvalExpr is the only class in the jRelix implementation that provides methods for evaluating atomic types. Therefore it is frequently used by the implementation of computation, the only place where atomic typed variables are allowed on the left hand side of expressions. The enhancement made to the EvalExpr class includes support for the evaluation of state varriables and of built-in functions (OP_FUNCTION)[7].
- *reproduce some of the code in the Interpreter class* This is the most extreme case. The only such case is OP_UPDATE. As the logic for handling updates was implemented in the Interpreter class itself (see [Sun00]) and coupled tightly with the method *evaluateTLExpression()*, there is no single method to call to handle just updates. In addition, the environment inside a computation is quite different from that of the top-level. It turns out that reproducing some of the flow-control code in *evaluateTLExpression()* and adjusting it for use within the environment of a computation is the most convenient way in this case.

[7] The support for built-in functions such as abs() or ceil() is based on corresponding methods offered by the Java Math package. Built-in functions are also supported at the top-level. Relevant code can be found in the *act*Cell* methods of the Actualizer class. Due to space limit, we will not discuss them any further.

In the code sketch for the *runSingleStmt()* method, we also mentioned that the result of evaluation will be put into appropriate places based on whether the destination variable is a parameter, a local variable, or a state variable. When a computation is invoked, an output parameter becomes one of the domains of the resulting relation. Therefore, we can refer to the output parameter by its position in the result relation. At any time of interpretation, we also keep track of the row of the relation being affected by the computation. Thus the result of evaluating an output parameter can always to put directly into its place in the result relation. The case of a local variable is even simpler. Since a local variable is valid only within the scope of the computation block for which it is defined, there is no need for its value to persist in the result relation. Storing its value in the LocalInfo object is sufficient. The storage of state variables will be discussed in Section 5.2.2. The basic mechanism is similar to that of the parameters.

Enhancement 3: Calculating the Block Type

It comes as no surprise that the rules to determine the type of a computation block in the *Computation.findBlockType()* method needs adjustment as new statement types are introduced into the computation block. We now give the updated rules for jRelix statements and commands that affect the block type, as follows (note that these rules apply only when a variable is first used in a block):

Assignment ID "<-" EXPR

Input: all variables in EXPR

Output: ID

Exception: Variables in the projector list of a selection or T-select expression are excluded from consideration[8].

	Statement	Input	Output
Example:	R <- [A] in S ijoin T	S, T	R
	R <- where A = 4 in S	S, A (only if A is a parameter)	R

[8]This is true even if they appear in the parameter list, as of this implementation

Incremental Assignment ID "<+" EXPR

Input: all variables in the statement

Output: none

Exception: same as those for Assignment

Example:

Statement	Input	Output
R <+ [A] in S ijoin T	S, T, R	
R <+ where A = 4 in S	S, R, A	
	(only if A is a parameter)	

View Declaration ID "is" EXPR

Input: all variables in EXPR

Output: ID

Exception: same as those for Assignment

Example:

Statement	Input	Output
R is [A] in S ijoin T	S, T	R
R is where A = 4 in S	S, A	R
	(only if A is a parameter)	

Update Add/Delete "update" ID "add"|"delete" EXPR

Input: ID and all variables in EXPR

Output: none

Exception: same as those for Assignment

Example:

Statement	Input	Output
update S add T ijoin R	S, T, R	
update S add ([A, B] in T ijoin R)	S, T, R	

Update Change "update" ID "change" STMTLIST "using" JOINOP EXPR

Input: ID, all inputs in STMTLIST and all variables in EXPR

Output: none

Exception: Variables in a projector list are ignored; outputs of the STMTLIST are ignored as well, as they are considered attributes of ID

Example:

Statement	Input	Output
update S change A <- C + 3 using ijoin T	S, C, T	

Conditional Statement "if" EXPR "then" STMT1 "else" STMT2

Input: all variables in EXPR, all inputs in STMT1 and STMT2

Output: all outputs in STMT1 and STMT2

Exception: Variables in a projector list are ignored

Example:

Statement	Input	Output
if [] in R then S <- T else S <- U	R, T, U	S

Computation Call COMPNAME "(" PARAMLIST ")"

Input: all variables in PARAMLIST

Output: none

Example:

Statement	Input	Output
CirArea(A)	A	

Selected Commands "pr"|"sr"|"dr"|"sc" EXPR

Input: all variables in EXP

Output: none

Example:

Statement	Input	Output
sr R	R	

Note that if a nested-level computation is called inside the block of its declaration, the system will run *findBlockType()* on this computation and propagate the result up to its enclosing computation block. [Bak98] provides an example in this case.

5.2.2 Implementation of State

The keyword "state" is used to declare a special variable in a computation, which we refer to as a *state variable*, or a *state* for short. JRelix uses states to represent hidden information inside an ADT. States are special in that they retain their values between computation calls, which distinguish them from local variables. In addition, states are

hidden, in the sense that they can only be accessed and manipulated by code from within the same computation for which they are defined. For instance, the state `_curVal` of the `Counter` computation (Figure 3.37) is only accessible to the statements in the two 'alt' blocks of `Counter`. As another example, states of an ADT can only be accessed or modified by the ADT's methods. We begin the topic of implementing state, with with a discussion of the storage of state variables in main memory. Next we describe the process of state creation. A related issue, state persistence, is discussed as well. Following that, we present, from a system programmer's point of view, methods to retrieve and modify the value of a state. The implementation of state hiding is described at the end of this sub-section.

Storage of a State

Depending on what syntax is used for a computation invocation, a state may be stored in two different ways in main memory. In the case of instantiating objects of an ADT using an ijoin syntax, all states of the ADT are exported as domains of the resulting relation, thus no special data structure is needed to hold the values of the states. Like any normal domain, the value of a state is stored in the `data` member of the Relation object that represents the result of ijoin. The relation `TopRacers` as shown in Figure 4.6 illustrates this point.

When a computation is invoked by a stand-alone call or a select expression, the situation is quite different. The state of the computation is not exported to the result relation (see the `Counter` example in Figure 3.38). As a result, the computation itself must "remember" the value of its state. For this purpose, we implemented the **StateInfo** class[9], as shown in Figure 5.7. The diagram in Figure 5.7 depicts the inheritance relationship among three classes: **IDInfo**, **StateInfo_2002**, and **StateInfo**. It also shows the members and methods of each class. We already covered the IDInfo class in

[9][Bak98] also mentioned this class, but it did not provide storage for a state; therefore this implementation added the StateInfo_2002 class. We could have modified the StateInfo class directly, but it was a design choice not to do so.

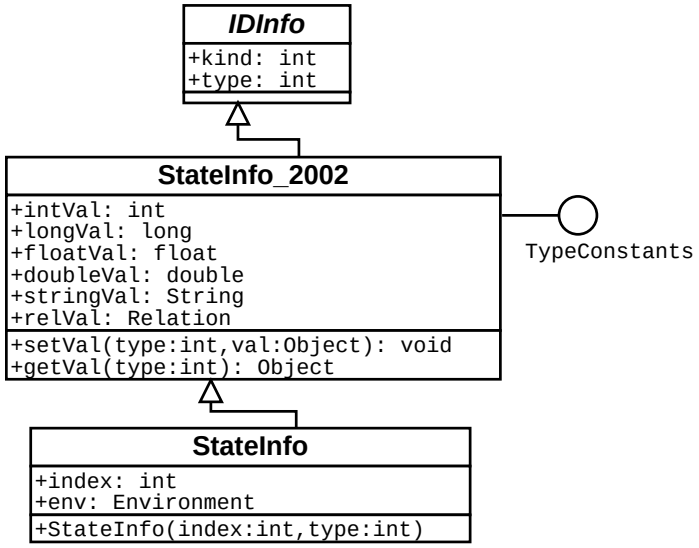

Figure 5.7: Class Diagram of StateInfo

Section 5.1.3. StateInfo_2002 is new to the system. Its purpose is to provide storage for various types of state variables and public methods for accessing and modifying the value of the state. The StateInfo class extends StateInfo_2002 by adding its own data members such as `index` and `env`. Since all data members of StateInfo_2002 are public, they are readily available to the subclass. The meaning of `index` and `env` will become clear in the next sub-sections.

In summary, states are stored like domains when the ijoin syntax is used to invoke a stateful computation. In this case many values of the same state co-exist, as long as they belong to different tuples of the ijoin result. Any other types of invocation result in states being stored in StateInfo objects. Consequently there can be only one StateInfo object, and thus one value, for any given state in this case.

Computation Initialization

The declaration of a computation leads to the instantiation of a Computation object. The Computation class constructor is called by the interpreter and performs the following tasks:

1. Store a reference to the declaration environment in `declEnv`.

2. Store the syntax tree of the computation declaration in `tree`.
3. Create a new Environment (`stateEnv`) for handling persistent states.
4. Parse the syntax tree to deduce parameters, local variables and states; store such information in a temporary Environment object.
5. Generate the CompBlock array `blocks` and calculate the block types at the same time, based on the information obtained in the previous step; store the block type of an 'alt' block in its corresponding CompBlock object.

The interpreter then adds an entry for the Computation object to its declaration environment, so that it can be looked up by name. Up until this point, no storage has been allocated for states or other variables in memory. It is only when the computation is invoked that these variables come into existence.

State Creation and Persistence

We now discuss how states are created and made persistent between invocations in the case of stand-alone computation calls and invocation using a select expression. The StateInfo object for a state is created the first time the computation is invoked. This object will be used to hold the value of the state. It is then added to both the environment for computation evaluation and the special environment, `stateEnv`. Every Computation object has an associated `stateEnv` which holds references to the StateInfo objects of that particular computation. Therefore, when further invocations of the same computation occur, there is no need to re-create StateInfo objects, as they are already in `stateEnv`. In addition, the value of a state from the most recent invocation can always be found by looking up the relevant StateInfo object from `stateEnv`. This is how a state persists between computation calls.

The creation of states upon an ijoin invocation is exactly the same as that described in the previous case. Only in this case, the actual values of states are stored in the resulting relation, not in the StateInfo objects. Since data of a relation is persistent even on secondary storage, the issue of state persistency is trivial.

The algorithms described above are coded in the *initializeEnvironment()* method, as follows:

1. Extract components such as computation name, parameter list etc. from the syntax tree (retrieved from the `tree` member of the Computation object).
2. Process the parameter list: create a ParamInfo object for each parameter and add bindings for them to the environment being initialized. The type and index[10] of each parameter is recorded with its ParamInfo object.
3. Process local/state variable declaration:

   - If this is a local variable declaration, create a LocalInfo object `li` and add a binding for it to the environment being initialized; then,
     - If the declared type is IDLIST,
       (1) set `li.relVal` to be a new empty relation,
       (2) add a binding for this new relation to the environment.
     - If the declared type is an atomic one,
       (1) create a new domain with the name and type of the local variable,
       (2) add a binding for this new domain to the environemnt.
   - If this is a state variable declaration,
     (a) Look up the state by name in `stateEnv`,
       - If no binding is found, this must be the first time the state is processed, then,
         (1) create a StateInfo object (`si`), set its type and index[11],
         (2) add a binding for `si` to both the environment and `stateEnv`,
         (3) if the type of the state is IDLIST, set `si.relVal` to a new empty relation and add the new relation to the environment; otherwise, create a new domain with the name and type of the state and add this new domain to the environment.

[10] The first parameter on the left has an index value of 0

[11] The first state encountered has an index value of n, if there are n parameters in the computation

- If a binding is found (`oldSi`), then,

  (1) add the existing StateInfo object (`oldSi`) to the environment,

  (2) if the type of the state is IDLIST, add `oldSi.relVal` to the environment.

(b) Export the state as a domain to the caller's Environment, if it is not already there. Set the `isStateDom` flag of the Domain object to 1.

The pseudo code for *initializeEnvironment()* also shows that support for relation (i.e. IDLIST) typed local and state variables has been built in as of this implementation. The last step of the code (3.b) sets `isStateDom` to 1 for the exported state. This step is important for the ijoin invocation, as this is how the environment holding the result relation becomes aware of the states. We will present the rest of the implementation details of hidden states in *Hiding States* on page 109.

Accessing and Modifying States

Modification to a state typically occurs when executing an assignment statement in a computation block where the left-hand side is a state variable. The first case (OP_ASSIGNMENT) in the code sketch given for *runSingleStmt()* on page 99 illustrates the first three steps of state modification:

1. Look up the IDInfo object that represents the variable on the left-hand side of the assignment operator. Note this object is actually an instance of one of the sub-types of IDInfo (ParamInfo, StateInfo, LocalInfo, etc.).
2. Check the `type` field of the IDInfo object.
3. Call an appropriate method in the EvalExpr class to evaluate the right-hand side of the assignement. For example, if `type` is INTEGER, the method *evalIntExpr()* should be called.

The next step is to place the result of evaluation in the storage for the state. There are two possibilities:

- If the computation was invoked using an ijoin syntax, the new value of the state should go to a slot in the `data` array of the result relation. The exact position

of the slot is determined by the current row number of the relation while the column number is the index stored in the StateInfo object. Where do we get the row number? It can be found in the `row` field of the current environment. Not coincidentally, we stored a pointer to the current environment in the StateInfo object when it was first created. In fact, any StateInfo object and its hosting environment cross reference each other. And this is true of objects of all sub-types derived from IDInfo.

- If the computation was invoked using other types of syntax, we simply store the new value within the StateInfo object itself.

Accessing a state for its value is normally encountered in the EvalExpr class. All it takes is to look up the StateInfo object and retrieve the value from either the object itself or the data of the result relation (in the case of an ijoin invocation).

Hiding States

States are meant to be internal to the computation in which they are defined. In the case of a stand-alone invocation or an invocation using a select expression, the states are not exported to the calling environment; therefore they are always invisible except in the computation's 'alt' blocks. An ijoin invocation, on the other hand, does export the states to the result relation. Unless some special actions are taken, the states will be accessible like any other normal domains. These special actions form the topic of this sub-section.

A new data member `isStateDom` has been added to the Domain class. For an exported state, its value is 1. Like other data members of this class (name, type and numref), its value is saved in the system file *.dom* when a jRelix session finishes and restored upon the start of a new session in the same database. Therefore, states are recognized even across sessions.

To prevent states from being accessed directly, we first consider where a normal domain can appear. Table 5.5 gives a summary of such places. In all these cases, an error message should be issued if a state is encountered.

Usage	**Example**	**Handling Method**
In projector list	[A] in R	*ExpressionListToDomains*
In select predicate	where A in R	*evaluateSelect*
	where (A - 1) > 0 in R	*traverseNode*
In assignment	R1 [A1 <- A2] R2	*IDListTodomains*
		ExpressionListToDomains
In join expression	R1 [A1:ijoin:A2] R2	*ExpressionListToDomains*
In virtual domain declaration	let A be red + of B	*traverseNode*
In relation declaration	relation R(A)	*RelationalDeclaration*
In domain declaration	domain B (A)	*executeDeclaration*
	domain B comp(A)	
A stands for domain; all methods are in the Interpreter class		

Table 5.5: Uses of Domains in jRelix

We already know that when a state is exported, a hint is given to the destination environment — its `isStateDom` field is set to 1. Therefore, the solution for hiding states is quite straight-forward: In the methods (column 3) that handle the situations listed in Table 5.5, we perform a check on the `isStateDom` field of a Domain object and throw an exception if its value is 1. The exception will result in the system aborting the attempt to access states illegally.

5.2.3 Implementation of Accessor Method

We saw how states are accessed and manipulated in main memory in Section 5.2.2. However, an application programmer making use of an ADT (a special form of stateful computation) would have to rely on methods provided by the ADT to gain indirect access to the state. Examples of using such methods were given in Chapter 4. We distinguish between two kinds of ADT methods in general: *accessor methods* and *modifier methods*. The former can be any purely functional computation which does not modify states, although in most cases an accessor method simply returns the current value of a state. The latter are non-functional and, as the name suggests, are intended to change the value of states. Therefore a programmer is obliged to invoke a modifier method only within an update statement. This section presents the implementation of the accessor

method mechanism in ADT.

Surrogate Based Representation and CompTable

As the examples in Chapter 4 illustrate, a public method (i.e. a method whose name appears in the parameter list) of an ADT is exported via an ijoin operation between the ADT and a relation. The method becomes a computation typed domain of the result relation, and is stored as a surrogate in the column titled with its name. Internally, the method is still represented by a Computation object. To link a surrogate to the Computation object it stands for, we use the CompTable object in the global environment. The CompTable class in this implementation has been augmented such that the mapping between a surrogate and a Computation object is two-way. That is, we can look up the Computation object by giving its surrogate. Or, we can give the name of a computation and get its surrogate.

The contents of the CompTable are dumped to the system file *.expr* upon exit. Therefore all information on the exported methods of an ADT is retained across jRelix sessions.

Transforming a Unary Singleton Relation

When computation was first introduced into Aldat, it was designed to be invoked at the top-level, i.e., the result of a computation invocation is always a relation. This is still true in this implementation. However, when it comes to using an accessor method, this principle complicates our life a little bit. In the example we gave in Figure 4.7, had we skipped the declaration of `speed` and plunged into the following statement instead:

```
AllSpeeds is [name, speed'] in TopRacers;
```

we would have obtained `AllSpeeds` as a nested relation, with `speed'` being a relation typed domain. The values stored in the column headed by `speed'` would have been surrogates that point to the entries in the dot relation `.speed'`. Another ijoin would be needed to produce the neat result as seen in the given example.

There are two special properties of an accessor method such as `GETSPEED[]` that we can exploit to avoid the complication. First, such method has only one parameter, thus the result relation has only one domain (i.e., unary). Second, the method works on one state at a time, which implies that the result is a singleton. Intuitively, we can just take the value of the tuple to be the value of the state. Although accessor methods can behave differently, what we have just described is by far the most common case. For this reason, we have implemented the machinery to lift the level of the tuple in a unary singleton relation, hereafter referred to as "level-lifting". It actually consists of two mechanisms: one to select a value among a column of data via a vertical `nop` operation; and the other to lift the level of the selected value through anonymity.

Vertical domain operations (reduction, equivalence reduction, functional mapping, and partial functional mapping) are handled in the Actualizer class. Although differerent in several ways, the actualization processes for these operations share some common attributes. They all use an accumulator to store intermediate results and they invoke the horizontal "cell methods" to calculate the new value of the accumulator based on its old value and the actual value of the virtual domain for the current tuple. To make the `nop`[12] operator available in vertical domain algebra, we just modify the "cell methods" so that they treat `nop` the same way as they do operators like + or max. Conceptually, the algorithm for `red nop` goes like this (note that net effect of such an implementation is to pick a value in the column pseudo-randomly):

1. Evaluate the virtual domain for the first tuple. Store the result in accumulator.
2. Evaluate the virtual domain for the next tuple, call the result *actVal*; randomly select between the value in the accumulator and *actVal*; store the selected value in the accumulator.
3. Repeat the previous step until all tuples are exhausted; copy the value in the accumulator to all the slots in the column for the virtual domain.

[12]The `nop` operation can also be used as a binary domain operator or a join operator between relations; due to the space limit, the implementation details for these cases will not be given in here. Appendix C contains pointers to the code in the system.

When there is only one tuple to begin with, the `red nop` operation is trivial. As a matter of fact, we could even have used `fun nop` in the car race example and the result would be the same.

Level-lifting is intended for use with the reduction operations only, as all the tuples have the same value for the virtual domain and no ambuiguity results. The syntax for level lifting resembles

```
let newDom be [red OP of DEXPR] in REXPR
```

where `OP` is a commutative and associative operator, `DEXPR` is a domain algebra expression, and `REXPR` is a relational expression that produces a unary singleton result.

To implement level-lifting, we modified both the Interpreter and the Actualizer class, in the following methods:

- *Interpreter.traverseType()* This method is responsible for deducing the type of `newDom` upon its declaration. When the syntax tree of type OP_PROJECT or OP_TSELECT is received, the method performs a check to see if there is only one element in the projector list, *and* if the element is a reduction expression. If either of these conditions fails, the normal routine is resumed and a type of IDLIST will be returned. Otherwise, it calls itself to recursively deduce the type of `DEXPR` and use that as the type of `newDom`.
- *Actualizer.buildTree()* This method performs run-time type checking to make sure the result agrees with the declared type as found by the interpreter. It also truncates the virtual domain syntax tree as appropriate. In the case of `newDom`, the node representing `red OP of DEXPR` will be taken off the syntax tree (because it does not have a name) and used as the declaration syntax for a temporary domain `reddom`. In its place, a node of type OP_IDENTIFIER will be substituted in the original syntax tree for `newDom`. The name of the identifier is "reddom". Thus in effect, two virtual domains will be actualized: `reddom` and `newDom` (see Figure 5.8). In addition, the Domain object representing `reddom` will have its `isRedTemp` flag set to 1. The run-time type checker will use the type found for `reddom` as the type of `newDom`, which should agree with the conclusion reached by the interpreter.

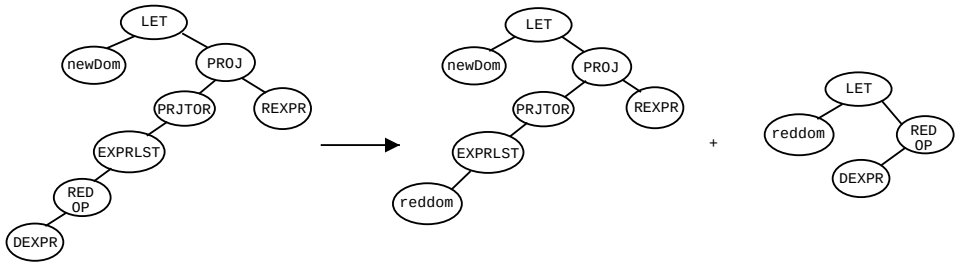

Figure 5.8: Syntax Tree Change for Level-Lifting

- *Actualizer.act*Cell()* These are the "cell methods" mentioned in Section 5.1.3. Special handling is needed when a node of type OP_PROJECT or OP_TSELECT is encountered and the `isRedTemp` flag for the only domain in the projector list has a value of 1. The node will be evaluated using *actRelCell()* as if the virtual domain were relational. The value of the tuple in the resulting relation is then used as the return value of the "cell method". In the case of a nested relation, tuples are extracted from the dot relation of `reddom` and placed into `reddom` itself (except for the ".id" domain).

Declaring and Exporting a Method

Declaring an accessor method takes no more than writing the code for the method as a nested-level computation in the ADT and listing its name in the parameter list. A method declaration inside an ADT is a jRelix statement, which is processed in the *runSingleStmt()* method. When a relation is ijoined with an ADT, all the statements in the 'alt' block of the ADT are executed in sequence for every tuple in the relation. Thus the method declaration statement is also processed once for each tuple. When the declaration is encountered for the first tuple, two cases need to be distinguished:

- *Case 1.* This is the first time the ADT is instantiated. Therefore the global CompTable object is ignorant of the method being declared. In this case, a new surrogate is generated and its value stored in the result relation. A new Computation object is also created to represent the method. Next, we add to the CompTable object a mapping from the surrogate to the Computation object. We

also add a reverse mapping, from the fully qualified name of the method to its surrogate. The fully qualified name is obtained by concatenating the parent computation's fully qualified name to an "!" mark, followed by the method's name. For example, the fully qualified name for the `GETSPEED` method in the `RaceCar` ADT (Figure 4.2) is "RaceCar!GETSPEED". The use of fully qualified names in CompTable makes it possible for different ADTs to have methods of the same name. It also benefits the implementation of packages.

- *Case 2.* A previous instantiation of the ADT exists. In this case, the CompTable object already has mappings for the method in question. Therefore, we can simply look up its surrogate value and put it in the result relation.

As the system processes other tuples, it will find that all the methods to be exported can be found in the global CompTable, and thus the action to be taken is similar to to that in Case 2 above. Therefore, all surrogates in a relation resulting from an ADT instantiation point to the same instance of the method. Furthermore, later instantiations of the same ADT also share this method instance.

Invoking an Accessor Method

Let us now review how the accessor method in the car race example (see Section 4.1.2) was invoked by a user. To save the reader going back and forth, the code is re-produced as follows:

```
>pr TopRacers;
+--------------------+------------+------+----------+-------+-----+------+
| name               | ACCELERATE | STOP | GETSPEED | *_a*  | *_v*| *_v0*|
+--------------------+------------+------+----------+-------+-----+------+
| James Bond         | 1          | 2    | 3        | 6000.0| 0.0 | 0.0  |
| Michael Schumacher | 1          | 2    | 3        | 5500.0| 0.0 | 0.0  |
+--------------------+------------+------+----------+-------+-----+------+
relation TopRacers has 2 tuples

1 >let speed' be GETSPEED[];
2 >let speed be [red nop of curSpeed] in speed';
3 >AllSpeeds is [name, speed] in TopRacers;
4 >pr AllSpeeds;
+----------------------+---------------+
| name                 | speed         |
+----------------------+---------------+
| James Bond           | 0.0           |
| Michael Schumacher   | 0.0           |
+----------------------+---------------+
expression has 2 tuples
```

In the code above, `TopRacers` is a relation which holds two instantiated `RaceCar` objects. Upon instantiation, methods defined in the `RaceCar` ADT (Figure 4.2) became computation typed domains of `TopRacers`, represented by their surrogates. `GETSPEED` is the accessor method we would like to invoke in order to get the current speed (value of the hidden state) for each racer. From the definition of the ADT , we see that it has one parameter, `curSpeed`, which is intended as the output.

The lines numbered 1 to 4 in the code illustrate the typical steps involved in using an accessor method. We now go over each of them and explain how jRelix has been implemented to respond in each step. The following discussion is generally applicable to all use cases of an accessor method.

Step 1. Declare a virtual domain to hold the result of GETSPEED(curSpeed)

e.g. `let speed' be GETSPEED[];`

Since `GETSPEED` is functional, we can use it in the domain algebra. When the virtual domain declaration for `speed'` is received by the interpreter, it first performs validity checks and deduces the type of the new domain. Methods *traverseNode()* and *traverseType()*, which are invoked in this step, have been augmented to recognize that `GETSPEED[]` is a computation invocation using the array syntax and as such, the type of `speed'` should be `IDLIST` — a relation. In addition, this relation is found to contain just one domain, `curSpeed`. Next, the interpreter records the new domain in a system table for future use. Meanwhile, a dot relation for `speed'`, namely `.speed'`, is created and recorded as well.

Step 2. Declare a virtual domain to hold the result of level-lifting

e.g. `let speed be [red nop of curSpeed] in speed';`

As per the discussion in the subsection *Transforming a Unary Singleton Relation* on page 111, the effect of this statement is to put the value of the tuple in the unary singleton relation represented by `speed'` into `speed`. However, this will happen only when `speed` is actualized. For now, the interpreter treats it like a normal virtual domain declaration, although some special action is needed to deduce the type of `speed`. Again

in the *traverseType()* method, code has been added to handle the case where a virtual domain is defined on a projection, the projector list of which consists of a reduction. The type of the virtual domain, in this case, is taken to be the type of the expression after the "of" keyword. `speed` is thus found to be of type FLOAT.

It is actually more succinct to combine steps 1 and 2 as:

```
let speed be [red nop of curSpeed] in GETSPEED[];
```

JRelix will respond in the same way as in the case of two separate declarations.

Step 3. Actualize the virtual domain(s)

e.g. `AllSpeeds is [name, speed] in TopRacers; pr AllSpeeds;`

These two statements could have been written as one, if `AllSpeeds` would no longer be used in the program:

```
pr ([name, speed] in TopRacers);
```

The interpreter evaluates the view `AllSpeeds` before executing the "pr" command. `name` is an actual domain of `TopRacers`, so it needs no further processing. To actualize the virtual domain `speed`, the interpreter first creates and initializes an Actualizer object, and then invokes its *actualize()* method. These two simple steps abstracts away all the details buried inside the actualizer, which we will discover now.

The most important step involved in the initialization process of an Actualizer object is transforming the syntax tree of `speed` according to the rules of level-lifting (covered in *Transforming a Unary Singleton Relation* on page 111). As a result, the declaration of `speed` turns into two:

```
let redtemp be red nop of curSpeed;
 let speed be [redtemp] in speed';
```

where `redtemp` is a special temporary domain usded exclusively by the level-lifting algorithms.

When the *actualize()* method is called subsequently, it performs the following initialization steps:

1. Create an NREnvironment object (say `nrObj`) to hold the bindings for the domains of `TopRacers`. Thus `nrObj` knows that `GETSPEED` is a computation typed domain, whose surrogate is available as data in the Relation object for `TopRacers`.

2. Set the current row number memory (`curRow`) to 0.

Next, actualization begins for `speed` in the first tuple of `TopRacers`. Since `speed` is of type FLOAT, the "cell method" *actDoubleCell()* is called and passed the syntax tree for `[redtemp] in speed'`. Recognizing the expression as a relational projection, *actDoubleCell()* delegates the task to the *evaluateTLExpression()* method of the interpreter.

The interpreter looks up the name `speed'` and finds that it contains a computation call to `GETSPEED`. Based on information found in the NREnvironment object (`nrObj`), it obtains the surrogate value for `GETSPEED` as 3. This value is then used as a key to look up the CompTable object for the computation representing `GETSPEED`. Finally, the interpreter invokes the appropriate method in of the Computation object to handle the call.

The `GETSPEED` computation takes over and executes its code on the first tuple of the relation `TopRacers`. In the end, it creates and returns a unary singleton relation defined on `curSpeed`. The value of the tuple is the value of the hidden state `_v` for "James Bond".

The interpreter passes along the result of the computation to the actualizer. After performing level-lifting, the actualizer finally puts the value of `curSpeed` into the appropriate spot of the relation `TopRacers`.

Every tuple of `TopRacers` goes through the same process as the first. That is, the accessor method `GETSPEED` is executed once per tuple.

In order to make the above algorithm work, adjustments have been made to the following classes: NREnvironment, Computation, Interpreter and Actualizer. Among other things, the look-up routines of the NREnvironment class have been modified to accomodate computations nested inside a relation. In addition, a new `applySelect()` method has been created in the Computation class. This method allows a Computation object to handle invocation on a nested computation.

5.2.4 Implementation of Modifier Method

To use a modifier method of an ADT, we must use the update statement on the relation that contains the method, as the internal states will be changed by the execution of the method, and thus the relation will also be modified. Consequently, the modifier method is implemented mainly by enhancing the routines in the Interpreter class that handle updates. Two such methods have been modified, as follows:

- *lookUpdate()* This routine generates the three parts of a trigger (see [Sun00]). In order to create the *new* part, the statements after the keyword `change` in an update statement are executed in turn. To support the invocation of a modifier method, the case of a computation call (OP_COMPCALL) must be handled. The solution is actually pretty straight-forward. First we extract the pure data part of the affected relation (not including computation typed domain) and put it into a temporary relation. Then we run the requested computation on each tuple of the temporary relation. The Computation object representing the modifier method is found via an NREnvironment object, in the same that an accessor method is located. Finally, we combine the result with data for the computation typed domains and create the *new* part of the trigger.
- *doTrigger()* This routine reassembles the three parts of a trigger and puts a new version of the affected relation in the appropriate environment. The original method was implemented with the assumption that the outer most relation affected by an update is a top-level one. However, in the case of ADT, this need not be the case. An update issued from within the body of a modifier method can affect a relational domain at any level of nesting of the relation holding the method. Therefore, some code changes have taken place to make the *doTrigger()* routine more flexible. In addition, efforts have been made to ensure that the new relation resulting from the update is always put in the same environment where the old one is found. This is crucial in view of deeply nested relations, as we allow domains of the same name to exist at different levels of nesting.

For the same reason that justifies an overloaded *applySelect()* method, we added an overloaded *applyInOut()* method to the Computation class. The update routine *doTrigger()* calls this overloaded method. This takes care of the issues that arise when a computation nested inside a relation is invoked. In particular, the computation not only needs to know the parameter values, it also must know where to find the data in its enclosing relation. This is because the modifier method is meant to modify hidden states, which are implemented as special domains in the relation resulting from an ijoin instantiation of an ADT.

Chapter 6

Implementation of Extended Domain Operation

This chapter describes the computation based extension to vertical domain operations. The reader is encouraged to consult the section "System Overview" (Section 5.1) to gain some knowledge of the system components, especially the Actualizer class.

6.1 Vertical Domain Actualization: Overview

A virtual domain is declared with a **let** statement and actualized when used in a relational algebra expression. The task of virtual domain actualization is accomplished by the Actualizer class. The methods in this class fall into two categories, those that handle horizontal operations (which we call "cell methods"), and those that work with vertical operations. As of the most recent version of jRelix prior to this writing, all four types of vertical domain operations (reduction, equivalence reduction, functional mapping and partial functional mapping) have been implemented. The operators that appear after the keyword ("red" or "equiv") in reductions must be commutative and associative. A broader range of operators are available in functional mappings[1]. However, in either case, the selection of operators is limited to the ones built into the system. For example,

[1] see Section 3.4.2 for a description of operators used in vertical operations

the data in an integer typed domain can only be processed with the common mathematical operators; vertical operations on relation typed domains can involve nothing else than `ijoin`, `ujoin` and `sjoin`. This restriction has been lifted with the introduction of a computation based extension. Before plunging into the implementation details, we first give a brief review of the process of vertical domain actualization. Such background knowledge is intended to help the reader understand how the extension fits in with an existing system.

6.1.1 Algorithms

Reduction

The syntax for declaring a virtual domain with reduction is:

```
"let" newDom "be" "red" OP "of" DEXPR ";"
```

where `newDom` is the name of the virtual domain; `OP` stands for the operator used in reduction; `DEXPR` is a domain expression.

An example of actualizing a virtual domain defined on `red nop` was given in Section 5.2.3. For the sake of completeness, we give the algorithm for a generic reduction as follows[2]:

1. Load the source relation, `srcRel` (i.e. the relation the virtual domain is actualized in).
2. Initialize the destination relation, `destRel`.
   (a) Create `destRel`, defined on the domains of `srcRel` and the virtual domain.
   (b) Copy the data from `srcRel` into `destRel`.
3. Initialize the accumulator.
4. Set current row number, `curRow`, to 0.
5. If there are more tuples in `destRel`, then
   (a) Evaluate `newDom` based on the current tuple data.

[2]From now on, we assume only one virtual domain is being actualized.

(b) If `curRow = 0`, assign the result of evaluation (`actVal`) to the accumulator. Otherwise, do accumulator `OP actVal`; store the result back into the accumulator.

6. Repeat step 4 until all tuples are exhausted.
7. For each tuple in the destination relation, set the value of the virtual domain to be the value in the accumulator.

Equivalence Reduction

The syntax for declaring a virtual domain with equivalence reduction is:

```
"let" newDom "be" "equiv" OP "of" DEXPR "by" BY-LIST ";"
```

where `BY-LIST` is a comma delimited domain list.

The algorithm for actualizing equivalence reduction needs to take grouping into account. Tuples that have the same value for the domains in the `BY-LIST` belong to the same group. Within the group, it is essentially similar to the algorithm of reduction. Here is the complete algorithmic description:

1. Load the source relation (`srcRel`) without sorting.
2. Initialize the destination relation (`destRel`).
   (a) Create `destRel` on the domains of `srcRel` and the virtual domain.
   (b) Copy the data from `srcRel` into `destRel`.
   (c) Sort `destRel` on the domains in `BY-LIST`.
3. Initialize the accumulator.
4. Set the current row number, `curRow`, to 0.
5. Initialize the start row number, `start`, with `curRow`.
6. If there are more tuples in `destRel`, then
   (a) Evaluate `newDom` for the current tuple to obtain `actVal`.
   (b) Check `curRow`.
      i. If `curRow = 0`, assign (`actVal`) to the accumulator.
      ii. If this is the last tuple, then
         A. do accumlator `OP actVal`,

B. update the data (between `start` and `curRow`) of `destRel` for the virtual domain with the accumulator,
C. set `start` to be (current row + 1).

iii. If this is a tuple in the middle, then
A. Check if grouping has changed. If so, then
(1) update `destRel` (between `start` and `curRow`) with the accumulator,
(2) set `start` to be (current row + 1).
B. Do accumulator `OP actVal`; store the result back into the accumulator.

7. Repeat step 6 until all tuples are exhausted.

Functional Mapping

The syntax for declaring a virtual domain with functional mapping is:

```
"let" newDom "be" "fun" OP "of" DEXPR "order" ORDER-LIST ";"
```

where `ORDER-LIST` is a comma delimited domain list.

With functional mapping, the operation (`OP`) is applied to the domain (`DEXPR`) in an order induced from the domains in `ORDER-LIST`. The algorithm to actualize functional mapping is:

1. Load the source relation (`srcRel`) without sorting.
2. Initialize the destination relation (`destRel`).
   (a) Create `destRel` on the domains of `srcRel` and the virtual domain.
   (b) Copy the data from `srcRel` into `destRel`.
   (c) Sort `destRel` on the domains in `ORDER-LIST`.
3. Initialize the accumulator.
4. Set the current row number, `curRow`, to 0.
5. If there are more tuples in `destRel`, then
   - if `curRow = 0` or if order has changed, then
     (a) calculate `actVal` by evaluating `newDom` based on the current tuple data,
     (b) do accumulator `OP actVal`; assign the result to the accumulator,
     (c) update `destRel` with the accumulator.

- Otherwise, update `destRel` with the current value in the accumulator (bypass).

6. Repeat step 5 until all tuples are exhausted.

Partial Functional Mapping

The syntax for declaring a virtual domain with partial functional mapping is:

```
"let" newDom "be" "par" OP "of" DEXPR "order" ORDER-LIST "by" BY-LIST
                                                        ";"
```

Partial functional mapping is the most complicated among the four types of vertical operations. It adds grouping on top of functional mapping. The algorithm to actualize partial functional mapping is as follows:

1. Load the source relation (`srcRel`) without sorting.
2. Initialize the destination relation (`destRel`).
   (a) Create `destRel` on the domains of `srcRel` and the virtual domain.
   (b) Copy the data from `srcRel` into `destRel`.
   (c) Sort `destRel` on the domains in `BY-LIST`.
   (d) Sort `destRel` on the domains in `ORDER-LIST`.
3. Initialize the accumulator, group memory[3], and order memory[4].
4. Set the current row number, `curRow`, to 0.
5. If there are more tuples in `destRel`, then
   - if `curRow > 0` and grouping has changed, re-initialize group memory and order memory,
   - otherwise,
     - if `curRow = 0` or if order has changed, then
       (a) calculate `actVal` by evaluating `newDom` based on the current tuple data,
       (b) do accumulator `OP actVal`; assign the result to the accumulator,
       (c) update `destRel` with the accumulator.

[3]Grouping memory tracks the current value of domains in `BY-LIST`.

[4]Order memory tracks the current value of domains in `ORDER-LIST`.

- Otherwise, update `destRel` with the current value in the accumulator (bypass).

6. Repeat step 5 until all tuples are exhausted.

6.1.2 Previously Implemented Methods

Most of the initialization steps for vertical domain actualization (load the source relation, create the destination relation, copy data from source to destination) are performed by the *actualizing()* method of the Actualizer class. This method also implements the control flow for reduction. There is, however, a separate method for each of the remaining three types of vertical operations. They are *actualizeEquiv()* [Yua98], *actualizeFun()* [Kan01], and *actualizeParFun()* [Kan01]. Apart from handling grouping and ordering, two other important tasks accomplished by these methods are: (1) evaluating the vertical domain expression for the current tuple, and (2) updating the accumulator. The first task is delegated to the "cell methods" (see Table 5.3 for a list of these methods), while the second is handled separately in each of the four methods for vertical operations.

6.2 Computation Based Extension

The extension to the vertical domain operations described in this thesis is based on computation. We know from the previous section that the process of actualization involves, at some point, updating the accumulator with the result of executing `accumulator OP actVal`, where `actVal` comes from the "cell methods" and `OP` from the virtual domain declaration. Extending the vertical domain operations means allowing user defined operators to be used as `OP`. That is to say, `OP` is no longer limited to the built-in operators such as "+" or "ijoin". It can be any computation (or the operation carried out by a block of computation, to be precise) that satifies the constaints imposed by reduction or functional mapping.

To support this kind of extension, we have enhanced the existing system with new syntax, which was introduced in Section 4.2.2. Consequently, the code in several classes

has been modified to recognize the new syntax and react accordingly.

6.2.1 Additions to the Constant Class and Parser Actions

To distinguish an extended vertical operation from an ordinary one, we defined a new opcode for the extended operation in the Constant class: OP_REDFUNCALL. When the parser encounters a virtual domain declaration compliant with the extended syntax, it sets the opcode field of the SimpleNode object for the declaration to `OP_REDFUNCALL`. In the ordinary case, this opcode field holds a constant corresponding to a system defined operator, such as OP_PLUS.

A second parser action has been added to mark a "redop" or "funop" computation block during the parsing process. The `bits` field of the SimpleNode (see Section 5.1.3) object representing a "redop" block has a value of 1; the value of `bits` of a "funop" block node is 2.

6.2.2 Additions to the CompBlock Class

Two new boolean data members have been added to the CompBlock class: `redop` and `funop`. When the CompBlock class constructor is invoked to create a new CompBlock object from a SimpleNode object, it checks the `bits` field and sets `redop` to true if its value is 1. If the `bits` field has a value of 2, both `redop` and `funop` are set to true. All other cases result in both being false. The `redop` and `funop` fields will be used for validity check purposes in the virtual domain actualization process.

6.2.3 Additions to the Actualizer Class

In view of the vertical domain actualization process given in Section 6.1, the extension theoretically only affects the way the accumulator is updated. In reality, the change of syntax necessitates corresponding code changes in the Actualizer class in general.

General Impact

A number of methods of the Actualizer class have been slightly modified to handle the syntax change. For example, the code in *actualizeEquiv()* to extract the *BY-LIST* from an equivalence reduction expression has been augmented with a test for `OP_REDFUNCALL`, as shown in Figure 6.1. The position index of the child node corresponding to the *BY-LIST* is incremented by one, due to the addition of an extra node for the computation call (see Figure 6.2 for an illustration). All general changes to the Actualizer class are of this nature.

```
SimpleNode orderby = null;
if(virtree.opcode != OP_REDFUNCALL)
  orderby = (SimpleNode)virtree.jjtGetChild(1);
else
  orderby = (SimpleNode)virtree.jjtGetChild(2);
```

Figure 6.1: Example of General Code Change in the Actualizer Class

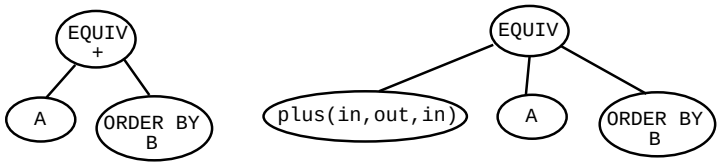

Figure 6.2: Syntax Tree of Extended Vertical Operation

The *RedFunCallAccumNode* Method

A new method, *RedFunCallAccumNode()*, has been added to the Actualizer class. All the methods for vertical domain actualization call this method when the accumulator needs to be updated in the extended case. This method accepts three arguments: the accumulator, the current value of the virtual domain (`actVal`), and the type of the virtual domain. The algorithm for this method is as follows:

1. If the accumulator is empty, assign `actVal` to the accumulator and exit.
2. Find the Computation object whose name is specified in the computation call. If no such computation is found, throw an exception.
3. Calculate the type of the target block.

- If no parameter list is present, assume the block type to be 011_2 (decimal 3).

- Otherwise calculate the block type from the parameter list. Specifically,
  - (in, in, out) ==> block type is 011_2 (decimal 3)
  - (in, out, in) ==> block type is 101_2 (decimal 5)
  - (out, in, in) ==> block type is 110_2 (decimal 6)

  Any other combination of "in" and "out" results in an exception.

4. Find the computation block according to the block type calculated in the previous step. If no matching block is found, throw an exception.
5. Check the `redop` and `funop` fields of the found block and verify the selected block is suitable for the vertical operation requested. That is, reduction and quivalence reduction require `redop == true`; functional mapping and partial functional mapping require `(redop || funop) == true`.
6. Verify that the type of the parameters of the computation matches that of the virtual domain being actualized.
7. Find out the names of the two "in" parameters (call them `firstIn` and `secondIn`) and the position of the "out" parameter (`outPos`).
8. Construct a syntax tree that corresponds to:

   ```
   where firstIn=firstVal and secondIn=secondVal.
   ```

   `firstVal` is the value of the accumulator and `secondVal` is the current value of the virtual domain. In the case of an IDLIST typed virtual domain, two temporary relations are created and registered into the environment; the names of the relations replace `firstVal` and `secondVal`.
9. Call the *applySelect()* method of the Computation object with the select predicate constructed in the previous step.
10. Extract the value of the output from the result of the computation invocation. The position of the output is indicated by `outPos`, found in step 7.
11. Place the extracted value (or relation, in the case of an IDLIST typed virtual domain) into the accumulator.

Chapter 7

Conclusions

This chapter begins with a summary and discussion of the work that has been accomplished. This is followed by suggestions for potential extensions and enhancements to the jRelix system in the future.

7.1 Conclusions

7.1.1 Summary of Present Work

This thesis documents the design and implementation of two new features to the database programming language, jRelix. Utilizing a nested relational model and an improved procedural abstraction facility, ADTs are declared as computations encapsulating states with their accessor/modifier methods. Objects of an ADT can be instantiated via a single join. As computation calls are embedded into updates and virtual domain actualization, objects are manipulated and accessed solely through the methods exported by the ADT.

The vertical domain algebra empowers jRelix with the capability to combine values along a domain using system defined operators. A mechanism has now been installed to run user defined computations as well. As a matter of fact, all built-in vertical operations can be similated by user-defined operations.

With these new features, application programmers of jRelix can now handle complex data objects on a higher conceptual level, using a modular approach.

7.1.2 Discussion

The database programming language described in this thesis has been built upon general purpose formalisms. As such, it is theoretically capable of supporting any application without special syntax or semantics. For example, it can handle geo-spatial computations required by a Geographic Information System (GIS). This is significantly different from the practices in the commercial world where GIS has grown independently of database languages.

To illustrate, we now give a code sketch for map overlay, a typical operation of GIS. An ADT, *MAP*, encapsulates the quad-edge representation [GS85] of a map and its associated spatial operations (e.g. splice). The quad-edge representation consists of three relations [MBC+01]: *QuadEdge*, *VertFace*, and *Geom*. The ADT is placed in a package, *Spatial*, together with other ADTs, as shown below[1]:

```
comp Spatial (MAP, EnumSeqADT) is
{ comp MAP(init,splice,makeEdge,delEdge) is
  state QuadEdge(e1,d1,e2,d2);
  state VertFace(e,orig,dest,l,r);
  state Geom(vf,x,y);
  { comp init(q,v,f) is
    { QuadEdge <- q;
      VertFace <- v;
      Geom <- f;
    };
    comp splice (pair) is
    { ... };
    comp makeEdge (pnts) is
    { ... };
    comp delEdge (edges) is
    { ...};
  };
  ...
};
```

Suppose that the above code has been given in a library. Furthermore, the library also supplies an *overlay* computation which consists of a *redop* block for calculating the overlay of two maps (map overlay is associative and commutative).

[1] The details of the ADT methods are not shown, as our intention is to show how the overlay problem can be tackled using jRelix constructs. The ensueing code adopts the same approach.

```
let Q1 be MAP;
let Q2 be MAP;
let Q3 be MAP;
comp overlay(Q1,Q2,Q3) is
redop
{ <<code to calculate the overlay of Q1 and Q2
  <<and assign the result to Q3
  ...
};
```

When an application programmer needs to perform map overlay, he or she will first include the spatial library code, then instantiate and initialize map objects as follows (`Q`, `V`, `T` are initial values for the quad-edge representation):

```
relation InitRel(ID,Q,V,T) <-
 {(101,{...},{...},{...}),
  (102,{...},{...},{...}),
  (103,{...},{...},{...})};
Maps <- InitRel ijoin ([MAP] in Spatial);
update Maps change
 (update MAP change init(in Q,in V,in T);
Maps <- [ID,MAP] in Maps;
```

In order to calculate the overlay of all three maps, the application programmer just needs one statement, taking advantage of the extended reduction and the level-lifting facility:

```
OverLayMap <- [red overlay of MAP] in Maps;
```

What we have just seen is an example of jRelix handling domain-specific application without special extensions. This capability is beyond any commercial relational database systems known as of this writing.

7.2 Future Work

7.2.1 Object Orientation and jRelix

JRelix supports *encapsulation* by ADT and *instantiation* by joining an ADT with an appropriate relation. The other main feature of object-orientation that is not yet available is *inheritance*.

The term *inheritance* describes mechanisms in which type definitions or implementations can be related to one another through a partial order [AH90]. The basic notion

is that we can modify type definitions incrementally by adding subtype definitions to enhance or override the original type. The combination of the supertype definition and the subtype modifications produces a completely defined new type.

A model of inheritance in jRelix could be built upon a special implementation of joining ADTs. Consider the following definition for a *person* ADT:

```
comp person (init_name, init_age, getName, getAge) is
state name strg;
state age intg;
{ name <- init_name;
  age <- init_age;
  comp getName(myName) is
  { myName <- name;};
  comp getAge(myAge) is
  { myAge <- age;};
};
```

We can define a *student* ADT as a subtype of *person*:

```
comp student(init_ID, getID) is
state ID long;
{ ID <- init_ID;
  comp getID(myID) is
  { myID <- ID;};
};

student <- student ijoin person;
```

The first part of the code above gives the special information and operation on a student. The second part specifies *student* as a subtype of *person* by means of an ijoin. The semantics of this assignment are special in that

- the join operation produces the complete definition of the subtype, and
- the name of the subtype must appear to the left of the assignment operator.

An alternative notation could be used as syntactic sugar for the assignment statement, as follows:

```
student isa person;
```

From this point on, we may instantiate *student* objects in the usual way (e.g. via ijoin). The instantiated objects would have *name*, *age* and *ID* as hidden states, as well as all three methods of the *student* type.

To implement this, the **Interpreter** class and the **Computation** class will need adjustments to handle the special semantics of inheritance. One possible solution is to add a `supertype` field to the Computation class. In the previous example, we could set `student.supertype` to `person` upon the "isa" operation. Then, given the statement

```
Class <- initRel ijoin student;
```

the interpreter should be able to

- recognize *student* as a subtype of *person* by checking `student.supertype`
- interpret the assignment as if it were:

```
Class <- (initRel ijoin person) ijoin student;
```

7.2.2 `red ujoin` vs. `red UJOIN`

With the support for user-defined computations in vertical domain operations, we may actually simulate any of the built-in vertical operations. Consider the following definition of *UJOIN*:

```
comp UJOIN(R1,R2,R3) is
{ R3 <- R1 ujoin R2;
};
```

Given a nested relation `A(a, R)` where `R` is a relation typed attribute, the expression `[AllR] in A` would produce the same result, whether we define

```
let AllR be red ujoin of R;
```

or

```
let AllR be red UJOIN of R;
```

However, when speed is a concern, the second form of virtual domain declaration should be avoided. The current implementation relies on the user-defined computation being called once for each tuple, not a very efficient solution. Future work may explore ways to improve execution efficiency in such cases.

7.2.3 Computation Implementation: Loose Ends

JRelix currently employs the pass-by-name parameter passing method for computation. With this method, the actual parameter is substituted for the corresponding formal parameter in all its occurrences upon invocation using the ijoin syntax. Theoretically speaking, any expression could be used as input parameters in a computation call. For example, given the declarations in Figure 3.35 and Figure 3.36, we could say

```
SuperSet(in where name>"G" in G2, in [name] in G3, out sup1)
```

However, in reality the parse will report an error, as it has been programmed to accept either a constant or an identifier as a parameter.

A second case of the imperfect implementation of computation is illustrated as follows:

```
comp Select(field,relIn,relOut) is
{ relOut <- [field] in relIn;
};

relation R(a,b) <- {...};
Select(in b, in R, out B);
Select(in a, in R, out A);
```

The intention of the last two statements is for A to evaluate to

```
[a] in R
```

and B to

```
[b] in R
```

However, this cannot be achieved with the current implementation. This is because all domains of a computation must be unambiguously defined before the computation itself. But in our example, the domain that `relOut` is defined on is unknown until invocation time. It may be desirable to change the implementation such that the output parameters of a computation are allowed to be left undefined until invocation.

Finally, the so-called "multi-valued computation" [Bak98] using an "also" syntax has not been implemented. Future work is needed to provide support for this functionality.

Appendix A

Backus-Naur Form for the Parser

This appendix shows the Backus-Naur form (BNF) of the grammar in our implementation. The convention of the BNF definition is shown in Table A.1.

Form	Meaning
<SYMBOL>	SYMBOL is a definition of token and must be substituted
"SYMBOL"	SYMBOL is reserved word or symbol and must be typed as it is
S1 | S2	either S1 or S2 can be used
(SYMBOL)?	SYMBOL is optional
(SYMBOL)*	SYMBOL may appear zero or more times
(SYMBOLS)	grouping SYMBOLS as one unit for high precedence

Table A.1: BNF convention.

The grammar is created from the grammar specification (in file Parser.jjt), using the JavaCC documentation generator called `jjdoc`. Because JavaCC is a top-down parser, left-recursion is not allowed in the grammar.

There are five token definitions: <EOF> for end-of-file; <IDENTIFIER> for identifier; <INTEGER_LITERAL> for integer constants; <FLOAT_LITERAL> for floating constants; and <STRING_LITERAL> for string constants. The formal definitions of identifier and constants are given in [Hao98].

```
Start ::= Command ";" | Statement ";"
           | ";" | <EOF>
Command ::= "help" (<IDENTIFIER>)?
    | "quit"
    | "input" <STRING_LITERAL>
    | "debug"
    | "time"
    | "trace"
    | "dd" IDList
    | "dr" (IDList
            | EventName ("[" IDList "]")?)
    | "pr" (Expression
            | EventName ("[" IDList "]")?)
    | "sd" (<IDENTIFIER>)?
    | "sr" (<IDENTIFIER>)?
    | "sc" <IDENTIFIER>
    | "srd"
    | "print" <STRING_LITERAL>
    | "ssd"
    | "ssr"
    | "undo"
    | "eventon" EventName ("[" IDList "]")?
    | "eventoff" EventName ("[" IDList "]")?
Statement ::= SequentialStatement
SequentialStatement ::= ParallelStatement
                    ("--" ParallelStatement)*
ParallelStatement ::= ChoiceStatement
                    ("||" ChoiceStatement)*
ChoiceStatement ::= PrimaryStatement
                    ("??" PrimaryStatement)*
PrimaryStatement ::= Declaration
     | Assignment
     | Update
     | ComputationCall
     | Conditional
     | ForLoop
     | WhileLoop
     | Exit
     | DeadLock
     | Exec
     | "(" Statement ")"
     | StatementBlock
StatementBlock ::= "{" Statement
                    (";" Statement)* (";")? "}"
Conditional ::= "if" Expression
                "then" (Statement | Command)
               ("else" (Statement | Command))?
ForLoop ::= ("for" Identifier)?
    ("from" Expression)?
    ("to" Expression)?
    ("by" Expression)?
     "do" Statement
WhileLoop ::= "while" Expression "do" Statement
Exit ::= "exit"
DeadLock ::= "deadlock"
Exec ::= "exec" Identifier
Declaration ::= "relation" IDList "(" IDList ")"
           (Initialization)?
| Identifier ("initial" Expression)?
  "is" Expression ("target" Expression)?
| "domain" IDList Type
| "let" (Identifier | Eval)
  ("initial" Expression)?
  "be" Expression
| ("comp" | "computation") CompName
  "(" (ParameterList)? ")"
  "is" ComputationBody
Initialization ::= "<-" ("{" ConstantTupleList "}"
   | Identifier | FilePath)
   | "rcp" (Identifier | FilePath)
   | "is" (Identifier | FilePath)
ConstantTupleList ::= (ConstantTuple
                      ("," ConstantTuple)*)?
ConstantTuple ::= "(" Constant ("," Constant)* ")"
Constant ::= Literal | "{" ConstantTupleList "}"
Identifier ::= <IDENTIFIER>
FilePath ::= <STRING_LITERAL>
Assignment ::= Identifier
               (AssignOperator Expression
       | "["IDList AssignOperator
                 ExpressionList"]" Expression)
AssignOperator ::= "<-" | "<+"
Update ::= "update" Identifier
           (UpdateOperator Expression
   | "change" (StatementList)?
             ("using" UsingClause)?
   | "["IDList UpdateOperator ExpressionList"]"
     Expression)
UpdateOperator ::= "add" | "delete"
StatementList ::= Statement ("," Statement)*
UsingClause ::= Identifier
| "(" Expression ")"
| JoinOperator Expression
| "[" ExpressionList ":" JoinOperator
  (":")? ExpressionList "]" Expression
IDList ::= Identifier ("," Identifier)*
ExpressionList ::= Expression ("," Expression)*
Expression ::= Disjunction
Disjunction ::= Conjunction
                (("|" | "or") Conjunction)*
Conjunction ::= Comparison
                (("&" | "and") Comparison)*
Comparison ::= Concatenation
       (ComparativeOperator Concatenation)?
Concatenation ::= MinMax ("cat" MinMax)*
MinMax ::= Summation
           (MinMaxOperator Summation)*
Summation ::= JoinExpression
      (AdditiveOperator JoinExpression)*
JoinExpression ::= Projection (
           (JoinOperator Projection
         | "["ExpressionList ":" JoinOperator (":")?
           ExpressionList"]" Projection))*
Projection ::= Projector
               (("in" | "from") Projection
       | Projector "gedit" Expression
       | "gedit" Expression
       | Selection)
       | Selection
Projector ::= (QuantifierOperator)?
              "[" (ExpressionList)? "]"
Selection ::= Selector | QSelector | Term
Selector ::= ("where" | "when") Expression
     ("in" | "from") Projection
    | "edit" (Projection)?
    | "zorder" Projection
QSelector ::= "quant" QuantifierList
    (("where" | "when") Expression)?
    ("in" | "from") Projection
QuantifierOperator ::= "." | "%" | "#"
```

```
QuantifierList ::= Quantifier ("," Quantifier)*
Quantifier ::= "(" Expression ")" Expression
Term ::= Factor (MultiplicativeOperator Factor)*
Factor ::= UnaryOperator Factor | Power
Power ::= Primary ("**" Power)*
Primary ::= Literal
    | QuantifierOperator
    | ArrayElement
    | PositionalRename
    | Identifier
    | Cast
    | "(" Expression ")"
    | Pick
    | AttribsOf
    | Quote
    | Transpose
    | Function
    | IfThenElseExpression
    | VerticalExpression
ArrayElement ::= Identifier "[" ArrayIndexList "]"
ArrayIndexList ::= (Expression)?
           ("," (Expression)?)*
PositionalRename ::= Identifier "(" (IDList)? ")"
Cast ::= "(" Type ")" Primary
Pick ::= "pick" Selection
AttribsOf ::= "AttribsOf" Selection
Eval ::= "eval" Expression
QuoteIdentifier ::= (Quote | Identifier)
Quote ::= "quote" Identifier
Transpose ::= "transpose" ExpressionList
Function ::= FunctionOperator "(" Expression ")"
Literal ::= "null"
    | "dc"
    | "dk"
    | "true"
    | "false"
    | (SignOperator)?
      (<INTEGER_LITERAL> | <FLOAT_LITERAL>)
    | <NUMERIC_LITERAL>
    | <STRING_LITERAL>
SignOperator ::= "+" | "-"
IfThenElseExpression ::= "if" Expression
  "then" Expression
  "else" Expression
VerticalExpression ::= "red"
 (ComputationCall | AssoCommuOperator)
 "of" Expression
       | "equiv"
 (ComputationCall | AssoCommuOperator)
 "of" Expression "by" ExpressionList
       | "fun"
 (ComputationCall | OrderedOperator)
 "of" Expression
 "order" ExpressionList
       | "par"
 (ComputationCall | OrderedOperator)
 "of" Expression
 ("order" ExpressionList "by" ExpressionList
        | "by" ExpressionList "order" ExpressionList)
Type ::= ("univ" | "universal")
 | ("attr" | "attribute")
 | ("bool" | "boolean")
 | "short"
 | ("intg" | "integer")
 | "long"
 | ("float" | "real")
 | "double"
 | "number"
 | ("strg" | "string")
 | "text"
 | ("stmt" | "statement")
 | ("expr" | "expression")
 | ("comp" | "computation") "(" (IDList)? ")"
 | "(" IDList ")"
AssoCommuOperator ::= ("|" | "or")
      | ("&" | "and")
      | "min"
      | "max"
      | "+"
      | ("ijoin" | "natjoin")
      | "ujoin"
      | "sjoin"
      | "*"
      | "nop"
OrderedOperator ::= AssoCommuOperator
    | "cat"
    | "-"
    | "/"
    | "mod"
    | "**"
    | "pred"
    | "succ"
ComparativeOperator ::= "substr"
| "="
| "!="
| ">"
| "<"
| ">="
| "<="
MinMaxOperator ::= "min" | "max"
AdditiveOperator ::= "+" | "-"
JoinOperator ::= "nop" | MuJoin
   | (("!" | "not"))? SigmaJoin
MuJoin ::= ("ijoin" | "natjoin")
   | "ujoin"
   | "djoin"
   | "sjoin"
   | "ljoin"
   | "rjoin"
   | "dljoin"
   | "drjoin"
SigmaJoin ::= ("icomp" | "natcomp")
      | "eqjoin"
      | "gtjoin"
      | ("gejoin" | "sup" | "div")
      | "ltjoin"
      | ("lejoin" | "sub")
      | ("iejoin" | "sep")
MultiplicativeOperator ::= "*" | "/" | "mod"
UnaryOperator ::= "+" | "-" | ("!" | "not")
FunctionOperator ::= "abs"
     | "sqrt"
     | "sin"
     | "asin"
     | "cos"
     | "acos"
     | "tan"
     | "atan"
     | "sinh"
     | "cosh"
```

```
     | "tanh"
     | "log"
     | "log10"
     | "round"
     | "ceil"
     | "floor"
     | "isknown"
     | "chr"
     | "ord"
ParameterList ::= Parameter ("," Parameter)*
Parameter ::= <IDENTIFIER> (":" "seq")?
ComputationBody ::= ComputationVariableDeclaration
    ComputationBlock
    ("alt" ComputationBlock)*
ComputationBlock ::= ("redop" | "funop")?
    "{" ComputationStatements "}"
ComputationVariableDeclaration
::= (LocalVariableDeclaration
   | StateVariableDeclaration)*
LocalVariableDeclaration ::= "local" IDList Type ";"
StateVariableDeclaration ::= "state" IDList Type ";"
ComputationStatements ::= CompStatement ((";" CompStatement
  | "also" CompStatement))* (";")?
CompStatement ::= Statement | Command
ComputationCall ::= Identifier "(" (CallParameterList)? ")"
CallParameterList ::= CallParameter ("," CallParameter)*
CallParameter ::= ( "in" (Literal | Identifier)?)
  | ( "out" ( Literal | Identifier )? )
CompName ::= CompIdentifier | EventName ("[" IDList "]")?
CompIdentifier ::= <IDENTIFIER>
EventName ::= (Prefix ":")? Action ":" (Identifier)?
Prefix ::= "pre" | "post"
Action ::= "add" | "delete" | "change"  | "contains"
| "cmpcontains"  | "cmpwithin" | "intersect"  | "within"
| "withindist"
```

Appendix B

JRelix System Class Map

Class Name	Description	Category
JRelix	Main Program	Front-end
JRelixInputStream	Extends BlockInputStream	Front-end
JRelixParser	Extends Parser	Front-end
Parser*	Language Parser	Front-end
ParserTokenManager*	Supports Parser	Front-end
ParserConstants*	Supports Parser	Front-end
ParserTreeConstants*	Supports Parser	Front-end
ParseException*	Supports Parser	Front-end
ParseError	Supports Parser	Front-end
ASCII_CharStream*	Supports Parser	Front-end
Token*	Supports Parser	Front-end
TokenMgrError*	Supports Parser	Front-end
Node*	Syntax Tree Node Structure	Front-end
SimpleNode	Enhanced Node Structure	Front-end
Interpreter	Interprets Syntax Trees	Front-end
PrettyPrint	Formats Display Output	Front-end
Constants	Operation Constants	Front-end, DB Engine
InterpretError	Supports Interpreter	Front-end, DB Engine
Global	Global Variables/Methods	Front-end, DB Engine
Relation	Implements Relational Algebra	DB Engine
RelTable	Supports Relation Lookup	DB Engine
RelInfo	Extends IDInfo for Relation	DB Engine
Domain	Implements Domain Algebra	DB Engine
DomTable	Supports Domain Lookup	DB Engine
DomInfo	Extends IDInfo for Domain	DB Engine
Actualizer	Implements Virtual Domain Actualization	DB Engine
Actualizer76	Supplements Actualizer	DB Engine
Note: Files with "*" are machine generated; Gedit** are all the classes the name of which begin with "Gedit"		

Table B.1: Map of JRelix Classes, Part 1

Class Name	Description	Category
Environment	System Environment	DB Engine
SemanticChecker	Semantic Check of User Input	DB Engine
SemanticCheckError	Supports Semantic Check	DB Engine
TypeConstants	Data Type Constants	DB Engine
Surrogate	Surrogate Values	DB Engine
Utility	Various Utilities	DB Engine
NREnvironment	Nested Relations Environment	DB Engine
NRInfo	Extends IDInfo for Nested Environment	DB Engine
Computation	Implements Computation	DB Engine
CompBlock	Supports Computation	DB Engine
CompTable	Supports Computation	DB Engine
IDInfo	Stores Variable Information	DB Engine
LocalInfo	Extends IDInfo for Local Variables	DB Engine
ParamInfo	Extends IDInfo for Parameters	DB Engine
StateInfo	Extends IDInfo for States	DB Engine
StateInfo_2002	Supplements StateInfo	DB Engine
EvalExpr	Evaluation of Expressions	DB Engine
Gedit**	GIS Editor Related Classes	DB Engine
AddLayersDialog	Supports GIS	DB Engine
CvDraw	Supports GIS	DB Engine
Legend	Supports GIS	DB Engine
TriggerNode	Supports Active Database	DB Engine
CTrigger	Supports Active Database	DB Engine
number	Implements Numerical Data Type	DB Engine
ExprEntry	Entry for .expr System File	DB Engine, DB Maintainer
ExprTable	Supports .expr Maintenance	DB Engine, DB Maintainer
BlockInputStream	Disk Input Operations	DB Maintainer
BlockOutputStream	Disk Output Operations	DB Maintainer
Note: Files with "*" are machine generated; Gedit** are all the classes the name of which begin with "Gedit"		

Table B.2: Map of JRelix Classes, Part 2

Appendix C

Summary of Enhancements

Class	Significance
StateInfo_2002 (new)	Provides storage for states
Actualizer	Supports OP_FUNCTION
	Supports OP_NOP
	Supports level lifting of anonymous domain
	Supports accessor method call in ADT
	Supports extended vertical domain operation
CompBlock	Supports extended vertical domain operation
CompTable	Supports nested computation lookup
Computation	Supports hidden states
	Supports enhanced set of statements and commands
	Supports invocation of nested accessor method
	Supports invocation of nested modifier method
Constants	Supports OP_REDFUNCALL
Domain	Supports hidden states
	Supports level lifting of anonymous domain
EvalExpr	Supports OP_FUNCTION
	Supports states
Global	Supports hidden state serialization
IDInfo	Supports type of subclass object
Interpreter	Supports top level OP_NOP
	Supports hidden states
	Supports level lifting of anonymous domain
	Supports executing statements and commands
	from inside a computation
	Supports update with computation call
LocalInfo	Supports relation typed local variable
NREnvironment	Supports lookup of nested computation
Parser	Supports new syntax related to extended domain operation
StateInfo	Supports states

Table C.1: New and Modified Classes

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proceedings of the 2nd ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 191–200, 1984.

[ABCea76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, and et al. System r: Relational approach to database management. *ACM Trans. Database Systems*, 1(2):97–137, 1976.

[AH90] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann publishers, 1990.

[AM84] M. P. Atkinson and R. Morrison. Persistent first class procedures are enough. *Lecture Notes in Computer Science*, 181:223–240, 1984.

[And99] Maxim Andreev. Operations on text in a database programming language. Master's thesis, McGill University, Montreal, Canada, 1999.

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

[BCG+90] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim. Data model issues for object-oriented applications. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann publishers, 1990.

[Bee88] C. Beeri. Data models and languages for databases. In *ICDT '88: 2nd International Conference on Database Theory*. Springer-Verlag, Bruges, Belgium, 1988.

[Bid87] N. Bidoit. The verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, 1987.

[BM88] F. Bancilhon and D. Maier. Multilanguage object-oriented systems: New answer to old database problems? In K. Fuchi and L. Kott, editors, *Future Generation Computer II*. North Holland, Amsterdam, 1988.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases*, Kyoto, Japan, 1986.

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

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

[Cod72a] E. F. Codd. Database systems: Further normalization of the data base relational model. In R. Rustin, editor, *Courant Computer Science Symposium 6*. Prentic-Hall, 1972.

[Cod72b] E. F. Codd. Database systems: Relational completeness of data base sublanguages. In R. Rustin, editor, *Courant Computer Science Symposium 6*. Prentic-Hall, 1972.

[Dat81] C. J. Date. *An Introduction to Database Systems, 3rd Edition*. Addison-Wesley, Reading, MA, 1981.

[DKA+86] P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended nf^2 relations: An integrated view on flat tables and hierarchies. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 356–366, Washington, 1986. ACM.

[DMB+87] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying complex objects: The PROBE approach to modelling and querying them. In H. J. Schek and G. Schlageter, editors, *Proc. GI Conf. on Database Systems for Office, Engineering an d Scientific Applications*, pages 17 – 37. Springer Verlag, 1987.

[DvG88] A. Deshpande and D. van Gucht. An implementation of nested relations. In *Proc. Int. Conf. on Very Large Databases*, pages 76–87, Los Angeles, 1988. Morgan Kaufmann.

[Fre87] J. C. Freytag. A rule-based view of query optimization. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 173 – 180, San Francisco, 1987. ACM.

[FT83] P. C. Fischer and S. J. Thomas. Operations on non-first-normal-form relations. In *Proceedings of IEEE COMPSAC'83*, pages 464–475, 1983.

[FvG85] P. C. Fischer and D. van Gucht. Determining when a structure is a nested relation. In *Proceedings of the International Conference on Very Large Databases*, pages 171–180, Stockholm, 1985.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 160 – 172, San Francisco, 1987. ACM.

[GP99] Peter Gulutzan and Trudy Pelzer. *SQL-99 Complete, Really*. R & D Books, 1999.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

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

[He97] Hongbo He. Implementation of nested relations in a database programming language. Master's thesis, McGill University, Montreal, Canada, 1997.

[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.

[Jia90] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, 1990.

[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, McGill University, Montreal, Canada.

[KK89] H. Kitagawa and T. L. Kunii. *The Unnormalized Relational Data Model for Office Form Processor Design.* Springer-Verlag, Tokyo, 1989.

[Lal86] N. Laliberté. Design and implementation of a primary memory version of aldat. Master's thesis, McGill University, Montreal, Canada, 1986.

[Lar88] P. Å. Larson. The data model and query language of LauRel. *IEEE Database Engineering Bulletin*, 11(3):23–30, September 1988.

[Lin90] V. Linnemann. Recursive functions in database language for complex objects. *Information Systems*, 15(6):627–645, 1990.

[LMP87] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 220 – 226, San Francisco, 1987.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 18 – 27, Chicago, 1988. ACM.

[LRV90] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2: An object-oriented data model. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems.* Morgan Kaufmann publishers, 1990.

[LS88] R. Lorie and H. J. Schek. On dynamically defined objects and SQL. In *Proceedings of the 2nd Workshop on Object-Oriented Database Systems*, 1988.

[LZ74] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, April 1974.

[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.

[Mar98] Angelica Valdivia Martinez. Implementing G.I.S. spatial operations in a database system. Master's thesis, McGill University, Montreal, Canada, 1998.

[MBC+01] 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. to be published, 2001. CS 617 course material of McGill University, Winter 2001.

[MD90] Frank Manola and Umeshwar Dayal. PDM: An object-oriented data model. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann publishers, 1990.

[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.

[Mer93] T. H. Merrett. Computations: Constraint programming with the relational algebra. In A. Makinouchi, editor, *International Symposium on Next Gereration Database Systems and Their Applications*, pages 12–17, Fukuoka, Japan, September 1993.

[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.

[MS90] David Maier and Jacob Stein. Development and implementation of an object-oriented dbms. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann publishers, 1990.

[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.

[OOM87] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987.

[OY85] Z. M. Ozsoyoglu and L. Y. Yuan. A normal form for nested relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 251 – 260, Portland, 1985. ACM.

[PA86] P. Pistor and F. Andersen. Designing a generalized nf^2 model with an SQL-type language interface. In *Proceedings of the International Conference on Very Large Databases*, pages 278–285, Kyoto,Japan, August 1986.

[PSS+87] H. B. Paul, H. J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt Database Kernel System. In *Proc. ACM SIGMOD Conf. on Management of Data*, San Francisco, 1987.

[RKB87] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: A query language for ¬1NF relational database. *Information Systems*, 12(1):99–114, 1987.

[RKS88] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

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

[Sch86] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proc. Int. Conf. on Database Theory*, pages 380 – 396, Rome, Italy, 1986. Springer Verlag.

[Sch89] H. Schöning. Integrating complex objects and recursion. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, pages 535–554, Kyoto, Japan, 1989. North-Holland.

[SDV01] 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, 2001. The web site contains documentation softwares for JavaCC and JJTree.

[Seb96] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, third edition, 1996.

[Ses98] Praveen Seshadri. Enhanced abstract data types in object-relational databases. *The VLDB Journal*, pages 130–140, July 1998.

[SH98] M. Stonebraker and J. M. Hellerstein. The roots. In M. Stonebraker and J. M. Hellerstein, editors, *Readings in Database Systems, 3rd Edition*. Morgan Kaufmann publishers, 1998.

[SP82] H. J. Schek and P. Pistor. Data structures for an integrated database management and information retrieval system. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 197–207, 1982.

[SPS87] M. H. Scholl, H. B. Paul, and H. J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137 – 146. Morgan Kaufmann publishers, 1987.

[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.

[SS86] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[SS87] M. H. Scholl and H. J. Schek. Theory and applications of nested relations and complex objects. In *International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, West Germany, 1987.

[Sto86a] M. Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of the 2nd IEEE Data Engineering Conference*, Los Angeles, 1986.

[Sto86b] M. Stonebraker. Object management in POSTGRES using procedures. In *Proceedings of the 1st International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986.

[Sto96] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers, 1996.

[Sun00] Weizhong Sun. Updates and events in a nested relational programming language. Master's thesis, 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.

[Ull82] J. D. Ullman. *Principles of Database Systems, 2nd Edition.* Computer Science Press, Rockville, MD, 1982.

[Wol89] A. Wolf. The DASDBS-geo kernel: Concepts, experiences, and the second step. In *Proc. Int. Symp. on the Design and Implementation of Large Spatial Databases.* Springer Verlag, 1989.

[WSSH88] P. F. Wilms, P. M. Schwartz, H. J. Schek, and L. M. Haas. Incorporating data types in an extensible database architecture. In *Proc. Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness.* Morgan Kaufmann publishers, 1988.

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

[Zam99] Saba Zamir. *Handbook of Object Technology.* CRC Press, 1999.

[Zha02] Hongyu Zhao. Implementation of QT-Expressions in a database programming language, 2002. Master's Report, McGill University, Montreal, Canada.