Design and Implementation of Database Computations in Java

Patrick Baker

School of Computer Science
McGill University, Montreal
July 1998

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

0-612-44121-0

Abstract

This thesis documents the design and implementation of computations, the procedural abstraction facility of a database programming language. Conceived as a special form of relation, computations fit neatly into the relational model. A computation is defined over a set of domains, which also act as parameters. Invocation is accomplished by means of the operators of the relational algebra. A computation is intended to embody a constraint amongst its parameters. These are grouped into inputs and outputs at call time. Given a set of input values, the computation returns output values that satisfy the constraint.

Computations may be recursive, and may also be nested within other computations. This latter property leads to a notion of scopes and environments.

With computations, a mechanism is also provided for the instantiation of stateful objects. It is designed to handle large numbers of such objects efficiently. Behaviour may be encapsulated within the object.

The implementation is part of the jRelix project at McGill University. jRelix is the successor of the Relix relational database programming language. A significant feature of the new implementation is the support for nested relations of arbitrary depth. The programming language Java was used exclusively for the implementation.

Résumé

Cette thèse documente la conception et l'implantation de computations, l'outil d'abstraction procédurale d'un langage de programmation pour les bases de données. Conçues comme étant une forme spéciale de relation, les computations s'insèrent nettement dans le modèle relationnel et sont définies sur un ensemble d'attributs qui agissent aussi comme paramètres. L'invocation est accomplie au moyen des opérateurs de l'algèbre relationnelle. Une computation devrait établir une contrainte parmi ses paramètres. Au temps d'appel ceux-ci sont regroupés en paramètres d'entrée et de sortie. Pourvue d'un ensemble déterminé de valeurs d'entrée, la computation délivre des valeurs de sortie satisfaisant la contrainte.

Les computations peuvent être récursives ou imbriquées dans d'autres computations. La dernière propriété conduit à une notion de visibilité des données et d'environnements.

Avec les computations, un mécanisme est inclus pour la création d'objets étatiques. Il est conçu pour en traiter efficacement un grand nombre. Des comportements peuvent être encapsulés dans l'objet.

L'implantation fait partie du projet jRelix de l'université McGill. jRelix est le successeur du langage de programmation de la base de données relationnelle Relix. Un trait significatif de la nouvelle implantation est le support des relations imbriquées à profondeur arbitraire. Le langage de programmation Java fut exclusivement utilisé pour le projet.

Acknowledgments

I wish to express my gratitude to my supervisor, Professor T.H. Merrett. Our discussions have guided me throughout the course of this project. As well, this thesis would not have its present form if it were not for his assiduity in reviewing the earlier drafts. To him I happily acknowledge my obligation.

My teammates on this project were also very helpful. Opinions, suggestions and criticisms were exchanged at several meetings. Biao coded most of the parser and designed the structure of the interpreter. Also, he has kindly allowed me to reproduce the BNF grammar for jRelix from his thesis. Zhongxia wrote the source code for handling the system relations, including moving them between memory and disk.

I would also like to thank Xiaoyan Zhao for her assistance with the ALDAT lab facilities. In addition, she sat in on many group meetings at the beginning of the project to offer advice on the internals of Relix.

I wish to thank my parents for their support throughout my studies. They have always encouraged and supported me in my endeavours. My mom also helped me to translate the abstract into French. This thesis is dedicated to them.

Finally, I wish to thank Suzan for her love, encouragement and patience. I love you very much.

Contents

Abstract i

Résumé ii

Acknowledgments iii

1 Introduction 1

1.1 Aim of the Thesis 1

1.2 Survey of Database Programming Languages 2

1.2.1 Language-oriented database systems 3

1.2.2 Persistent programming languages 8

1.2.3 Object-oriented languages 11

1.2.4 Knowledge-base management systems 15

1.3 Motivation for the Thesis 19

1.4 Outline of the Thesis 21

2 Overview of jRelix 22

2.1 Relations and Domains 22

2.2 Relational Algebra 27

2.2.1 Unary operators 28

2.2.2 Binary operators 31

2.3 Domain Algebra 34

2.3.1 Horizontal operations 34

2.3.2 Vertical operations 37

2.4 Update Statement 40

2.5 Nested Relations 41

3 User Manual on Computations 46

3.1 Creating Computations 46

3.2 Invoking Computations 48

3.2.1 Invoking a computation with a select expression 49

3.2.2 Array syntax for computation invocation 49

3.2.3 Join of a computation with a relation 50

3.2.4 Stand-alone invocation of computations 52

3.2.5 Multivalued computations 53

3.2.6 Properties of computations 53
3.3 Stateful Computations 54
3.4 Recursive Computations 56
3.5 Constraint Verification 58
3.6 Commands 59

4 Implementation of Computations 61
4.1 Overview of the jRelix Implementation 61
4.1.1 Data dictionary 62
4.1.2 Representation of nested relations 65
4.1.3 Storage of relations 67
4.2 Parsing a JJTree Syntax Tree 69
4.3 Environments in jRelix 71
4.3.1 The environment model 72
4.3.2 Creating objects with state 80
4.3.3 The static environment model 82
4.3.4 Implementation details 85
4.3.5 Scopes induced by nested relations 88
4.4 Application of Computations 91
4.5 Creation of Computations 94
4.6 Implementation of Stateful Computations 99
4.7 Storage of Computations 101
4.7.1 Top-level computations 103
4.7.2 Computations nested in a relation 103
4.8 Recursion 105

5 Conclusion 106
5.1 Summary 106
5.2 Future Work 107
5.2.1 A note on the commutivity of the natural join 107
5.2.2 Natural join of two computations 108
5.2.3 Overloading other relational algebra operators 109
5.2.4 Events and triggers 109
5.2.5 Block algorithm: select clause 110
5.2.6 Storing a view in a relation 111
5.2.7 Object-oriented features 112

A Backus-Naur Form for jRelix Commands 116

Bibliography 122

List of Figures

1.1 Datalog program 18
1.2 Expressing a constraint with a relation, part 1 20
1.3 Expressing a constraint with a relation, part 2 20

2.1 A flat relation 23
2.2 Domain declarations 24
2.3 Command sd 25
2.4 Creating relations 26
2.5 Commands sr and pr 26
2.6 Assignment and incremental assignment 27
2.7 Projection 28
2.8 Selection 29
2.9 T-Selection 30
2.10 Relation orders 33
2.11 Natural join 33
2.12 Actualization of a virtual domain, part 1a 35
2.13 Actualization of a virtual domain, part 1b 35
2.14 Constant virtual domains 36
2.15 Unary virtual domains 36
2.16 Binary virtual domains 36
2.17 Conditional virtual domains 37
2.18 Predefined functions in virtual domains 37
2.19 Reduction 38
2.20 Actualization of a virtual domain, part 2a 39
2.21 Actualization of a virtual domain, part 2b 39
2.22 Equivalence reduction 39
2.23 The domain algebra 40
2.24 Update statement 42
2.25 A nested relation 42
2.26 Nested domain declaration 44
2.27 Creating a nested relation 45
2.28 Nested domain algebra 45

3.1 Velocity computation 47
3.2 Velocity computation as a relation 47
3.3 Sum of squares computation 48

3.4 Applying the velocity computation with a selection 49
3.5 Array syntax for computation invocation 50
3.6 Applying the velocity computation with an ijoin 51
3.7 Stand-alone computation call 52
3.8 Also operator 53
3.9 Declaration of a bank account class 55
3.10 Instantiation of bank account objects 56
3.11 Transfer of money between bank accounts 56
3.12 Recursive computation 57
3.13 Constraint verification with a computation 58
3.14 Commands for computations 60

4.1 System relations 64
4.2 In RAM system metadata structures 66
4.3 Implementation of nested domains, part 1 67
4.4 Implementation of nested domains, part 2 68
4.5 Tools used to create the jRelix parser 70
4.6 JJTree syntax trees 70
4.7 Debug mode 71
4.8 Variable look ups in an environment 73
4.9 Max computation, part 1 74
4.10 Max computation, part 2 75
4.11 Static binding 77
4.12 Dynamic binding 77
4.13 Sum of squares, part 1 78
4.14 Sum of squares, part 2 78
4.15 Sum of squares, part 3 79
4.16 Sum of squares, part 4 79
4.17 Bank account computation, part 1 81
4.18 Bank account computation, part 2 81
4.19 Bank account computation, part3 82
4.20 Static environment, part 1 83
4.21 Static environment, part 2 84
4.22 Static environment, part 3 84
4.23 A deeply nested relation 90
4.24 Nested-relation environment 90
4.25 Block type 94
4.26 Nested Computation 98
4.27 Hidden state variables, part 1 101
4.28 Hidden state variables, part 2 102
4.29 Storage of a computation nested in a relation 104

5.1 Computation that does not embody a constraint 107
5.2 Some musicians 114
5.3 Musician objects 115

List of Tables

1.1 PS-algol procedural interface to the persistent store 10
1.2 The golden rules 12
1.3 Features of ObjectStore, O_2, and ONTOS 16

2.1 Primitive types in jRelix 24
2.2 Summary of μ-joins 32
2.3 Summary of σ-joins 34

4.1 System relations 62
4.2 Component structures of the Environment class 86
4.3 Classes encapsulating the different kinds of variables 87
4.4 Interface methods of the Environment class 89

5.1 Ambiguous block types 110

A.1 BNF convention 116

Chapter 1

Introduction

1.1 Aim of the Thesis

This thesis documents the design and implementation of a procedural abstraction mechanism for the jRelix relational database programming language. jRelix consists of a database management system (DBMS) together with the query language Aldat—ALgebraic DATa language. The work is based on a previous version, called Relix, that was written using the C programming language [38]. Over the years, many features were added to this base implementation: computations—a mechanism for procedural abstraction, managing constraints and modeling objects; procedures—which provide a different procedural abstraction facility [43], as well as a method to define event handlers [26]; support for nested relations with at most one level of depth [31]. One of the motivations for building jRelix is to integrate these features neatly in a new system. Experience with Relix has led to new ideas for the design of the language. This also provides incentive for a fresh implementation. One of the features that has been redesigned in jRelix is the full support for nested relations of arbitrary depth. This has profound implications for the implementation of the relational algebra as well as the domain algebra. It also permits the unification of computations and procedures, thereby leading to a simpler language. Computations have also been redesigned to enable the modeling of objects in a more manageable and efficient manner. The programming language Java was used for the implementation of jRelix. It may be used on any system which supports the Java Run-time Environment (JRE). Such an environment is included in today's most popular web browsers. The combination of Java's graphical user interface generation tools and networking facilities make possible the development of a Java front end to jRelix that will make it readily available to a large number of people over the Internet.

The full documentation for jRelix is split across three theses. Although there was much collaboration, especially in the design of jRelix and the implementation of the core of the DBMS, the work was roughly divided as follows. The relational algebra was implemented by Biao Hao and is documented in [30]. The domain algebra was the responsibility of Zhongxia Yuan, details of which are to be found in [70]. Finally, computations are the subject of this thesis.

1.2 Survey of Database Programming Languages

The relational model, first introduced by Codd [21], has attracted much attention both in the academic world and in industry. This has led to several research and commercial implementations. It has proven itself exceptionally useful for many business applications. The basic model, however, is lacking in expressive power and in the ability to handle complex data. Many applications are arising in the science and engineering fields for which the relational model, as originally proposed, is not an adequate tool. This has fueled the need for continued research in the field of database programming languages (DBPL).

The goal is to create programming languages which ease the development of large applications which are characterized by a combination of the following properties:

- involve large amounts of data
- require the data to persist
- involve complex structured data
- allow concurrent access to the data
- involve complex operations on the data

Computer automated design (CAD) and VLSI chip design are examples of such applications.

DBMS are capable of dealing with large amounts of persistent data and allowing concurrent access to it even if it is distributed among several sites. Programming languages provide well proven and powerful techniques for creating, organizing and manipulating data that is in memory. Database programming languages seek to combine these two aspects in order to provide a programming environment suitable for the development of applications of the type just described. In this section, a survey of attempts to create such languages is presented. In [13], F. Bancilhon identifies four classes of DBPL. These are

- language-oriented database systems
- persistent programming languages
- engineering database systems
- object-oriented database systems

A fifth class should also be included

- knowledge-base management systems

Although we do not discuss engineering database systems, the organization of this survey is based on the other four classes.

1.2.1 Language-oriented database systems

An early approach to creating a DBPL has been to embed a database query language into an existing programming language. An example of this is the INGRES relational database system [64]. Its query language, called QUEL, resembles tuple relational calculus in which variables represent tuples of a relation. QUEL has been embedded into the C programming language yielding EQUEL [4]. This was accomplished by means of a precompiler. QUEL variables and statements are inserted into a C program in lines that begin with '##'. The precompiler only takes action on these lines. The other lines of code are output without modification. If a line beginning with '##'

contains a variable declaration, then this is noted for future use. If it is a QUEL statement, then the INGRES parser is invoked on it. If this statement does not contain any C variables then code is inserted to handle the query. If however a C variable appears in the statement then its value and type must be passed to the INGRES lexical analyzer, and then the rest of the line is passed to INGRES as well. C code is then generated for this statement.

A major disadvantage of this approach is that it requires the programmer to be fluent with both the host language as well as the query language. It is unwieldy to make the bulk types of the query language, like the relation, fit together with the typing system of the host language. This difficulty in integrating the host language with the query language has been referred to as *impedance mismatch*. This approach therefore yields an awkward programming environment, and more integrated solutions are needed. An other example of this kind of language is System/R [5] which combines SEQUEL and PL/1.

Another approach to creating a DBPL is to add database features to an existing programming language. A popular and widely used language in this class is Pascal/R [59]. It is an attempt at combining the relational data model with the Pascal programming language [33]. The **record** type of Pascal is used to represent a tuple of a relation. The constructor **relation of** has been added to the language. It takes as arguments a record type, which effectively specifies the schema of the relation, and a key for the relation, which must be a subset of the fields in the record. The **database** constructor is used to specify the relation types of the relations that make up a database. A new iteration construct, **for each**, is added for looping over the tuples of a relation. Also, there is a set of four operators that permit the traversal of a relation—**low**, **next**, **high** and **eor**. Two other mecanisms are available for manipulating relations. First, relational operators are provided by the system, and secondly, procedures may have relations as their arguments.

Some criticisms [11] of the Pascal/R language are that the **for each** operator does not apply to the other bulk types that are provided in Pascal—the **file**, the **array** and the **set**; only a single database may be used by any program; persistence only applies to specific types—such as **file** and **relation**; there is a lack of support for concurrency control and transaction management.

The language DBPL [44, 58, 45] is a successor of Pascal/R. It is an extension of the Modula-2 programming language [69] (which is itself a successor of Pascal) which permits the modeling of a relational database. Although the language supports arrays, records and variant records, the most important bulk data type is the set. All elements of a set must have the same type, and the number of elements in a set may vary over time. Sets may have a key associated with them. These keyed sets have the property that no two elements of the set can share the same key value. Relations may be modeled as keyed sets of flat records. Nested relations may also be modeled by a combination of these type constructors. First-order predicates may be used for manipulating sets. These predicates are a part of *selectors* and *constructors*. An example of a selector is:

```
EACH e in S: p(e1,...,en)
```

This specifies the subset of S whose elements satisfy the predicate p. This may be combined with the iteration construct **FOR** as follows:

```
FOR EACH e in S: p(e1,...,en) DO ... END
```

The constructors extend the notion of selectors and provide the power of a deductive database [45]. DBPL is a type complete language. The bulk types that were mentioned above can consist of elements of any type, and there are no restrictions on how some type may be used. For example, any type may be passed to a procedure. As well, persistence is orthogonal to type. The implementation of persistence is based on the module concept of Modula-2. The variables that are in the outermost scope of a module that is declared as DATABASE are persistent. DBPL also has facilities for concurrency control and transaction management, and can access multiple databases in a single program. DBPL programs are strongly typed and statically checked. This helps the programmer by signaling errors at compile time instead of at run time. Thus, many of the weaknesses of Pascal/R have been resolved in DBPL.

Database programming languages have also been created by extending a relational database system. The language RIGEL [56] is built on top of INGRES. Its type system includes relations, views and tuples. In order to manipulate these, the notions of *bind* *expressions* and *generators* have been singled out as fundamental. The following example will be used to explain these concepts.

```
FOR T in RELATION WHERE T.ATTRIB = "CONST" DO ... END
```

Here, the generator is the 'WHERE' clause. It generates a sequence of values. The bind expression is 'T in RELATION'. It binds the iteration variable, T (a tuple variable), to the sequence of values provided by the generator in turn. There is also the colon operator.

```
T.ATTRIB : T in RELATION
```

The right operand is a bind expression. In this case it binds the tuple variable T to each tuple of RELATION in turn, although more complicated expressions may be used. The left operand is an expression specifying the value to be returned by the operator. In the example, the ATTRIB field of RELATION will be returned for each binding of the iteration variable T. The whole expression is then a generator as it produces a sequence of values. This is convenient as it allows relations to be passed as arguments to a procedure. This is actually accomplished by passing a generator to the procedure. The procedure can then access the sequence of values which the generator produces. This is in fact similar to the way DBPL works. The predicates used in that language are generators. This general notion of bind expressions and generators leads to a consistent form of iteration in RIGEL. Loops over relations. arrays and files are all constructed by means of these two mechanisms.

RIGEL supports a data abstraction facility that is based on the modules concept of Modula [69]. The module has three sections. The public section is where the interface of the module is specified. The implementation of this interface is in the private section. There is also an intialization section. Updates and views are also available. Views may be updated, but this is limited and requires extra effort on the part of the programmer. The ambiguities arising in view updates are well recognized [24], however, the designers of RIGEL considered this to be a useful feature that should not be omitted.

The POSTGRES DBPL [63, 62, 65] is an extension of the INGRES relational database system. Its query language POSTQUEL derives from QUEL. The system is designed to address a class of applications which involve large amounts of data, complex structured data and sophisticated interrelationships amongst the data. In order to service these requirements POSTGRES contains data, object and knowledge management facilities.

The POSTQUEL query language extends the functionality of a relational query language. It supports user defined functions and operators, abstract data types, arrays, transitive closures and time travel.

The types that are available in POSTGRES are the base types (integers, character strings, etc.), arrays of base type elements and composite types. An abstract data type facility allows the programmer to define new base types. The POSTGRES data model is based on the notion of a class. A class is created by specifying a name for it and a list of attributes and their types. Each object has a unique and permanent identifier assigned to it. The keyword **inherit** may be used to indicate that the attributes of another class be included as attributes of the new class. Multiple inheritance is supported. There are three types of classes. Real classes have instances that are stored in the database. Instances of derived classes are not actually stored in the database, but are calculated when needed. This is how views are implemented in POSTGRES. Instances of a versioned class are only stored differentially, with respect to the parent class, in the database.

Many object-oriented features are present in POSTGRES. Classes and inheritance have already been mentioned. Behaviour is not encapsulated within the class however. A method is a function that has a class name as an argument. It becomes a method of the class indicated by its argument, and is available to all the subclasses of that class.

POSTGRES also has a rule management system. This is used for a variety of purposes including enforcing integrity constraints, view management, triggers, and version control.

The POSTGRES storage system does not use write-ahead logging. Instead, it uses a *no-overwrite* storage manager. Updates do not erase the previous records which remain in the database. The system has to make a distinction between records that are actually part of the database, and those that have resulted from aborted transactions. This approach simplifies crash recovery since all the records are still in the database. There is no need to access records from a log file. A problem with this approach is that at the end of every transaction the affected memory pages must be saved to disk in case of a system failure. This is more expensive than writing out to a log file. The no-overwrite storage manager admits the possibility for *time travel*. This is the ability to pose queries on historical data previously contained in the database. The historical data is actually still in the database because of the no-overwrite architecture. These historical records are maintained seperately from those of the 'current' database in a history database.

1.2.2 Persistent programming languages

It is very common for a computer application to require some data to persist beyond its execution. If it is only a small amount of data that needs to be saved, then the file system may be used. If the application deals with large amounts of data, then a database management system may ease the burden of its storage and retrieval. A problem with these approaches is that the conventional programming language constructs (e.g arrays and records), do not correspond to those for persistent storage. (e.g the abstraction of a file or of a relation). The programmer must map the data from the forms used in primary memory to those used on the persistent store. It has been estimated that about 30% of the code in a typical application is dedicated to such tasks [6]. Another disadvantage of this translation is that the typing mechanisms provided by the programming language to aid the developer are usually lost across this mapping. Persistent programming languages obviate this problem by eliminating the need for this mapping. In [6], the following properties of persistent data are tabulated.

1. persistence independence: the persistence of a data object is independent of how the program manipulates that data object.
2. persistence data type orthogonality: all data objects should be allowed to persist.
3. The choice of how to provide and identify persistence at a language level is independent of the choice of data objects in the language.

The second property is a consequence of the principle of data type completeness which is essentially that all data types have equal rights.

The first successful persistent programming language to be developed is PS-algol [52, 7, 6] which derives from S-algol [51, 22]. The data types that are available in PS-algol are the usual primitives such as *int* and *real* (a *pic* type is also included for two dimensional drawings) as well as the composite types *vector* and *pntr*. A *vector* is a dynamically resizable array which can hold elements of a single but arbitrary type. A *pntr* is an infinite union type of all *structures*. It is essentially an untyped pointer. A *structure* consists of a set of fields in which data of any type (including procedures which are first-class types in PS-algol) may be stored. When a *pntr* is dereferenced it is projected onto a particular type. The system checks dynamically at run time to make sure that this type corresponds to that asserted by the program. This technique allows the rest of the program to be typed checked statically.

The implementation of persistence in PS-algol is accomplished with minimal change to the S-algol compiler. The functionality is achieved by a collection of procedures. These provide facilities for opening and closing a database, for transaction management and for associative lookup of variables. Table 1.1 on the following page summarizes this interface. A *table* is a directory structure that stores pointers to objects in the persistent store indexed by their name. There can be many of these tables, and concurrent access to them by several programs is permitted. Data automatically moves between memory and the persistent store. The system deduces which objects persist by reachability.

As previously mentioned, procedures in PS-algol are first-class. Thus, they may be included in structures. In this way, code, as well as data, may be moved to the persistent store. Since the PS-algol compiler is itself a procedure, it may also be placed in the store. This admits the possibility for *linguistic reflection*—the ability of a running program to create new program fragments and execute them. Source code for a procedure can be passed to the compiler procedure. It will return a compiled version of the procedure that may then be used by the currently executing program. Continued research with orthogonally persistent languages has led to the discovery of new paradigms that can not be made available by conventional programming languages. We mention the notions of *hyper-programming* and *persistent schema evolution*, but do not investigate them. The reader is referred to [54] for further details. The language

Procedure	Inputs	Return	Description
	Basic functionality		
open.database	string name string mode string user string password	pntr	open a database
close.database	pntr root	n/a	close a database
get.root	pntr database	pntr	get the root of the db
set.root	pntr old.root pntr new.root	n/a	set the root of the db
commit	n/a	n/a	commit the transaction
abandon	n/a	n/a	abandon the transaction
	Associative lookup		
table	n/a	n/a	create an empty table
lookup	pntr table string name	pntr	look up a variable in a table
enter	pntr table pntr value string key	pntr	add an entry to a table
scan	pntr table pntr value proc user	pntr	apply a procedure to every entry in a table

Table 1.1: PS-algol procedural interface to the persistent store

Napier88 [25, 53] is a successor to PS-algol in which all of these advanced features are available.

The programming language Java has recently gained immense popularity. Incorporating persistence into Java would make this technology available to a very large number of programmers. Not surprisingly then, the possibility of adding persistence to Java has been investigated [8]. A number of properties must be present in a programming language in order for it to be amenable to persistence. In [54] it has been indicated how these are indeed provided by Java. The properties listed in this article are

- "persistent store with roots"
- "reachability and referential integrity"
- "code as data"
- "an infinite union type with dynamic injection and projection"
- "two type magic procedures"
  - "one to find a type representation of a value"
  - "one to convert a sequence of bytes into a language value"

The project concerned with the implementation of persistent Java, originally called PJava, is now called PJama. A prototype implementation is currently available.

1.2.3 Object-oriented languages

An object-oriented database management system (OODBMS) is a programming system which incorporates the facilities of a DBMS with the object-oriented computing paradigm. In almost every case, an OODBMS is a persistent object-oriented programming language. Due to the copious amount of literature devoted to this topic, we treat it separately here. In *The Object-Oriented Database System Manifesto* [10], the 'golden rules' outlining the properties that an OODBMS should have are stated. These rules may be grouped into two sets; those which describe DBMS features, and those which address object-oriented features. These are shown in table 1.2. There

DBMS features	Object-oriented features
Thou shalt remember thy data	*Thou shalt support complex objects*
Thou shalt manage very large databases	*Thou shalt support object identity*
Thou shalt accept concurrent users	*Thou shalt encapsulate thine objects*
Thou shalt recover from hardware and software failures	*Thou shalt support types or classes*
Thou shalt have a simple way of querying data	*Thy classes or types shall inherit from their ancestors*
	Thou shalt not bind prematurely
	Thou shalt be computationally complete
	Thou shalt be extensible

Table 1.2: The golden rules

are some additional rules and suggestions as well. In particular, we mention the very last rule of the manifesto: *Thou shalt question the golden rules.*

The ObjectStore [39] OODBMS is based on the C++ programming language. It is available as a C++ library interface, but ObjectStores's extended C++ compiler is a more tightly integrated version[1]. This system offers all of the features that are to be expected from a DBMS—persistent storage, query facility with internal optimization, transaction management, distributed data management, and provisions for ensuring data integrity. Access to persistent data is seamless to the programmer. This is because the system was designed so that persistence is orthogonal to type. Indeed, not only objects and structures[2] (which may contain pointers to other entities), but even primitive data such as integers, may persist. The movement of data between memory and the persistent store and the necessary locking involved is handled transparently by the system. The keyword **persistent** is provided to qualify the storage class of a variable as persistent.

```
persistent<db> hockeyteam* montrealCanadiens;
```

In this example the variable *montrealCanadiens* is persistent and belongs to the database pointed to by *db*. This variable may then be manipulated in subsequent code in the same manner that variables are usually handled in regular C++ programs. This has many favourable consequences: procedures do not have to be concerned with whether they are accessing persistent or transient data, the same code will work for

[1]A C library interface to ObjectStore is also available.

[2]The 'structure' is the C equivalent to 'records' in Pascal.

both; no translation is necessary to bring in data from the persistent store; there is a single type system that is statically checked. The resulting close integration with the C++ programming language also yields its share of benefits: the system is easy to learn for programmers already familar with C++; the language is computationally complete; existing C++ libraries can access persistent data, often even without the need to recompile.

ObjectStore offers a number of bulk types for managing large amounts of data. These are provided via a class library, and offer a high-level interface. These collections include ordered lists, sets and bags. For better performance, structures such as hash tables and B-trees may be used. However, this can be handled transparently by the library. It will pick an appropriate representation in response to hints provided by the programmer as to the size and usage patterns that are expected. The representation may even change over time in response to an increase in the size of the collection. The programmer, once again, must provide hints for this to happen. The class library that provides this functionality consists of two class hierarchies. One of these contains class definitions for lists, sets and bags. Methods are available for operating on these structures in the usual ways. The second class hierarchy specifies the representations (hash table, B-tree, etc.) of the interfaces found in the first hierarchy. Although these classes are available to the programmer, it is intended that the system pick the most appropriate respresentation of a type based on hints that are provided by the user. The **foreach** iteration construct is provided for sets.

Queries in the extended C++ version of objectStore are contained between the delimiters '[:' and ':]'. These may be nested arbitrarily. Selection predicates consist of boolean conditions that are placed within these delimiters. For example,

```
superStars = montrealCanadiens [: points > 100 :];
```

An important goal of the ObjectStore system is efficiency. This applies, in particular, to a class of applications that deal with large amounts of data but only in small chunks. The repetitive accessing of data in small quantities followed by computation is known as *fine-interleaving*. It is required that such access to data be as fast for persistent data as it is for transient data. This problem is essentially that the dereferencing of a pointer should be fast regardless of the type of the data that is pointed to. In order to accomplish this, it is necessary that the code for this operation compile to the same machine instructions regardless of the data's persistence. This is done by taking advantage of the virtual memory system provided by the underlying operating system. ObjectStore marks certain pages of the process' address space so as to not allow access. When the program attempts to dereference a pointer to persistent data, a memory fault occurs. This is caught by ObjectStore which then retrieves the data and places it in a cache, which is also in the process address space. The next time that the pointer is dereferenced, the targetted object will be found in the cache. As mentioned, this operation compiles to the same machine code, essentially a load instruction, regardless of the level of persistence of the data.

A few of the other features included in ObjectStore will now be mentioned briefly. Transactions are specified by the programmer by means of the **begin** and **commit** methods and provide all-or-none semantics. Locks are handled transparently by ObjectStore. Modifications are logged to permit recovery in the case of a system failure. Support for versioning is available. It is orthogonal to type. This implies that any instance of any object may be versioned. As well, the manipulation of such a variable is carried out in the usual way. The same code will work for both versioned and non-versioned data.

Another full-featured OODBMS is the O_2 system [27, 12]. At the core of the system is the O_2Engine which is responsible for the storage and management of objects. Two types of interfaces permit access to the engine. Language interfaces allow access via regular programming languages. C, C++, Basic and Lisp are supported at this time. The other interface is the O_2 environment.

The O_2Engine has a three layer architecture. The upper layer is the schema manager. It is responsible for managing classes, methods and global names. The middle layer is the object manager. It includes a mechanism for the garbage collection of objects and for determining which objects are persistent through reachability. It also handles indexes and clustering of objects for efficient storage. The bottom layer is the storage manager, which is based on the Wisconsin Storage System [20].

The O_2 environment consists of the O_2C programming language, the O_2Query query language and the O_2Look user interface generator. The query language is similar to SQL. Queries consist of three parts—a *select* clause specifying which attributes of a class are wanted, a *from* clause indicating the objects the query is on and a *where* clause that filters the result based on a provided predicate. The O_2C programming langauge is a complete programming language that includes O_2Query as a subset. User interfaces can be rapidly developed with O_2Look, often with a single statement. Ready-made and customizable presentations ease this task considerably for the programmer. A graphical programming environment called O_2Tools is also included. This is itself an O_2C application. It provides browsers for the database and schema (i.e. the class definitions) as well as a powerful debugger.

The O_2 data model consists of values and classes. Values are the primitive types, but the constructors *tuple*, *list* and *set* can be used to construct new types. Both values and classes may persist, and a variable can be made persistent at any time. The manipulation of a data object is independent of its persistence.

Encapsulation is provided at three different levels. There is the usual notion of encapsulation for classes in which the behaviour of a class is specified by methods belonging to that class. There is also a notion of schema encapsulation. A schema is just a set of class definitions. Some of the classes in a schema may be exported (made public) so that they can be used in other schemas. This is for large scale programming. The final form of encapsulation applies to databases. A method's definition may require it to run against a database that is not the current database being used by the application. Interoperability between heterogenous databases is based on this feature.

Several other OODBMS have been implemented. For completeness, we mention some of the more prominent ones, but without investigating them: GemStone [55], Iris [28, 29], Ontos [60], Orion [35, 37], and Starburst [42]. Table 1.3 on the next page is taken from [60]. It summarizes the main features of the ObjectStore, O_2 and ONTOS systems.

1.2.4 Knowledge-base management systems

Knowledge-base management systems (KBMS) combine the traditional features of a DBMS with the logic programming paradigm. In sharp contrast to the imperative

Features	ObjectStore	O_2	ONTOS
page server architecture	+	+	+
SQL-like interface	-	+	+
graphical schema designer	+	+	+
graphical browser	+	+	+
graphical data editor	-	+	+
debugger	+	+	-
C++ interface	+	+	+
easiness of existing C and C++ program migration	+	-	-
persistence at the level of objects rather than at the class level	+	+	-
metaclass support	-	-	+
indexing	+	+	+
inverse data members	+	-	+
explicit object deletion instead of garbage collection	+	+	+
dynamic adding of new classes	-	+	+
data aggregate support	+	+	+
query optimization	+	-	-
conventional transactions	+	+	+
nested transactions	+	-	+
long transactions	+	-	-
optimistic transactions	-	-	+
fault recovery	+	+	+
cooperative group model	+	-	-

Table 1.3: Features of ObjectStore, O_2, and ONTOS

style of programming, in which one specifies *how* to compute a task, logic programming involves declaring *what* is to be computed. The logic programming language Prolog has achieved considerable popularity. Attempts have been made to combine database features with Prolog [18, 14, 32]. The usual approach taken is to create an interface between Prolog and a relational DBMS. This typically involves tuple-by-tuple interactions between the interpreter and the database, and is therefore rather inefficient [16]. More tightly integrated systems are required in order for KBMS to become commercially viable. The language Datalog [17] is a database query language based on Prolog, but designed to be more amenable to integration with databases.

A Datalog program consists of a set of facts and rules. An example, taken from [67, page 103], is shown in figure 1.1 on the following page. Items such as *parent*(X, Y) are called (positive) literals. In this case, *parent* is a predicate symbol, and X and Y are terms. A literal may be negated (e.g. $\neg$*parent*(X, Y)). Each of the statements shown in the example are known as Horn clauses[3]. The portion to the left of the ':-' symbol is known as the *head* of the clause, and that to the right as the *body*. The clause is interpreted as follows: 'if *body* then *head*'. The meaning of the first clause in our example is thus that X and Y are siblings if there is a Z such that Z is the parent of X, and Z is the parent of Y, and X is not Y[4]. A Horn clause that has a non-empty body is known as a rule. If the body is empty, then the clause is a fact. Facts and rules are forms of knowledge, and it is these that comprise a Datalog program.

Datalog was developed for use with relational databases. In our example, the predicates *sibling* and *cousin* are derived from *parent* which must be known. The *parent* predicate is stored in a relation of the same name. Every tuple of this relation represents a fact indicating a child-parent association. Predicates which are stored explicitly in the database are called extensional database predicates. In contrast, the *sibling* and *cousin* predicates are not stored in the database, but are computed when

[3]Formally, a Horn clause is the logical disjunction of literals, at most one of which is positive. It is easily seen that the clause $\neg p_1 \vee \cdots \vee \neg p_n \vee q$ is logically equivalent to $p_1 \wedge \cdots \wedge p_n \rightarrow q$.

[4]Note that we have implicitly assumed that the variable Z, which appears only in the body of the clause, is existentially quantified. In fact, in this case it is also correct to assume that all variables are universally quantified. In general, however, the rules governing the quantification of variables can affect the semantics of a logic program [41].

```
sibling(X,Y) :- parent(X,Z) & parent(Y,Z) & X ≠ Y.
cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) & sibling(Xp, Yp).
cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) & cousin(Xp, Yp).
```

Figure 1.1: Datalog program

necessary. These are called intensional database predicates. This is similar to the distinction between relations and views in a relational system.

Datalog is a simplified logic programming language. Certain features have been left out so that the language could be optimized for use with large quantities of data. Many of these features have reappeared in 'extensions' to datalog which we now consider. One of these is in fact included in our example—built in predicates such as $X \neq Y$. These may be thought of as extensional database predicates, although instead of being contained in the database, they must be computed. This presents a safety problem. It is necessary that the set of derivable facts be finite. In order to guarantee this, some restrictions—'safety rules'—are placed on the variables in clauses with built-in predicates.

Another extension to Datalog is the use of negated predicates in the body of a clause. These are therefore no longer Horn clauses. This also raises a safety problem as the complement of a finite set need not be finite. Indeed, it is not always clear with respect to which universe the complement is to be taken. Datalog$^{\neg}$, Stratified Datalog$^{\neg}$ and Inflationary Datalog$^{\neg}$ are three different systems which attempt to deal with negation in Datalog [17].

One of the differences between Prolog and Datalog is that the terms appearing in a literal may only represent variables and constants. In Prolog, a term may also be a function. An extension to Datalog is to allow function symbols. The LDL system, for example, supports this.

Finally, as previously mentioned, support for complex types is often required by modern applications. Thus another extensions to Datalog is to include bulk type constructors that may be used as terms. In [3], an object-oriented extension to Datalog is presented. The features included are methods, classes, inheritance, overloading and late binding.

It has already been noted that POSTGRES includes facilities for logic programming. Three other existing database systems that use logic as a query langauge are NAIL! [50], LDL [19], and LOGRES [15].

1.3 Motivation for the Thesis

The computation is the procedural abstraction mechanism provided by the relational database programming language jRelix. The purpose of this thesis is to document their design and implementation. In order to minimize the number of different concepts and to keep the language unified, computations are designed to resemble relations as closely as possible.

A computation may be thought of in two different ways. Firstly, it may be regarded as a procedure having a set of parameters and local variables which executes a block of code when called. This view, however, obscures the similarity between computations and relations. An alternative perspective is to regard a computation as expressing a constraint [48]. As a simple example, consider the constraint expressed by the following:

APHORISM: *Happiness* IS *Peace* AND *Quiet*[5].

Here, *Happiness*, *Peace* and *Quiet* are boolean variables. Given values for any two of these, the constraint implies that of the third. Another way of expressing this is to list all assignments of these variables that satisfy the given constraint, as shown in figure 1.2 on the next page.

Thus, the aphorism can be expressed either by a rule, or by a table. Herein lies the idea which links the two different views of a computation. When creating a computation, the programmer actually writes a block of code. But by thinking of the code as embodying a constraint, the computation may be thought of as a table, that is, as a relation. In the following chapters it will be shown how the relational operators

[5]This aphorism is, of course, inspired by the great moral teachings of Snoopy.

APHORISM	(Happiness	Peace	Quiet)
	True	True	True
	False	True	False
	False	False	True
	False	False	False

Figure 1.2: Expressing a constraint with a relation, part 1

PRIMES	(N	P)
	1	2
	2	3
	3	5
	4	7
	5	11
	⋮	⋮

Figure 1.3: Expressing a constraint with a relation, part 2

have been extended to allow their arguments to be computations. Indeed, the jRelix syntax for manipulating relations and computations coincide.

In the previous example, the table corresponding to the constraint was finite. This need not be so. Consider the following:

PRIMES: P is the N^{th} prime.

The table for encapsulating this constraint is infinite. It is shown in figure 1.3. Note that given a value for N, the corresponding value for P (the N^{th} prime) may be deduced, and vice versa.

A call to the computation PRIMES could pass as inputs a set of prime numbers. The computation would then return the indices into the sequence of prime numbers corresponding to those provided. Alternatively, if a set of integers is passed to the computation then the corresponding prime numbers are returned. Hence, computations may be used in several ways—they are symmetrical with respect to their parameters. Any subset of the parameters may serve as inputs for a given computation call. The complement set of parameters will be calculated by the computation. In practice, however, sufficiently many input parameters must be provided so that the output values may be deduced from the constraint embodied by the computation.

Computations have also been used to model states and instantiation. A subset of the features available in object-oriented systems are present. The model of instantiation differs from that offered by object-oriented programming languages. It is based on the natural join operator of the relational algebra. Behaviour may be encapsulated within the stateful 'objects'.

1.4 Outline of the Thesis

This introductory chapter has indicated the intent and purpose of this thesis, and has surveyed existing database programming languages. The next chapter introduces the relational model while presenting jRelix in a tutorial-like fashion. Chapter 3 introduced the notion of a computation and also provides a tutorial on how they are used in jRelix. The next chapter documents the implementation of computations. Finally, chapter 5 summarizes the thesis and speculates about possibilities for future work.

Chapter 2

Overview of jRelix

The purpose of this chapter is to acquaint the reader with jRelix to the extent that the remainder of the thesis will be intelligible to him or her. jRelix is a relational database programming language. At its core is a database management system which is responsible for organizing and storing data. In this tutorial discussion, two types of syntax are discussed: langauge expressions and statements as well as housekeeping commands.

Section 2.1 lays the foundation for the rest of this chapter by explaining precisely what relations and domains are. High-level programming features are included for manipulating the stored data. The operators provided fall into two categories: the relational algebra and the domain algebra. These two topics are the subjects of sections 2.2 and 2.3. The update statement is covered in section 2.4. Finally, section 2.5 discusses nested relations.

2.1 Relations and Domains

The jRelix database programming language is based on the relational model. The relation is the only data type available to the user. Figure 2.1 on the next page[1] illustrates how a relation may be viewed as a table having rows and columns. The rows are referred to as tuples, and their ordering is not important. Each tuple of the *cigars* relation describes one type of cigar. The columns are headed by a label.

[1]Prices listed are for boxes of ten cigars, and are only approximate.

cigars	(BRAND	COUNTRY	SIZE	PRICE)
	Cohiba	Cuba	Robusto	250
	Cohiba	Cuba	Lonsdale	200
	Hoyo de Monterrey	Honduras	Double Corona	28
	Romeo y Julieta	Cuba	Petit Corona	120
	Fuente	Dominican Republic	Figurado	120
	Montecristo	Cuba	Petit Corona	125
	Montecristo	Dominican Republic	Lonsdale	65

Figure 2.1: A flat relation

Therefore, the ordering of the columns is also immaterial, as they may be referred to by their label. The technically correct term for label is *attribute*. All values found in some column are taken from a fixed pool, or *domain*, of possible values. In our example, the first three columns have entries consisting of character strings, and the last contains only integers. The primitive types supported by jRelix are summarized in table 2.1 on the following page. jRelix does not make a strong distinction between attributes and domains. Rather, the two are inextricably tied into one entity simply called domain. Thus we have

Definition 1 *A domain is a named set consisting of all elements of some primitive type.*

In the remainder of this thesis the term domain will bear this meaning.

For instance, if SHORTS is defined to be a domain of type **short**, then SHORTS is the name of a set of all short integers. More precisely, since short integers are 16 bit quantities in jRelix, SHORTS is the set of all integers from -32766 to -32767 together with the two special null values DC and DK. See [30] for additional information on null values in jRelix.

A relation is then defined as follows.

Definition 2 *A relation on a set, D, of domains is a subset of the cartesian product of the domains in D.*

The syntax for declaring a domain in jRelix is

Type	Shorthand	Size
boolean	bool	1 byte
short	short	2 bytes
integer	intg	4 bytes
long	long	8 bytes
float	real	4 bytes
double	double	8 bytes
string	strg	variable
text	text	variable

Table 2.1: Primitive types in jRelix

```
>domain BRAND string;
>domain COUNTRY string;
>domain SIZE string;
>domain PRICE intg;
```

Figure 2.2: Domain declarations

Syntax

```
DomainDecl := "domain" IDList Type ";"
IDList     := Identifier ("," Identifier)*
```

The `IDList` specifies the list of domains that is being declared. It is possible to declare several domains at once provided they are all of the same type. Type can be any of the primitive types supported by jRelix[2].

Some example domain declarations are given in figure 2.2. In the following figure are examples of the **sd** command which is used either to show all the domains currently defined in the system, or to show a specific domain. The syntax for this command is

Syntax

```
"sd" (Identifier)?  ";"
```

[2]Appendix A on page 116 contains the complete jRelix syntax.

```
>sd BRAND;
--------------------------- Domain Entry ---------------------------
Name         Type            NumRef       Dom_List
--------------------------------------------------------------------
BRAND        string              0
--------------------------------------------------------------------
>sd;
--------------------------- Domain Table ---------------------------
Name         Type            NumRef       Dom_List
--------------------------------------------------------------------
PRICE        integer             0
SIZE         string              0
BRAND        string              0
COUNTRY      string              0
--------------------------------------------------------------------
```

Figure 2.3: Command sd

The syntax for the declaration of a relation in jRelix is

Syntax

```
RelationDecl      := "relation" IDList "(" IDList ")" (Initialization)?  ";"
Initialization    := "<-" "{" ConstantTupleList "}"
                     | Identifier
ConstantTupleList := ConstantTuple ("," ConstantTuple)*
ConstantTuple     := "(" Constant ("," Constant)* ")"
Constant          := Literal
```

The first `IDList` in `RelationDecl` specifies the relations that are to be created. All of these will be defined on the same domains and will be initialized identically. The `IDList` within parentheses is the list of domains on which the new relation(s) will be defined. Initialization is optional.

An example of the creation of a relation is given in figure 2.4 on the next page. In the following figure are examples of the **sr** command which is used either to show information about all the relations currently defined in the system, or to show information about a specific relation if the optional parameter `Identifier` is provided. The **pr** command, which is used to print a relation on the screen, is also demonstrated. The syntax for these commands are

Syntax

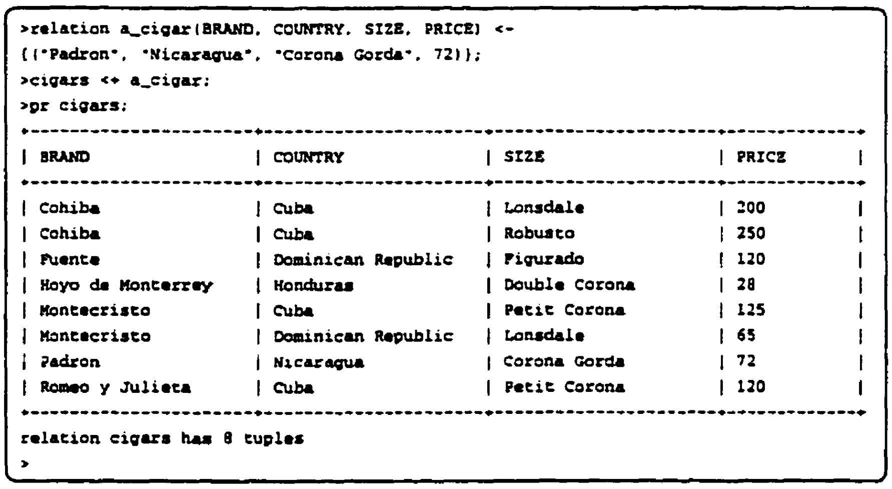
```
"sr" (Identifer)?  ";"
"pr" Expression ";"
```

```
>relation cigars(BRAND, COUNTRY, SIZE, PRICE) <- {
("Cohiba", "Cuba", "Robusto", 250),
("Cohiba", "Cuba", "Lonsdale", 200),
("Hoyo de Monterrey", "Honduras", "Double Corona", 28),
("Romeo y Julieta", "Cuba", "Petit Corona", 120),
("Fuente", "Dominican Republic", "Figurado", 120),
("Montecristo", "Cuba", "Petit Corona", 125),
("Montecristo", "Dominican Republic", "Lonsdale", 65)
};
>
```

Figure 2.4: Creating relations

```
>sr;
------------------------------ Relation Table ------------------------------
Name        Type          Arity       NTuples      Sort
----------------------------------------------------------------------------
cigars      relation          4           7           4
----------------------------------------------------------------------------
>pr cigars;
+----------------------+----------------------+----------------------+------------+
| BRAND                | COUNTRY              | SIZE                 | PRICE      |
+----------------------+----------------------+----------------------+------------+
| Cohiba               | Cuba                 | Lonsdale             | 200        |
| Cohiba               | Cuba                 | Robusto              | 250        |
| Fuente               | Dominican Republic   | Figurado             | 120        |
| Hoyo de Monterrey    | Honduras             | Double Corona        | 28         |
| Montecristo          | Cuba                 | Petit Corona         | 125        |
| Montecristo          | Dominican Republic   | Lonsdale             | 65         |
| Romeo y Julieta      | Cuba                 | Petit Corona         | 120        |
+----------------------+----------------------+----------------------+------------+
relation cigars has 7 tuples
>
```

Figure 2.5: Commands sr and pr



```
>relation a_cigar(BRAND, COUNTRY, SIZE, PRICE) <-
{("Padron", "Nicaragua", "Corona Gorda", 72)};
>cigars <+ a_cigar;
>pr cigars;
+----------------------+----------------------+---------------------------+-------------+
| BRAND                | COUNTRY              | SIZE                      | PRICE       |
+----------------------+----------------------+---------------------------+-------------+
| Cohiba               | Cuba                 | Lonsdale                  | 200         |
| Cohiba               | Cuba                 | Robusto                   | 250         |
| Fuente               | Dominican Republic   | Figurado                  | 120         |
| Hoyo de Monterrey    | Honduras             | Double Corona             | 28          |
| Montecristo          | Cuba                 | Petit Corona              | 125         |
| Montecristo          | Dominican Republic   | Lonsdale                  | 65          |
| Padron               | Nicaragua            | Corona Gorda              | 72          |
| Romeo y Julieta      | Cuba                 | Petit Corona              | 120         |
+----------------------+----------------------+---------------------------+-------------+
relation cigars has 8 tuples
>
```

Figure 2.6: Assignment and incremental assignment

jRelix provides two assignment operators, one for regular assignment and another for 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 then copied to it. If a relation with the same name as the destination already exists in the system, then it is first removed. The source relation is not affected by the assignment operator. The incremental assignment behaves identically to the assignment if there does not already exist a relation with the name specified on the left of the operator. If such a relation does exist then the destination relation becomes the union of the source and destination relations provided that both of these relations are defined on the same set of domains. If their domains do not coincide an error is reported. The syntax for these operators is shown below. Examples of how to use these are provided in figure 2.6.

Syntax

```
Assign    := Identifier "<-" Expression ";"
IncAssign := Identifier "<+" Expression ";"
```

2.2 Relational Algebra

The relational algebra consists of a set of functional operators which act on either one or two relations and produce a relation as a result. This closure property of the relational algebra permits many operators to be chained together. In this way, intricate queries of a database may be posed tersely.

2.2.1 Unary operators

jRelix supports four unary operators. These are projection, selection, T-selection and QT-selection. All of these operators require a relation as input and produce a relation as output. In all cases, the source relation is not affected by the operator.

Projection

Syntax

```
Projection := "[" (IDList)?  "]" "in" Expression ";"
```

The project operator returns a subset of the source relation consisting of the domains which are named in the projection list, `IDList` above. The source relation is the result of the `Expression`, which may be an arbitrary relational expression.

Example

Retrieve the brand and size of all cigars in the *cigars* relation.

```
>brands <- [BRAND,SIZE] in cigars;
>pr brands;
+--------------------+---------------------+
| BRAND              | SIZE                |
+--------------------+---------------------+
| Cohiba             | Lonsdale            |
| Cohiba             | Robusto             |
| Fuente             | Figurado            |
| Hoyo de Monterrey  | Double Corona       |
| Montecristo        | Lonsdale            |
| Montecristo        | Petit Corona        |
| Padron             | Corona Gorda        |
| Romeo y Julieta    | Petit Corona        |
+--------------------+---------------------+
relation brands has 8 tuples
>
```

Figure 2.7: Projection

Selection

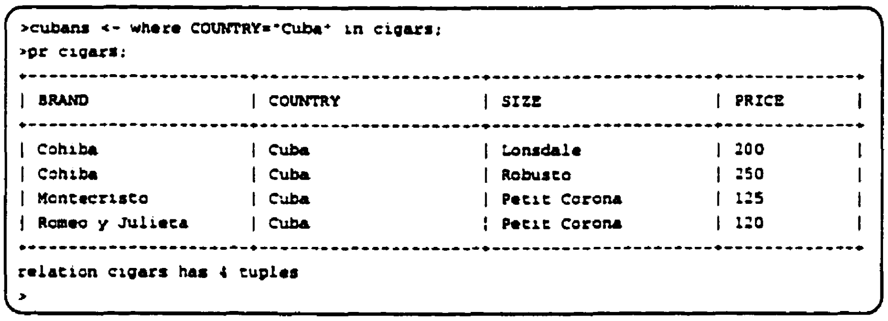
Syntax

```
Selection    := "where" SelectClause "in" Expression ";"
SelectClause := Expression
```

The select operator returns a subset of the source relation consisting of those tuples which satisfy the conditions of the selection clause. The source relation is the result of the `Expression` in `Selection`.

Example

Retrieve all Cuban cigars from the *cigars* relation.



```
>cubans <- where COUNTRY="Cuba" in cigars;
>pr cigars;
+-------------------------+-------------------------+-------------------------+---------------+
| BRAND                   | COUNTRY                 | SIZE                    | PRICE         |
+-------------------------+-------------------------+-------------------------+---------------+
| Cohiba                  | Cuba                    | Lonsdale                | 200           |
| Cohiba                  | Cuba                    | Robusto                 | 250           |
| Montecristo             | Cuba                    | Petit Corona            | 125           |
| Romeo y Julieta         | Cuba                    | Petit Corona            | 120           |
+-------------------------+-------------------------+-------------------------+---------------+
relation cigars has 4 tuples
>
```

Figure 2.8: Selection

T-Selection

Syntax

```
Projection   := "[" (IDList)?  "]" "where" SelectClause "in" Expression ";"
SelectClause := Expression
```

The T-selector combines the two preceding operators into one. It is considered as an operator in its own right, instead of simply as a selection followed by a projection, because it possible for both of these tasks to be accomplished in a single pass of the data. It is worthwhile for the user to be aware of this issue in order to write more efficient code.

Example

Retrieve the brand and size of all Cuban cigars found in the *cigars* relation.

```
>cubanBrands <- [BRAND, SIZE] where COUNTRY='Cuba' in cigars;
>pr cubanBrands;
+--------------------+--------------------+
| BRAND              | SIZE               |
+--------------------+--------------------+
| Cohiba             | Lonsdale           |
| Cohiba             | Robusto            |
| Montecristo        | Petit Corona       |
| Romeo y Julieta    | Petit Corona       |
+--------------------+--------------------+
relation brands has 4 tuples
>
```

Figure 2.9: T-Selection

Array syntax

There is a syntactic sugar for a special case of the T-select statement in which the select clause consists of a conjunction of comparisons for equality and the projection list consists of all domains not mentioned as part of the selection. This is called the array syntax because it resembles the arrays provided by other programming languages. The syntax is

Syntax

```
arraySelect := "[" (Identifier)?  (, (Identifer)?)* "]"
```

For example, the following two statements are equivalent.

```
cubans <- cigars[ , "Cuba", ,];
cubans <- [BRAND, SIZE, PRICE] where COUNTRY="Cuba" in cigars;
```

This works like a positional notation. The commas have created four slots which correspond to the four domains of *cigars*, from left to right. This works by selecting for equality on all values that are provided. In this example, only one value is specified—entries under the second domain, *COUNTRY*, must have the value "Cuba". In general, any number of values may be supplied. This particular example can be simplified even further to

```
cubans <- cigars[, "Cuba"];
```

That is, the trailing commas can be left off and will be inferred by the system.

QT-Selection

Support for the QT-selector has not yet been included in jRelix, although the parser has been programmed to recognize the syntax for it. More information on this operator can be found in [47].

2.2.2 Binary operators

jRelix supports 19 binary operators. All of these operators require two relations as input and produce a relation as output. In all cases, the source relations are not affected by the operator. The domains which are common to both source relations are called the join domains. The binary operators are classified into two groups: the μ-joins and the σ-joins. The syntax for these operators is

$\boxed{\textbf{Syntax}}$

```
JoinExpression := Expression JoinOperator Expression
                  | Expression "[" ExpressionList ":" JoinOperator
                  (":")? ExpressionList "]" Expression
```

μ-joins

The μ-joins, of which there are 7, correspond to the binary set operations of union, intersection and difference. Except for the difference joins, the result of their applica-

Join	Syntax	Description
natural join	ijoin or natjoin	center
union join	ujoin	left ∪ center ∪ right
left join	ljoin	left ∪ center
right join	rjoin	center ∪ right
left difference join	dljoin or djoin	left
right difference join	drjoin	right
symmetric difference join	sjoin	left ∪ right

Table 2.2: Summary of μ-joins

tion is a relation which has as its domains the union of the domains of the two input relations.

The μ-joins are summarized in table 2.2. `JoinOperator` may be any one of these. In the case where the two relations resulting from the expressions have no common domains, the user may specify which domains form the join domains. This is the purpose of the two `ExpressionList` elements in the second form of the `JoinExpression` shown above.

The μ-join operators can be defined in terms of what are called the left, center, and right components of the join. Given two relations R, defined on sets of domains X and Y, and S, defined on sets of domains Y and Z, these components are defined as follows:

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

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

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

As mentioned, the variables X,Y, and Z represent sets of domains in the above definitions. The only restriction is that the sets X, Y and Z have empty intersection.

ijoin

The ijoin is also known as the natural join, and it corresponds to the intersection operation on sets. The natural join of two relations, R and S, is equal to center(R,S).

```
+---------------------+-------------------------+-------------+
| BRAND               | SIZE                    | QTY         |
+---------------------+-------------------------+-------------+
| Cohiba              | Robusto                 | 5           |
| Fuente              | Figurado                | 15          |
| Hoyo de Monterrey   | Double Corona           | 30          |
| Partagas            | Double Corona           | 15          |
+---------------------+-------------------------+-------------+
relation orders has 4 tuples
```

Figure 2.10: Relation orders

The relation *orders* shown in figure 2.10 will be used to illustrate this operator.

Example

```
>i3 <- cigars ijoin orders;
>pr i3;
+--------------------+--------------------+--------------------+-------------+-------------+
| BRAND              | SIZE               | COUNTRY            | PRICE       | QTY         |
+--------------------+--------------------+--------------------+-------------+-------------+
| Cohiba             | Robusto            | Cuba               | 250         | 5           |
| Fuente             | Figurado           | Dominican Republic | 120         | 15          |
| Hoyo de Monterrey  | Double Corona      | Honduras           | 28          | 30          |
+--------------------+--------------------+--------------------+-------------+-------------+
relation i3 has 3 tuples
>
```

Figure 2.11: Natural join

The other μ-join operators are not pertinent to the remainder of this thesis. We refer the reader to [47] for a discussion of these operators.

σ-joins

The σ-joins, of which there are 12, correspond to the set comparison operators such as 'is a subset of?' and 'equals?'. The result of their application is a relation whose domains are the symmetric difference of the two sets of domains of the two input relations. The `JoinOperator` of the syntax section shown earlier in this subsection can also be any of the σ-joins which are summarized in table 2.3 on the next page[3].

[3]The table only lists 7 σ-joins, when in fact there are 12. The five that are missing correspond to the logical negation of entries 2 though 6 in the table. The first entry, called natural composition, is in fact the logical negation of the last entry. It is included in the table because it is exceptionally useful.

Join	Syntax	Mathematical operator
natural composition	icomp or natcomp	$\not\cap$
equal join	eqjoin	$=$
greater than or equal join	gejoin or sup or div	$\supseteq$
greater than join	gtjoin	$\supset$
less than or equal join	lejoin or sub	$\subseteq$
less than join	ltjoin	$\subset$
empty intersection join	iejoin or sep	$\cap\!\!\circ$

Table 2.3: Summary of σ-joins

As of this writing, σ-joins are not implemented in jRelix, although they are supported by Relix and will soon be included in jRelix. For this reason, their use will not be demonstrated. See [47] for more details on σ-joins in Relix.

2.3 Domain Algebra

The operators of the domain algebra can be classified into two groups, horizontal and vertical. The following two sections discuss these, respectively.

2.3.1 Horizontal operations

The example of figure 2.12 will be used to explain the horizontal operations of the domain algebra. The first statement is a declaration of the virtual domain *SUBTOTAL*. In the following statement, this new domain is included in a projection list just as regular domains are. The relation resulting from the join of *cigars* and *orders* (which was shown in figure 2.11 on the preceding page) does not contain a column labeled *SUBTOTAL*. Therefore, the system automatically creates this column and fills it in with values corresponding to the expression provided in the 'let' statement for *SUBTOTAL*. This extra column is shown in figure 2.13 on the next page outside of the parentheses, which contain only the regular domains. In each tuple, the value for *SUBTOTAL* is equal to the product of the values for *PRICE* and *QTY*, as prescribed

```
>let SUBTOTAL be PRICE * QTY;
>bill <- [BRAND, SUBTOTAL] in (cigars ijoin orders);
>pr bill;
+-----------------------+-------------+
| BRAND                 | SUBTOTAL    |
+-----------------------+-------------+
| Cohiba                | 1250        |
| Fuente                | 1800        |
| Hoyo de Monterrey     | 840         |
+-----------------------+-------------+
relation bill has 3 tuples
```

Figure 2.12: Actualization of a virtual domain, part 1a

(BRAND	SIZE	COUNTRY	PRICE	QTY)	SUBTOTAL
Cohiba	Robusto	Cuba	250	5	1250
Fuente	Figurado	Dominican Republic	120	15	1800
Hoyo de Monterrey	DoubleCorona	Honduras	28	30	840

Figure 2.13: Actualization of a virtual domain, part 1b

by the virtual domain declaration. Thus, when the print statement of figure 2.12 is executed, the relation *bill* is indeed as shown in that figure.

In summary, to make use of the domain algebra, it is necessary to declare a virtual domain beforehand. These virtual domains may then be actualized in any number of relations. In order to accomplish this, the user simply includes the virtual domains in the projection list of a relational algebra statement. The resulting relation will, as usual, be a relation defined on the domains which are specified in the projection list. In the case of regular domains, the entries are copied over from the source relation. However, in the case of virtual domains, the source relation is not defined on the domain and, therefore, the data under this column in the destination relation must be calculated and then filled in. Once this has been done, actual data exists under the domain in the destination relation. We thus say that the virtual domain has been actualized.

Syntax

```
HorizontalDecl := "let" Identifier "be" Expression ";"
```

The horizontal operators may be grouped according to their 'arity'. Constants are really just operators of zero-arity. There is only one ternary operator, the if-then-else construct. Predefined functions can be thought of as operators of arbitrary arity. In fact, it is a very fine line which separates operators from functions[4].

Constants

Example

```
>let ONE be 1;
>let TRUE be true;
```

Figure 2.14: Constant virtual domains

Unary Operators

Example

```
>let MINUSONE be -ONE;
>let FALSE be not TRUE;
```

Figure 2.15: Unary virtual domains

Binary Operators

Example

```
>let SUBTOTAL be PRICE * QTY;
```

Figure 2.16: Binary virtual domains

[4]The difference is a syntactical one. The arguments of a function are enclosed in parentheses, where as those of an operator are not.

Conditional Expression

Example

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

Figure 2.17: Conditional virtual domains

Predefined Functions

Example

```
>let ROOTX be sqrt(X);
```

Figure 2.18: Predefined functions in virtual domains

2.3.2 Vertical operations

With the horizontal operators of the preceding section, the actualization of a virtual attribute in each tuple of a relation only involved values in the current tuple. This is not true of the vertical operators of the domain algebra. The actualization of these operators depends on values found across several tuples of the source relation.

Syntax

```
VerticalDecl := "let" Identifier "be" VerticalExpr ";"
VerticalExpr := "red" AssoCommuOperator "of" Expression
                | "equiv" AssoCommuOperator "of" Expression
                "by" ExpressionList
                | "fun" OrderedOperator "of" Expression
                "order" ExpressionList
                | "par" OrderedOperator "of" Expression
                ( "order" ExpressionList "by" ExpressionList
                  | "by" ExpressionList "order" ExpressionList )
```

The vertical operations fall into four categories: reduction, equivalence reduction, functional mapping and partial functional mapping. Only the first two of these will be discussed here. More information on this may be found in [47].

Reduction

The reduction operator works by looping over the tuples of the input relation while keeping an accumulator. At each tuple, the value of **Expression** is combined with the accumulator by means of the operator specified by **AssoCommuOperator**. The end result is then inserted as the value of the actualized domain in every tuple of the input relation. Because the notion of relation abstracts over the order of its tuples, the **AssoCommuOperator** must be commutative for the result to be well-defined. The need for associativity is also apparent. Indeed, regardless of the abstraction over the ordering of tuples, without associativity the result of a reduction would not be well-defined for relations having more than two tuples.

Example

```
>let COUNT be red + of 1;
>let AVGPRICE be (red + of PRICE)/COUNT;
>let TOTAL be red + of SUBTOTAL;
```

Figure 2.19: Reduction

Figure 2.20 on the following page illustrates the actualization of a reduction operator. It builds on that of figure 2.12 on page 35 to calculate the total cost of the order for cigars. The example also illustrates how virtual domains may themselves depend on virtual domains. The first statement is a declaration of *TOTAL*. In the following statement, this new domain is included in a projection list just as regular domains are. The relation resulting from the join of *cigars* and *orders* (which was shown in figure 2.11 on page 33) does not contain a column labeled *TOTAL*. Therefore, the system automatically creates this column and fills it in with values corresponding to the expression provided in the 'let' statement for *TOTAL*. This extra column is shown

```
>let TOTAL be red + of SUBTOTAL;
>bill2 <- [TOTAL] in (cigars ijoin orders);
>pr bill2;
+-------------+
| TOTAL       |
+-------------+
| 3890        |
+-------------+
relation bill2 has 1 tuple
```

Figure 2.20: Actualization of a virtual domain, part 2a

(BRAND	SIZE	COUNTRY	PRICE	QTY)	SUBTOTAL	TOTAL
Cohiba	Robusto	Cuba	250	5	1250	3890
Fuente	Figurado	Dominican Republic	120	15	1800	3890
Hoyo de Monterrey	DoubleCorona	Honduras	28	30	840	3890

Figure 2.21: Actualization of a virtual domain, part 2b

in figure 2.21. Since the expression to calculate $TOTAL$ depends on $SUBTOTAL$, this virtual domain is shown as well. The system will resolve such dependencies to any depth. In each tuple, the value for $TOTAL$ is equal to the sum of all the values for $SUBTOTAL$, as prescribed by the virtual domain declaration. Thus, when the print statement of figure 2.20 is executed, the relation *bill2* is indeed as shown in that figure.

Equivalence Reduction

Equivalence reduction works in much the same way as reduction except that the tuples of the source relation are grouped according to a 'by' clause, which is an **ExpressionList**. All tuples that agree on all the expressions in Expressionlist are considered 'equivalent'. The reduction is carried out as usual within these groups.

Example

```
>let COUNTBYBRAND be equiv + of 1 by BRAND;
>let AVGBYBRAND be (equiv + of PRICE by BRAND)/COUNTBYBRAND;
```

Figure 2.22: Equivalence reduction

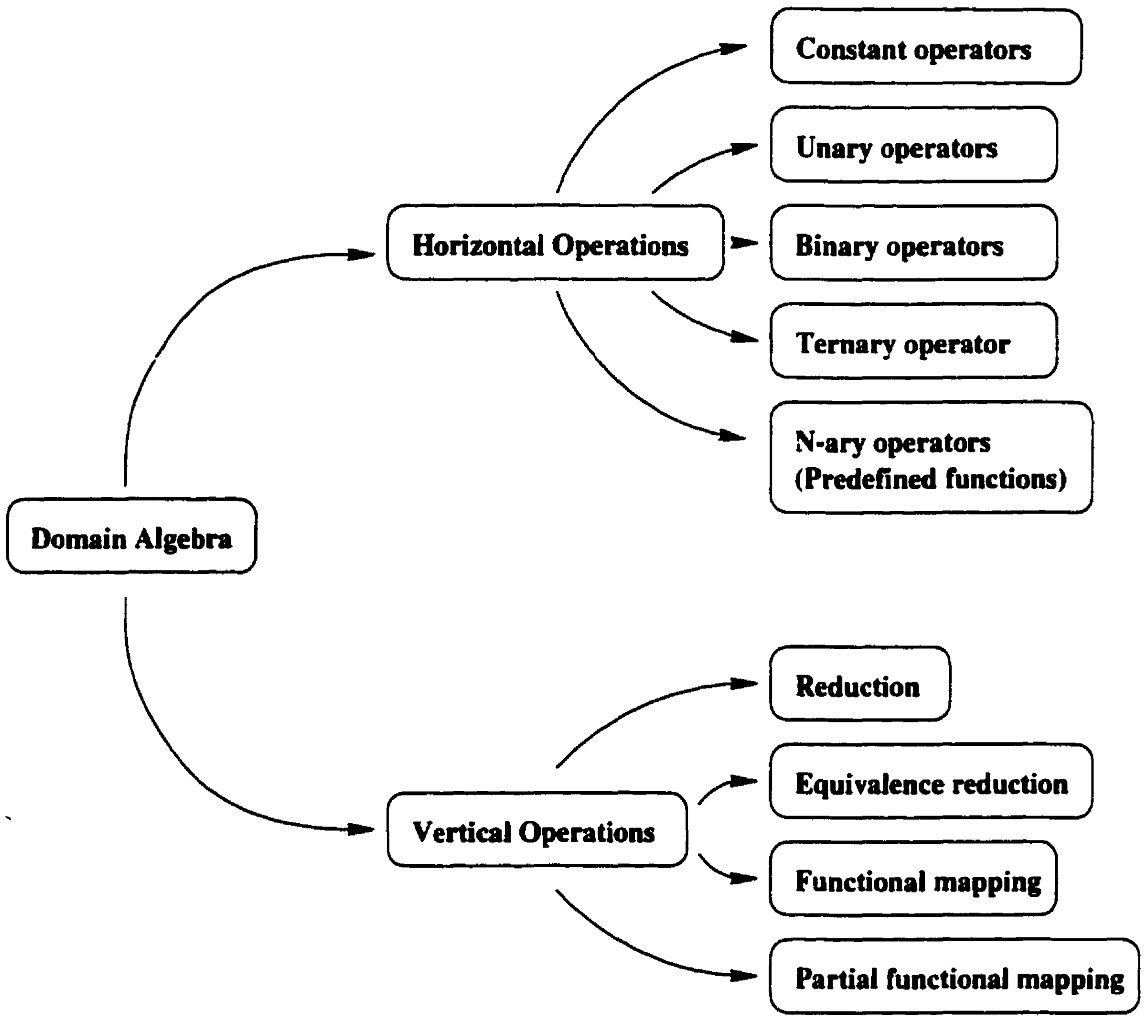


Figure 2.23: The domain algebra

Figure 2.23 provides a summary of the domain algebra.

2.4 Update Statement

The update statement may be used to add, delete or modify entries in a relation. The syntax for the update statement is

Syntax

```
"update" Identifier ("add" | "delete") Expression ";" |
"update" Identifier "change" (StmtList)?  (UsingClause)?  ";"
UsingClause := "using" (JoinOperator)?  Expression
```

`Identifier` is the name of the relation that is to be updated.

Figure 2.24 on the next page demonstrates the use of the update statement. The relation *orders* from figure 2.10 on page 33 will be used. The first form of the update statement allows tuples to be added or deleted from a relation. Adding tuples in this way is equivalent to the incremental assignment operator. The result is the union of the relation to be updated with the relation that results from `Expression`. The first form of the update statement may also be used to delete a tuple. This works like the left difference join. The following are equivalent.

```
update Identifier delete Expression;
Identifier <- Identifier djoin Expression;
```

The second form of the update statement may be used to change entries in some tuples of a relation. The purpose of `UsingClause` is to specify the tuples of the relation that will be updated. In the example, only the tuples that have *BRAND* set to "Cohiba" are updated. If `JoinOperator` is missing, then the natural join is assumed. `StmtList` is a list of statements specifying the changes that will be made to the affected tuples.

2.5 Nested Relations

jRelix includes full support for nested relations. An important design goal of the system, however, is to present to the user a simple and elegant programming paradigm. Therfore, it is not desirable to have separate definitions of both a flat relation and of a nested relation. Indeed, this would lead to a convoluted syntax for the programming language in which flat relations and nested relations would be treated differently. The user would always have to keep this dichotomy in mind. It is preferable to come up with a definition which conceptually unites these two ostensibly disparate types of relations.

Figure 2.25 on the next page illustrates how a nested relation may be thought of by the user. The domain *DEPT* is a regular domain of type **string**. *EMP* is a compound domain. In the column underneath it are stored relations. There is, however, a restriction on the type of relations that may be placed there—they must

```
>relation newItem(BRAND, SIZE, QTY) <-
{ ("Montecristo", "Petit Corona", 25) };
>update orders add newItem;
>update orders delete where BRAND = "Fuente" in orders;
>update orders change QTY <- 12 using ijoin
    where BRAND = "Cohiba" in orders;
>pr orders;
+----------------------+----------------------+--------------+
| BRAND                | SIZE                 | QTY          |
+----------------------+----------------------+--------------+
| Cohiba               | Robusto              | 12           |
| Hoyo de Monterrey    | Double Corona        | 30           |
| Montecristo          | Petit Corona         | 25           |
| Partagas             | Double Corona        | 15           |
+----------------------+----------------------+--------------+
relation orders has 4 tuples
>
```

Figure 2.24: Update statement

employees	(DEPT	EMP	)
		(NAME	SAL)
	stereo	M. Gordon	35k
		P. McConnel	32k
		T. Anastasio	27k
		J. Fishman	44k
	television	J. Medeski	35k
		B. Martin	38k
		C. Wood	32k

Figure 2.25: A nested relation

all have the same schema. In this case, the relations are defined on the two domains *NAME* and *SAL*.

Before giving the definition of a relation, we review what is meant by the term domain.

Definition 3 *A primitive domain, also called a simple domain, is a named set which consists of all elements of some primitive type.*

This is the definition that was given earlier in the chapter. Domains may also be created from existing domains. A domain which is defined on a set of existing domains is called non-primitive or compound. The domains on which it is defined form the type of the new domain.

Definition 4 *A non-primitive domain, also called a compound domain, is defined to be a named power set*[5] *of the cartesian product of the domains that make up its type.*

Only a small extension to the data definition language is required to support nested relations. First, the syntax must allow for the declaration of both simple and compound domains.

Syntax

```
DomainDecl := "domain" IDList Type ";"
IDList     := Identifier("," Identifier)*
Type       := PrimType | "(" IDList ")"
```

This differs from the syntax for domain declarations given previously only in that the **Type** may be a parenthesized list of identifiers, as well as one of the primitive types listed in table 2.1 on page 24. The list of identifiers is used when declaring a compound domain, and it specifies the schema that a relation must have in order for it to be a valid data entry for the newly declared domain. An example declaration is provided by figure 2.26 on the next page.

Now that the definition of a domain has been stated, it is possible to define a relation.

[5]The power set of a set A is defined as the set of all subsets of A.

```
>domain DEPT, NAME string;
>domain SAL integer;
>domain EMP(NAME, SAL);
```

Figure 2.26: Nested domain declaration

Definition 5 *A relation on a set, D, of domains is a subset of the cartesian product of the domains in D.*

This turns out to be precisely the same definition that was given earlier for flat relations. All the hard work has been put into the definition of a domain. Consequently, it is possible to give a simple definition of a relation that unites nested relations with flat relations. Doing it this way is quite advantageous. It is because we have a single definition of relation, that the syntax of jRelix requires so few alterations from the original Relix syntax. This also permits jRelix to continue to have only one data type as Relix did, namely, the relation. All the operations in jRelix apply to and return relations, that is the only data type which is made available to the user. Having only one data type permits relational operators to be cascaded thereby achieving much expressivity with small amounts of code. This is an example of the principle that 'everything is a relation' at work. We will see this principle pushed even further when computations are discussed.

The second extension to the data definition language allows the declaration and initialization of nested relations.

Syntax

```
RelationDecl        := "relation" IDList "(" IDList ")" (Initialization)?  ";"
Initialization      := "<-" "{" ConstantTupleList "}"
                       | Identifier
ConstantTupleList := ConstantTuple ("," ConstantTuple)*
ConstantTuple       := "(" Constant ("," Constant)* ")"
Constant            := Literal | "{" ConstantTupleList "}"
```

This is quite similar to its counterpart for flat relations that was given earlier in this chapter. The only difference is that a `Constant` can now be a parenthesized `ConstantTupleList` as well as a `Literal`. The 'curly bracket' syntax was extended

```
>relation employees(DEPT, EMP) <- {
("stereo", {
  ("M. Gordon", 35000),
  ("P. McConnel", 32000),
  ("T. Anastasio", 27000),
  ("J. Fishman", 44000)
  }),
("television", {
  ("J. Medeski", 35000),
  ("B. Martin", 38000),
  ("C. Wood", 32000)
  })
};
```

Figure 2.27: Creating a nested relation

```
>let SeniorSalesman be [NAME] where SAL >= 38000 in EMP
>seniors <- [SeniorSalesman] in employees;
```

Figure 2.28: Nested domain algebra

in jRelix to support the initialization of nested relations. The definition is recursive in order to support nested relations of arbitrary depth. An example is given in figure 2.27.

The inclusion of nested relations in jRelix has led to an important enhancement of the domain algebra. Put briefly, the domain algebra has now been extended so as to include the relational algebra. This is perfectly natural since the items which are stored in a column headed by a compound domain are relations. It thus makes sense to apply relational operators to these entities.

An example of the use of relational expressions within domain algebra statements is provided in figure 2.28. The query determines those employees who have a salary of at least 38000 dollars.

Chapter 3

User Manual on Computations

Support for procedural abstraction is available in jRelix in the form of computations. In order to make use of computations, the user must first define one. Section 3.1 describes how this may be accomplished. Once the computation has been declared, it may then be invoked. Several methods are possible. These are covered in section 3.2. A mechanism has been included in computations which permits the creation of objects with state, having their proper accessor and mutator methods. This topic is explored in section 3.3. Recursive computations are discussed in section 3.4. The following section explains how computations may be used to verify constraints. Finally, section 3.6 introduces the various utility commands pertinent to computations.

3.1 Creating Computations

An example of the declaration of a computation is shown in figure 3.1 on the following page. The name of the computation is *velocity*. Its parameters, D, V and T, must first be declared as domains, otherwise an error is generated. There are three 'alt'—alternate—blocks in this example. All three of these satisfy the constraint $V = D/T$. Given values for any two of these variables, the value of third is implied by the constraint. Each 'alt' block contains the algorithm necessary for computing the value of one of these variables in terms of the other two. Thus, in our example, the first block calculates D when V and T are provided, and similarly for the other blocks.

A central design principle used in developing the notion of computation is to make

```
>domain D,V,T float;
>comp velocity(D,V,T) is
{ D <- V*T;
} alt
{ V <- D/T;
} alt
{ T <- D/V;
};
>
```

Figure 3.1: Velocity computation

VELOCITY	(D	V	T)
	1	1	1
	2	1	2
	2	2	1
	⋮	⋮	⋮
	34.1	5.5	6.2
	⋮	⋮	⋮

Figure 3.2: Velocity computation as a relation

them resemble relations. In the previous example, *velocity* embodied a constraint. It is the constraint that provides the link between viewing a computation as a procedural abstraction mechanism is the usual sense (i.e. a modular chunk of algorithmic code) and viewing it as a relation. In figure 3.2, the relation corresponding to *velocity* is shown. It is an infinite relation, every tuple of which satisfies the constraint $V = D/T$. Furthermore, every triplet of values for D, V and T which satisfies the constraint is included in this relation. We may think of the keyword **comp** as standing for COMPressed relation. The parameters of a computation become the domains of its associated relation. We therefore use the terms *parameters of a computations* and *domains of a computation* interchangeably.

An example of a computation which includes local variables is shown in figure 3.3 on the following page. The keyword **local** is used to declare two integer variables, *a*

```
>comp sum_squares(X,Y,Z) is
local a,b intg;
{ a <- X*X;
  b <- Y*Y;
  Z <- a+b;
} alt
{ a <- X*X;
  b <- Z-a;
  Y <- sqrt(b);
} alt
{ a <- Y*Y;
  b <- Z-a;
  X <- sqrt(b);
};
>
```

Figure 3.3: Sum of squares computation

and *b*, as local to the computation. They are only accessible within the 'alt' blocks of the computation. Local variables must have as their type one of the primitive types provided by jRelix. These were listed in table 2.1 on page 24.

The formal syntax for the declaration of a computation is

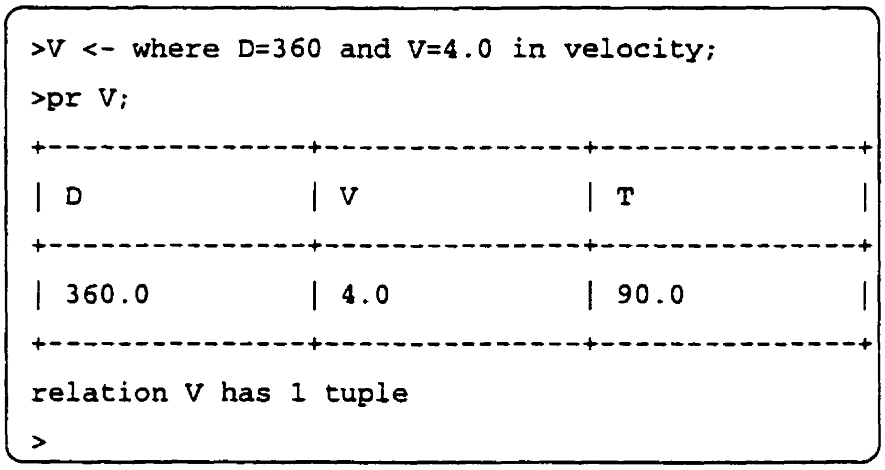
Syntax

```
CompDecl    := "comp" Identifier "(" ParamList ")" "is" CompBody ";"
ParamList   := (Identifier ("," Identifier)*)?
CompBody    := (CompVarDecl)* CompBlock ("alt" CompBlock)*
CompVarDecl := ("local" | "state") IDList Type ";"
CompBlock   := "{" (Statement ";")+ "}"
```

The keyword **state** which appears above will be described in section 3.3.

3.2 Invoking Computations

This section discusses the invocation of computations. This is accomplished without additional syntax. Certain operators of the relational algebra have been overloaded so



```
>V <- where D=360 and V=4.0 in velocity;
>pr V;
+---------------+---------------+---------------+
| D             | V             | T             |
+---------------+---------------+---------------+
| 360.0         | 4.0           | 90.0          |
+---------------+---------------+---------------+
relation V has 1 tuple
>
```

Figure 3.4: Applying the velocity computation with a selection

that they may take computations as well as relations as their arguments. The main issue at hand is how data is passed to a computation.

3.2.1 Invoking a computation with a select expression

Data may be supplied for the input parameters of a computation by means of a restricted form of the select expression. Figure 3.4 shows an example of this with the velocity computation. Specific values have been supplied for the parameters *D* and *V*. This causes these two domains to become the input parameters, so the third block is excuted in order to calculate *T*. The selection clause is restricted in this example as it must always be when selecting from a computation. Only the '**and**' and '=' operators may be used.

3.2.2 Array syntax for computation invocation

As described in section 2.2.1 on page 30, jRelix provides an array syntax which is essentially syntactic sugar for a T-select expression. This may be used as a positional notation for computation invocation. The expression

```
velocity[,4,90]
```

is equivalent to a T-select statement whose projection list consists of only the first parameter of *velocity*, and in which the second parameter has the input value 4, and

```
>V <- velocity[,4,90];
>pr V;
+---------------+
| D             |
+---------------+
| 360.0         |
+---------------+
relation V has 1 tuple
>
```

Figure 3.5: Array syntax for computation invocation

the third has the input value 90. Since the first parameter of the *velocity* computation is D, the second is V and the third is T, the above expression is equivalent to

```
[D] where V=4 and T=90 in velocity.
```

Figure 3.5 demonstrates this.

3.2.3 Join of a computation with a relation

It has been explained how a computation may be thought of as a relation, perhaps an infinite one. It therefore makes sense to take the natural join of a computation with a relation. The effect of this join of a relation, such as *DistnTime* shown in figure 3.6 on the next page, with the *velocity* computation is to pick out a (finite) subset of the tuples from the infinite relation corresponding to *velocity*. The result is also shown in the figure.

When two relations are joined together, their common domains are called the join domains. This terminology is retained when a computation is joined with a relation. It has already been observed that the parameters of a computation can be thought of as domains of the computation. The join domains become the input parameters of the computation. The remaining parameters become outputs. Thus, in our example, it is the second 'alt' block of *velocity* that is selected for execution by the system.

The properties of the natural join stay intact under this generalization of the operator. For instance, the join still behaves like a μ-join in that the resulting relation is

```
>pr DistnTime;
+---------------+---------------+
| D             | T             |
+---------------+---------------+
| 50.0          | 17.0          |
| 60.0          | 3.0           |
| 360.0         | 90.0          |
+---------------+---------------+
relation DistnTime has 3 tuples
>V <- velocity ijoin DistnTime;
>pr V;
+---------------+---------------+---------------+
| D             | V             | T             |
+---------------+---------------+---------------+
| 50.0          | 2.9411764     | 17.0          |
| 60.0          | 20.0          | 3.0           |
| 360.0         | 4.0           | 90.0          |
+---------------+---------------+---------------+
relation V has 3 tuples
>
```

Figure 3.6: Applying the velocity computation with an ijoin

```
>computation JoinComp(A,B,C) is
{ C <- A ijoin B;
};
>JoinComp(in cigars, in orders, out result);
>sr result;
------------------------------ Relation Entry ------------
Name         Type           Arity        NTuples      Sort
----------------------------------------------------------
result       relation           5            3           0
----------------------------------------------------------
>
```

Figure 3.7: Stand-alone computation call

defined on a set of domains consisting of the union of the domains of its two arguments. The relation that is joined with the computation is unaffected by the operation. So the operator remains functional. Finally, the natural join operator is still commutative and associative.

3.2.4 Stand-alone invocation of computations

Computations may also be invoked by means of a top-level call. The previous invocations involved generalizations of relational algebra operators and were a part of relational algebra expressions. The stand-alone invocation is a top-level statement. The keywords **in** and **out** are used to specify the input and output parameters of the computation. Figure 3.7 illustrates this for a simple computation which calculates the natural join of its first two parameters. The call specifies that the first parameter is an input and has value *cigars*. This must be a valid relation within the scope that the call is made in, otherwise an error is reported. Similarly, the second parameter is an input and the value passed in is the relation *orders*. The third parameter is an output. The relation returned by this output will be called *result* and will be placed in the scope that the call is made in. The **sr** command demonstrates that the *result* is indeed in that scope.

```
>comp sq_root(X,R) is
{ R <- sqrt(X)
   also
  R <- -sqrt(X);
};
>
```

Figure 3.8: Also operator

3.2.5 Multivalued computations

An output parameter of a computation may be assigned multiple values for each set of inputs. This is accomplished with the **also** syntax. Figure 3.8 illustrates this with a computation which calculates square roots. For every value of the input parameter, *X*, there are two values for the output parameter, *R*; the positive root, and the negative root. The 'also' statement may only be used between assignment statements, and the l-value (the identifier to the left of the '<-' symbol) of these must be the same. Due to the implementation, a further restriction is enforced. The 'also' statement may occur at most once in any 'alt' block, and, if used, must occur in the last statement of the block.

3.2.6 Properties of computations

We now summarize the main properties of computations.

- Computations are the procedural abstraction mechanism provided by jRelix. In order to keep the language simple and unified, they have been designed to resemble relations as closely as possible. Once declared, they are used in the same way as relations.

- Computations are symmetric with respect to their parameters. A parameter of a computation can serve as both input or output, depending on how it is called. There must, however, be an 'alt' block which corresponds to the set of input parameters provided.

- Computations are intended to embody a constraint. The code in the various 'alt' blocks should reflect this. However, ensuring consistency among the 'alt' blocks is left to the user and is not enforced by the system.

- Computations do not use a positional form of parameter passing as is common with procedural abstraction facilities in other programming languages. Instead, the name of the parameter must match the name of the data item being passed to the computation. This is a consequence of the design choice of modeling computations after relations. The relational algebra operators use name equivalence in determining join domains and in selection clauses. This therefore continues to be true when these operators are generalized to permit their arguments to be computations. To alleviate this restriction, the array syntax may be used as a positional syntax. The extended notation for join operators that permits the user to specify the join attributes explicitly may also be used. See appendix A on page 116 for the complete jRelix syntax.

3.3 Stateful Computations

Computations have been designed to include a facility which allows the user to create objects with state. We want to avoid the object-oriented solution of a separate entity, the *class* which contains state, and see if we can do it all with computations, a single construct. It is also possible to define accessor and mutator methods for these objects.

Figure 3.9 on the following page illustrates the concepts with a bank account example (*ba* stands for bank account). The objects are bank accounts. Each of these contains a stateful variable reflecting the current balance in that account. This variable is called *bal* and is declared with the keyword **state**, to indicate that it is a local variable whose value will be remembered between calls. There are two computations nested inside of *ba*. These are methods of the class *ba*. Note that the two nested computations have the same name as the formal parameters. This implies that these parameters are outputs for the only 'alt' block of *ba*. The nested computations will be exported when *ba* is called. That is, the relation that will result from the invocation of *ba* will contain the domains *DEPOSIT* and *BALANCE*, both of type computation.

```
>domain DEP,B intg;
>domain DEPOSIT comp(DEP);
>domain BALANCE comp(B);
>comp ba(BALANCE, DEPOSIT) is
state bal intg;
state oldbal intg;
{  comp DEPOSIT(DEP) is
   {  oldbal <- bal;
      bal <- bal + DEP;
   } alt
   {  DEP <- bal - oldbal;
   };
   comp BALANCE(B) is
   {  B <- bal;
   };
   bal <- 0;
};
>
```

Figure 3.9: Declaration of a bank account class

More specifically, the type of *DEPOSIT* is computation on *DEP*, and the type of *BALANCE* is computation on *B*.

The example also illustrates the declaration of domains of computation type. In the columns of the result relation headed by these domains, the computations defined within *ba* will be stored. This relation is thus a nested relation, although, instead of having relations nested within a relation, there are computations stored within a relation.

The declaration of *ba* does not create any objects, it is a class declaration. In order to instantiate objects of this class, an ijoin or a select expression must be used to call the *ba* computation. This is demonstrated in figure 3.10 on the next page. Two bank accounts are created in this example. Each will have their own copy of the variable *bal*. The last statement in the figure demonstrates how the client Suz can make a deposit of $100. Since this operation will change the relation accounts, the non-functional update

```
>domain ACCNO intg;
>domain CLIENT string;
>relation accts(ACCNO, CLIENT) <-
{  (1729, "pat"),
   (4104, "suz")
};
>accounts <- accts ijoin ba;
>update accounts change DEPOSIT(100)
  using where ACCNO = 4104 in accounts;
>
```

Figure 3.10: Instantiation of bank account objects

```
>comp transfer(FROMACC,TOACC,AMT) is
{ update accounts change DEPOSIT(-AMT)
   using where ACCNO=FROMACC in accounts;
  update accounts change DEPOSIT(AMT)
   using where ACCNO=TOACC in accounts;
};
>transfer(in 1729, in 4104, in 50);
```

Figure 3.11: Transfer of money between bank accounts

statement must be used. The computation stored in the column labeled *DEPOSIT* is the *DEPOSIT* method of the class *ba*. It is called using the stand-alone invocation syntax.

The example shown in figure 3.11 demonstrates how a binary operation may be implemented. The *transfer* computation removes *AMT* dollars from account *FROMACC* and adds it to account *TOACC*. Each of these actions is accomplished by calling the *DEPOSIT* method of the class *ba*.

3.4 Recursive Computations

Computations may be recursive. The example of figure 3.12 on the next page demonstrates this with a computation which computes factorials.

```
>domain M,N intg;
>comp factorial(M, N) is
{  if (M = 0) then
   { N <- 1;
   }
   else
   { N <- M * factorial[M-1];
   };
};
>R <- factorial[6];
>pr R;
+-------------+-------------+
| M           | N           |
+-------------+-------------+
| 6           | 720         |
+-------------+-------------+
relation R has 1 tuple
>
```

Figure 3.12: Recursive computation

```
>pr R;
+--------------+--------------+--------------+
| D            | V            | T            |
+--------------+--------------+--------------+
| 6.0          | 3.0          | 2.0          |
| 6.0          | 3.0          | 3.0          |
| 360.0        | 4.0          | 90.0         |
+--------------+--------------+--------------+
relation R has 3 tuples
>CV <- velocity ijoin R;
>pr CV;
+--------------+--------------+--------------+
| D            | V            | T            |
+--------------+--------------+--------------+
| 6.0          | 3.0          | 2.0          |
| 360.0        | 4.0          | 90.0         |
+--------------+--------------+--------------+
relation CV has 2 tuples
```

Figure 3.13: Constraint verification with a computation

3.5 Constraint Verification

Computations may be used to verify if the data entries of a relation satisfy a constraint. This occurs when a call is made in which all the parameters are inputs. The example in figure 3.13 demonstrates this. The relation R is defined on the same domains as the *velocity* computation. The relation CV is the relation that results from the join of R with the infinite relation that corresponds to *velocity*. Since the tuples in this latter relation consist, exclusively, of those which satisfy the constraint $V = D/T$, only those tuples of R which satisfy this constraint are found in CV.

There is a feature in jRelix which is useful in this context. An empty projection clause in a relational expression returns a relation defined on the system domain *.bool* having a single tuple. This relation will contain the boolean value *true* if the relation to which the project operator applies is not empty, otherwise, it will contain *false*.

Thus for the following two equivalent statements,

```
CV2 <- velocity[6,3,2];
CV2 <- [] where D=6 and V=3 and T=2;
```

the relation *CV2* will contain the value *true*.

3.6 Commands

The commands for displaying information about and deleting relations have been overloaded to apply to computations. In addition, there is the **sc** command which applies exclusively to computations. Figure 3.14 on the next page illustrates these commands.

Print Relation

The **pr** command prints the contents of a relation. When applied to a computation, it displays the source code of the computation.

Show Relation

The **sr** command is used to display information about a relation, view or computation. The *Type* field indicates which of these three possibilities the argument to the command happens to be. If no argument is provided, information is provided for all existing relations, views and computations.

Delete Relation

The **dr** command is used to delete a relation, view or computation.

Show Computation

The **sc** command applies to computations exclusively. It displays information about the 'alt' blocks of a computation. The format used is

[*Input Parameter List*] – > [*Output Parameter List*]

```
>sr velocity;
------------------------------ Relation Entry -----------
Name         Type             Arity        NTuples      Sort
-----------------------------------------------------------
velocity     computation          3            0            0
-----------------------------------------------------------
>sc velocity;
velocity (D, V, T)
  [ V T ] -> [ D ]
  [ D T ] -> [ V ]
  [ D V ] -> [ T ]
>pr velocity;
comp velocity(D,V,T) is
{ D <- V*T;
} alt
{ V <- D/T;
} alt
{ T <- D/V;
};
>
```

Figure 3.14: Commands for computations

Chapter 4

Implementation of Computations

In this chapter, the implementation of computations is described in considerable detail. In order to accomplish this, some aspects of the jRelix system must first be covered, such as the format of the data dictionary both on disk and in RAM, and the way nested relations are implemented. These and other related topics are addressed in section 4.1. jRelix relies on the Javacc compiler compiler and on JJTree. a tool which generates syntax trees while parsing statements. Section 4.2 explains how this is done. An important issue which arises when including a mechanism for procedural abstraction in a programming language is how the environment model for storing variables is coupled with procedures. There are, in fact, several possibilities. Environments form the subject of section 4.3. The next two sections describe how computations are created and how they are invoked. Details of the implementation of stateful computations are covered in section 4.6. The following section describes the manner in which computations are stored, both in RAM and on disk. Finally, the implementation of recursive computations is discussed in section 4.8.

4.1 Overview of the jRelix Implementation

One of the most important design principles of the jRelix system is that there be only one data type—the relation. The syntax of the language reflects this philosophy. Indeed, there is no way to declare primitive types such as integers or strings. Other than for regular and virtual domain declarations, which can not store data, only

Relation	Domain	Description
.rel	.rel_name	name of a relation
.rel	.tuples	number of tuples in the relation
.rel	.attributes	number of domains of the relation
.rel	.rvc	is a relation, view or computation
.rel	.sort	number of domains relation is sorted on (for implementation purposes only)
.dom	.dom_name	name of a domain
.dom	.type	type of the domain
.dom	.count	reference count for domain
.rd	.rel_name	name of a relation
.rd	.dom_name	name of a domain
.rd	.position	index of the domain in the relation

Table 4.1: System relations

relations, views, and computations may be created. It is intended that the last two items on this list also be thought of as special types of relations by the user. A view can be thought of as a virtual relation that only comes into existence when it is used, for instance in a **pr** command. A computation can be seen as an infinite relation. Calling a computation by either an ijoin or a select statement is like selecting a finite number of tuples from the infinite set of tuples contained within it. Of course, in the implementation, a computation cannot store an infinite amount of data, but instead evaluates one of its 'alt' blocks to produce a result. This is, nonetheless, a fruitful way of viewing computations. The syntax of jRelix supports this approach in as much as a view or a computation may be used anywhere that a relation can. Only at the semantic level are some restrictions enforced. Fortunately, these restrictions are few in number, and eliminating them is a topic of current research.

4.1.1 Data dictionary

The jRelix system keeps information about all the domains and relations that exist at any given moment in the form of system relations. On disk, this metedata is split up into three relations in accordance with good database design principles. The three relations are *.rel*, *.dom*, and *.rd*. The convention used in previous versions of Relix that system identifiers begin with a period is maintained in jRelix. This was useful since Relix was originaly intended to run on UNIX machines—as a matter of fact, the name Relix derives from the phrase RELational database on unIX—and in most UNIX systems, files which begin with a period are hidden. jRelix will run on any system which supports the Java run-time environment, therefore, on some of these the system files will not be hidden. The relation *.rel* contains information about relations, views and computations, while the *.dom* relation keeps track of domains. The *.rd* system relation provides the link between relations and domains. More specifically, it records which domains each relation is defined on. The domains of *.rel*, *.dom* and *.rd* are summarized in table 4.1 on the preceding page. The example in figure 4.1 on the next page will help clarify the situation.[1]

These system relations are intended for internal use by the system only. Certain fields, such as *.rvc*, are implemented as integer constants and need to be decoded. The dictionary for doing so is the class Constants. The casual user should use the **sr** command, not print the *.rel* table, to find out what relations are currently defined in the system. This command will not ordinarily display information about the sytem relations—those that begin with a period. However, the user can toggle the 'show system relations' mode. There is also a similar command to toggle the 'show system domains' mode that will cause the **sd** command to show system domains.

Syntax

```
ssr;
ssd;
```

The metadata about relations and domains is stored differently in RAM. Only two classes are defined for this purpose. They are the RelTable and the DomTable. The DomTable is essentially just a hash table that contains Domain objects keyed on their name, and a few useful access methods. The Domain object simply stores information about a single domain: the name, the type, a reference count of the number

[1]The figure is not entirely accurate. Information on the system relations is also kept in the system relations themsleves. For simplicity, this is not shown in the figure.

```
>domain BRAND, COUNTRY, SIZE string;
>domain PRICE float;
>relation cigars(BRAND, COUNTRY, SIZE, PRICE);
>pr .rel;
+--------------------+-----------+-------------+------------+------------+
| .rel_name          | .tuples   | .attributes | .rvc       | .sort      |
+--------------------+-----------+-------------+------------+------------+
| cigars             | 0         | 4           | 14         | 0          |
+--------------------+-----------+-------------+------------+------------+
>pr .dom;
+--------------------+-----------+-------------+
| .dom_name          | .type     | .count      |
+--------------------+-----------+-------------+
| PRICE              | 4         | 1           |
| BRAND              | 6         | 1           |
| COUNTRY            | 6         | 1           |
| SIZE               | 6         | 1           |
+--------------------+-----------+-------------+
>pr .rd;
+--------------------+--------------------+------------+
| .rel_name          | .dom_name          | .position  |
+--------------------+--------------------+------------+
| cigars             | BRAND              | 0          |
| cigars             | COUNTRY            | 1          |
| cigars             | PRICE              | 3          |
| cigars             | SIZE               | 2          |
+--------------------+--------------------+------------+
>
```

Figure 4.1: System relations

of relations which are currently defined on it, and code in the form of a syntax tree if the domain happens to be virtual. Similarly, the RelTable is a hash table which stores Relation objects keyed on their name. In this case, however, the Relation class is quite large because this is where the algorithms for implementing the relational algebra are defined. But the basic information is also there: the name, the rvc, and a syntax tree in the case of computations and views. An array of Domain objects, called domains, holding references to the domains that a given relation is defined on, is also kept since it is efficient to have this information handy. So there is no need for an in RAM version of the *.rd* relation, as this information has been absorbed into the Relation class. Figure 4.2 on the following page depicts the situation schematically for the *cigars* relation. A few system relations and domains have been included in the figure. The Relation objects have an array of pointers to Domain objects. However, only some of the pointers have been shown to avoid complicating the picture.

4.1.2 Representation of nested relations

The implementation of nested relations is built on top of flat relations. Whenever a domain of non-primtitive type is created, a relation of the same name, but prefixed with a dot, is automatically created by the system. It is intended that this relation be hidden from the user. That is, the implementation should not concern the user, and he or she should continue to think about the relation in the simple manner described in sections 2.1 and 2.5. The domains of this new relation are precisely those which make up the type of the domain which is being declared, as well as one additional attribute called '.id'. This is the surrogate field which the system will use to link up various components of the nested relation.

The relation *employees* from section 2.5 will serve as an example. In figure 4.3 on page 67 it is shown how the declaration of the nested domain *EMP* has led to the creation of the relation *.EMP* which, of course, is initially empty. Figure 4.4 on page 68 demonstrates how the creation of the nested relation *employees*, as shown in figure 2.27 on page 45, has caused tuples to be added to *.EMP*. Under the *EMP* field of *employees* are stored the surrogates. For example, the surrogate in the tuple containing "television" is 1017. By linking this surrogate to those in the '.id' field of

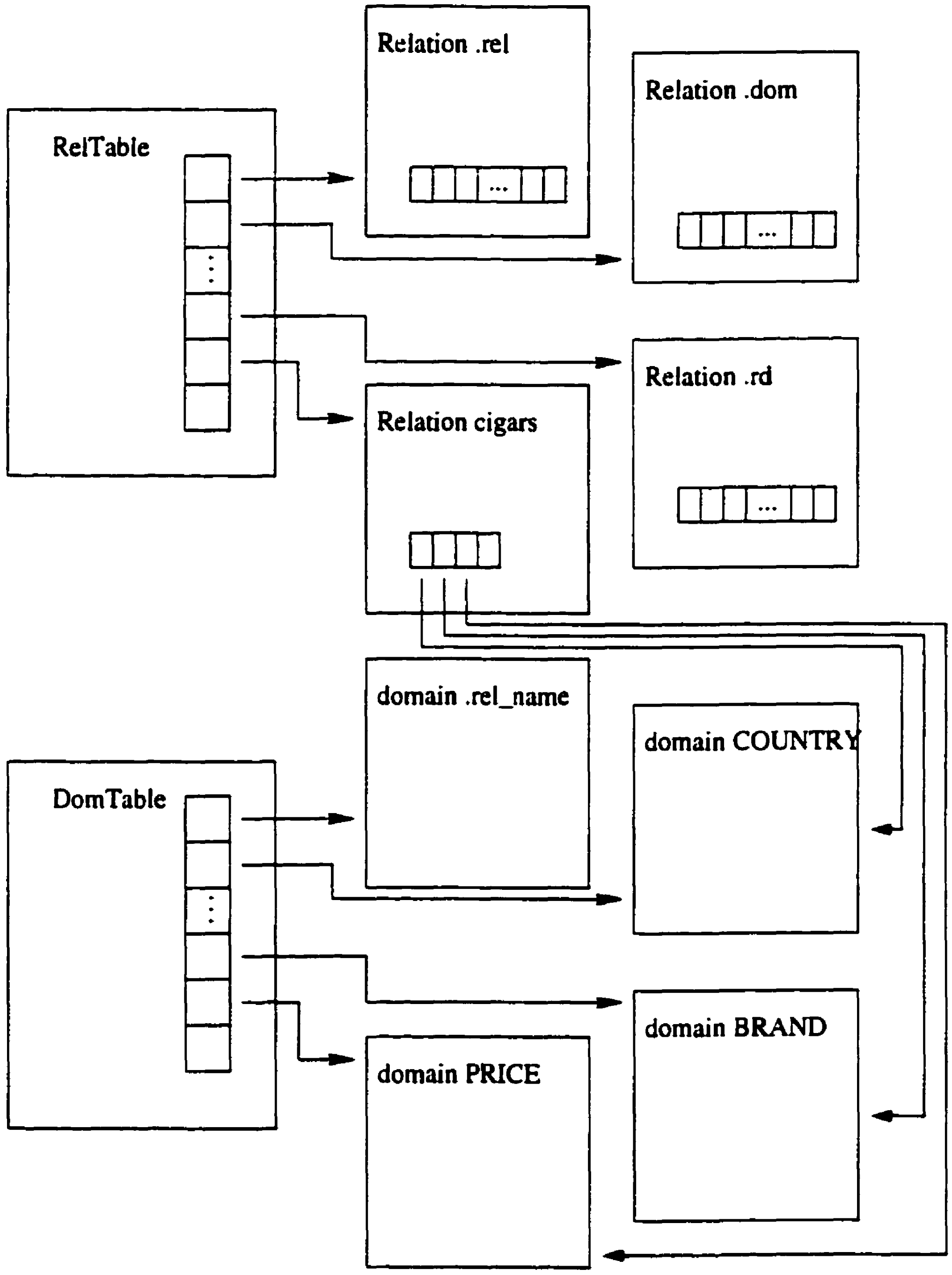


Figure 4.2: In RAM system metadata structures

```
>domain EMP(NAME, SAL);
>pr .EMP;
+----------------------+----------------------+-------------+
| .id                  | NAME                 | SAL         |
+----------------------+----------------------+-------------+
+----------------------+----------------------+-------------+
relation .EMP has 0 tuples
>
```

Figure 4.3: Implementation of nested domains, part 1

the hidden relation *.EMP*, it is found, for example, that the employees of the television department are J. Medeski, B. Martin, and C. Wood. All non-empty relations that will be defined on the domain *EMP* will cause tuples to be added to the relation *.EMP*.

The surrogates[2] are unique throughout the system so that there can never be any confusion as to which tuples of the hidden relation correspond to which top-level relation. This scheme is relatively simple and permits some optimizations, some of which were implemented in jRelix. For more information on this see [30].

4.1.3 Storage of relations

The method in which the data of a relation is stored in RAM is relevant to the implementation of computations. The way in which the data is stored on disk does not affect computations since the assumption is made that relations are small enough to fit in RAM. The data of a relation would be loaded into RAM before being used by a computation. See [30] for more information on the persistent storage of relations.

The class Relation contains a field called 'data'. This field is an array of Java Objects. In Java, Object is the ultimate ancestor of every class. Each object in this array is also intended to be an array. The data of a relation is stored here in column major format. The objects will be arrays of the types indicated by the domains of the relation. This proves to be an efficient mechanism both in terms of storage and

[2]Surrogates are implemented as 64 bit integers.

```
>pr employees;
+----------------------+----------------------+
| DEPT                 | EMP                  |
+----------------------+----------------------+
| stereo               | 1016                 |
| television           | 1017                 |
+----------------------+----------------------+
relation employees has 2 tuples
>pr .EMP;
+----------------------+----------------------+--------------+
| .id                  | NAME                 | SAL          |
+----------------------+----------------------+--------------+
| 1016                 | J. Fishman           | 44000        |
| 1016                 | M. Gordon            | 35000        |
| 1016                 | P. McConnel          | 32000        |
| 1016                 | T. Anastasio         | 27000        |
| 1017                 | B. Martin            | 38000        |
| 1017                 | C. Wood              | 32000        |
| 1017                 | J. Medeski           | 35000        |
+----------------------+----------------------+--------------+
relation .EMP has 7 tuples
```

Figure 4.4: Implementation of nested domains, part 2

execution time. Operations that involve columns, such as projection, are implemented by swapping pointers. The actual data of the relation does not need to be touched. Also, arrays are implemented efficiently in Java. Java interpreters use system specific interfaces in order to optimize operations on arrays such as the copying of large blocks of data from one array to another.

4.2 Parsing a JJTree Syntax Tree

Former versions of Relix were written in C and made use of Lex [40], a lexical analyzer, and Yacc [34], a parser generator, for their front end. The combination of these two tools produced a parser for the Relix interpreter. The front end of jRelix was written using a similar tool, the JavaCC compiler compiler [57]. JavaCC conveniently combines the two jobs of lexical decomposition and parser generation into one. Only a single source file is necessary. An additional tool called JJTree was also used [57]. A source file containing the specification of the jRelix syntax annotated with instructions for JJTree is fed into JJTree which produces an output file which is then given to JavaCC for further processing. The result is a parser for jRelix, written in Java, that also produces syntax trees according to the annotations that were provided in the source code. Figure 4.5 on the following page summarizes this process. The jRelix interpreter traverses these syntax trees and responds accordingly. The syntax trees for virtual domains, views and computations are kept in the system until the corresponding item, that is relation or domain, is explicitly deleted by the user. When jRelix is shut down these syntax trees are saved to disk using the Java object serialization mechanism [23]. Figure 4.6 on the next page illustrates a syntax tree which is created by JJTree for a simple jRelix statement.

The structure of each node in the syntax tree is specified in the class SimpleNode. Each node has four public fields which are of interest to a systems programmer. These are the *name*, *type*, *opcode*, and *bits* fields. The *type* field indicates what the node represents. For example, the type could be 'declaration', 'selection', or 'identifier'. These values are stored as integers. The class Constants defines all of these integer constants. The *opcode* field is really just a subtype, and it is an integer as well. For

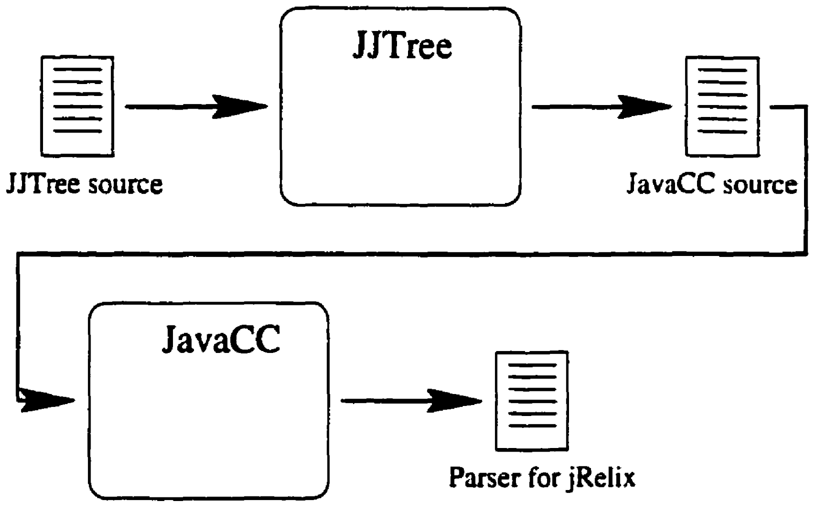


Figure 4.5: Tools used to create the jRelix parser

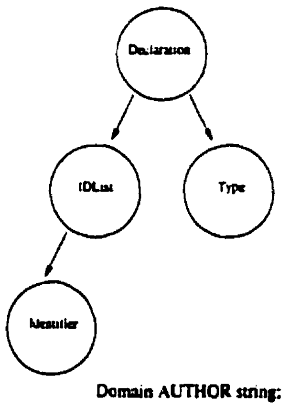
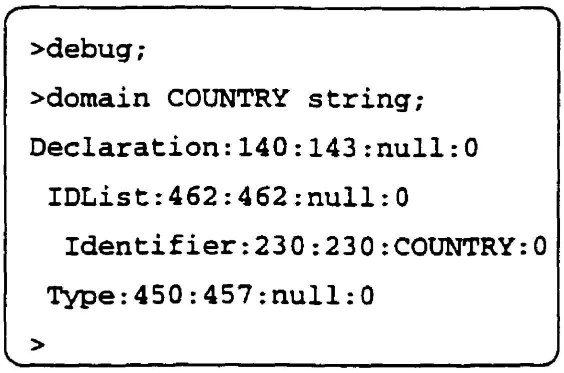


Figure 4.6: JJTree syntax trees



```
>debug;
>domain COUNTRY string;
Declaration:140:143:null:0
 IDList:462:462:null:0
  Identifier:230:230:COUNTRY:0
 Type:450:457:null:0
>
```

Figure 4.7: Debug mode

example, if the type is 'declaration' the subtype could be 'computation'. The *bits* field is used to pass extra information in a few special cases. The *name* field records the name of the item which the node represents when this is pertinent. The user can make jRelix dump the syntax trees of the statements it is interpreting by toggling the debug mode.

Syntax

```
"debug" ";"
```

By default, debugging is off. Figure 4.7 shows an example of this. Five items, separated by colons, are displayed per node. The first item is simply the identifier for the type of the node. This is only included for human readability. It expresses in english the meaning of the integer constant found in the next field. In the example there are nodes of type 'Declaration', 'IDList', 'Identifier' and 'Type'. The remaining fields are the *type*, *opcode*, *name* and *bits* fields mentioned in the previous paragraph. Consider the Identifier node. Its *type* and *opcode* are 230 which means identifier. The *name* field is set to COUNTRY which is the name of the item which this node represents. The *bits* field is set to zero since it is not necessary to record any special information for this type of node.

4.3 Environments in jRelix

This section examines the environment model for storing variables which is used by many programming languages. The model is not an absolute one. Rather, it comes in several different flavours. These are introduced and explained with examples in order to clarify the differences among them. Subsequently, a new model, termed the static environment model, which is better suited to database programming languages is proposed. Procedures are intimately involved with environments for it is their creation and application which cause the environment to expand and then shrink. It is also conceivable for the environment to expand when a thread of execution of a program enters a new block, but this is not the focus of the present document. In fact, many programming languages do not expand the environment in such cases. jRelix, in particular, does not. It will also be shown how objects with local state variables having accessor and mutator methods can be modeled in programming languages which provide these various environment models. Some of the discussion in this section is modeled on that of [2, pages 184-199].

4.3.1 The environment model

An environment is an object which holds bindings of variables to their values. These variables may represent primitive data such as an integer, or complex data like an object or procedure. It is not sufficient for environments to simply associate names with values. A programming language which supports the concept of nested scopes, for instance, permits variables with the same name to exist concurrently as long as they are in different scopes. In terms of an environment, we say that the variables are bound in different frames of the environment. An environment consists therefore of many frames which are linked together by means of parent pointers. For instance in figure 4.8 on the next page, a look up of the variable 'a' in frame f3 will return the binding that is found in the frame f3. The binding of 'a' in f1 is said to be shadowed by that in f3. A look up of 'a' with respect to frame f4, however, would force the system to follow the pointer from f4 to f2 because 'a' is not in f4. As there is no binding for 'a' in f2, the search continues in f2's parent frame which is f1. Finally a binding for 'a' is found and can be returned. In figure 4.8, the two bindings for 'b' can never shadow each other since neither of the frames to which they belong is an ancestor of the other. It is therefore correct to think of an environment as providing places for variables to be stored, not just names. It is clear, then, that a variable, by

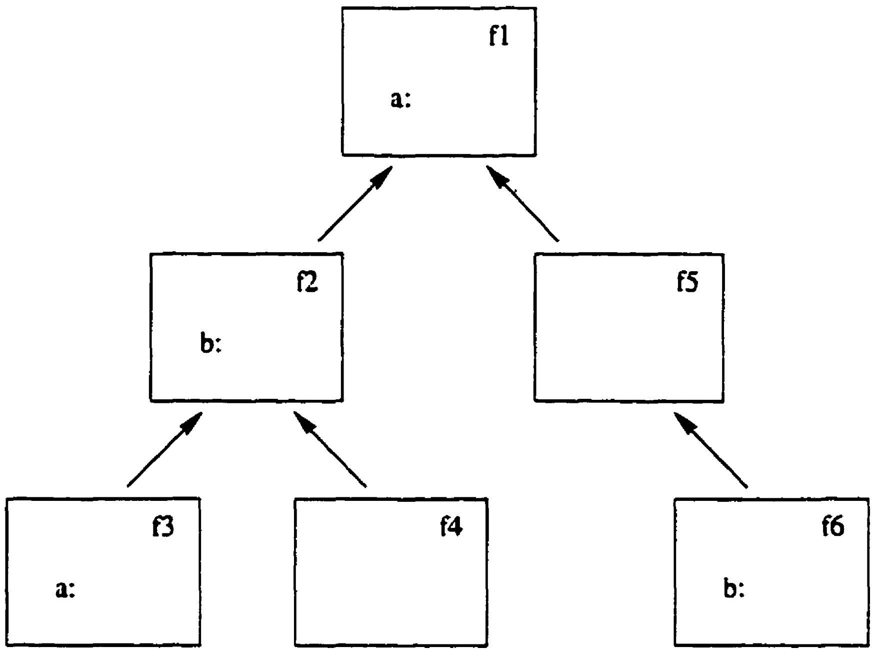


Figure 4.8: Variable look ups in an environment

itself, is utterly useless. It only acquires meaning when coupled with an environment. Any statement of a program must be evaluated with respect to some environment.

There is a 'global' frame that is initially created by the system. This is the common ancestor of all the frames that will ever exist during the execution of a program. New frames are created when a thread of execution enters a new scope. A procedure call or a new block of statements are the typical causes of this. It is the latter case that concerns us here. We will first consider the rules for procedure declaration and application used by common programming languages, such as PASCAL [33] and SCHEME [1] (a statically bound dialect of LISP). They are as follows:

A procedure is created by evaluating a declaration relative to an environment. The resulting procedure object is a pair consisting of the code of the procedure, as well as a pointer to the environment in which the declaration was evaluated. A binding of the procedure's name to the newly created procedure object is then placed in this environment.

To apply a procedure, a new frame is created in which the procedure's formal parameters are bound to the values specified in the call. This new frame is attached to the environment by setting its parent pointer to

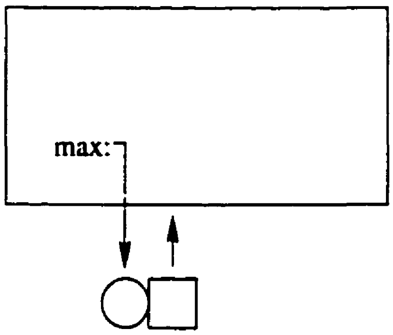


Figure 4.9: Max computation, part 1

point to the frame indicated by the environment portion of the procedure object. The code of the procedure object is then evaluated relative to this new environment.

This method is called static binding or lexical scoping. Figures 4.9 and 4.10 on the following page illustrate these rules for a simple example program.

Program 1

```
computation max(a, b, c) is
{  if (a <= b) then
   {  c <- b;
   }
   else
   {  c <- a;
   };
};
```

The declaration of this procedure in the global environment results in a binding of *max* to a procedure object pair. In the diagrams that follow, the circles represent the code portion of the procedure object pair, while the squares denote the environment pointer. Note that in figure 4.9 this pointer has been set to the environment in which *max* was declared. Figure 4.10 shows the state of the environment just after the following call to *max*.

```
C <- max[2,3];
```

A new frame has been created, and in it 'a' and 'b' are bound to the values that were supplied in the invocation of *max*. The new frame has the global environment as its

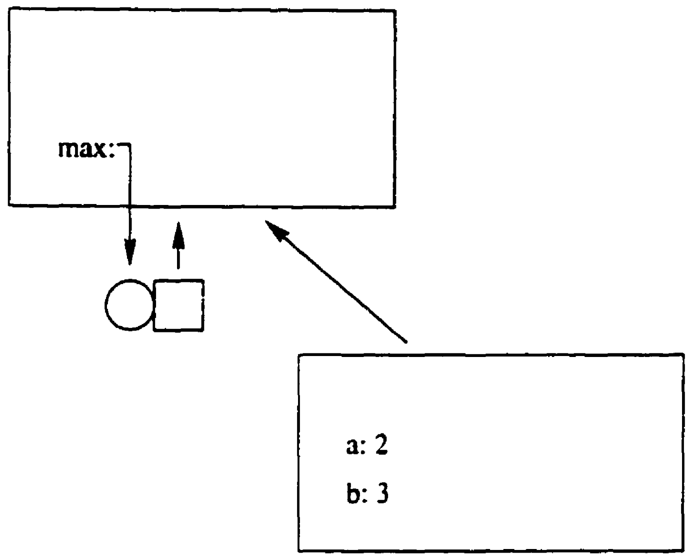


Figure 4.10: Max computation, part 2

parent frame because that is what the environment portion of the procedure object for *max* specified.

Alternatively, a slightly different set of rules could be used:

> A procedure is created by evaluating a declaration. A binding of the procedures name to the newly created procedure object is then placed in the current environment.
>
> To apply a procedure, a new frame is created in which the procedure's formal parameters are bound to the values specified in the call. This new frame is attached to the environment by setting its parent pointer to the current environment in which the procedure call is made. The code of the procedure is then evaluated relative to this new environment.

This method is called dynamic binding, because the unbound variables of a procedure get their values from the environment in which the procedure was called, instead of that in which it was declared. This is a consequence of the manner in which the parent pointer of the new frame is set—namely, it points to the environment in which the procedure *call* is made. With static binding, a pointer to the environment in which the procedure is *declared* is stored, and is later used as the parent pointer of the new frame created when the procedure is applied. To summarize, with static binding the unbound variables are looked up in the environment in which procedure is *declared*, and with dynamic binding the unbound variables are looked up in the environment in which the procedure is *called*. The programming language LISP [61] implements the discipline of dynamic binding.

The next example program illustrates the difference between static and dynamic binding.

Program 2

```
computation power(x,y)
{  y <- exp(x,n);
};

computation sum_power(a,b,c,n) is
{  c <- power[a] + power[b];
};
```

The invocation is

```
c <- sum_power[9,10,,3];
```

Figures 4.11 and 4.12 depict the structure of the environment just after the call to power[a] in the body of *sum_power* has been made. Figure 4.11 on the next page shows the result for the case of static binding and figure 4.12 on the following page does the same for dynamic binding. The essential difference is with the parent pointer of frame f3. With static binding, a pointer to frame f1 is stored at declaration time in the procedure object for *power*. This pointer is used as the parent pointer of frame f3 when *power* is called. However, with dynamic binding, the parent frame of f3 is f2 since that is the frame in which the call to *power* is made. The result is quite different. In the latter case the free variable 'n' of *power* is found in frame f2, whereas in the former case this variable is unbound with respect to frame f3 causing an error to be reported.

For safety, however, we decided to use static binding in jRelix, although this hampers us from running code such as that in program 2. We will not be concerned with dynamic binding any further.

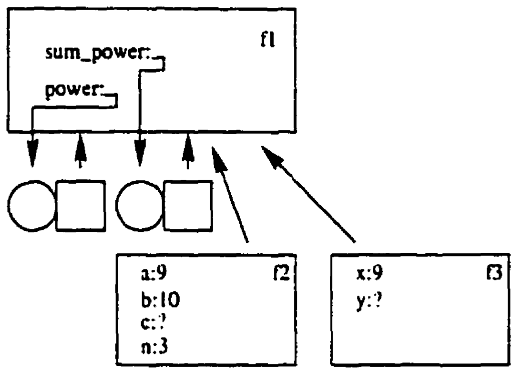


Figure 4.11: Static binding

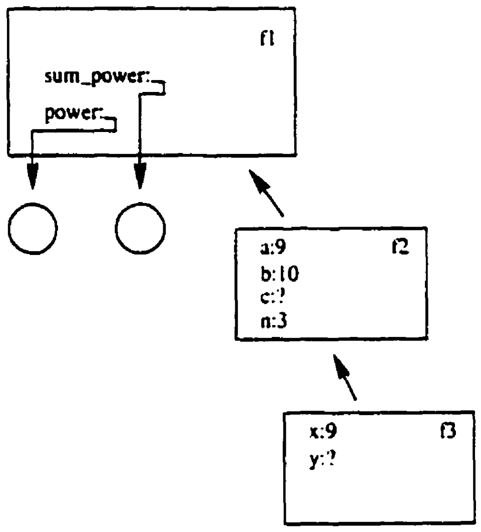


Figure 4.12: Dynamic binding

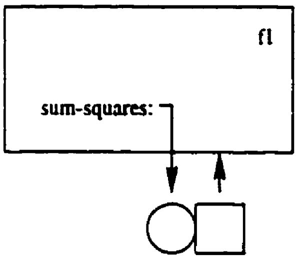


Figure 4.13: Sum of squares, part 1

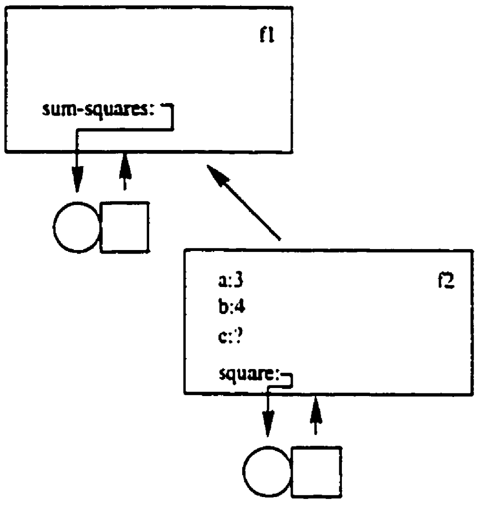


Figure 4.14: Sum of squares, part 2

The following example includes a nested procedure.

Program 3

```
computation sum_squares(a,b,c)is
{  computation square(a,b) is
    {  b <- a*a;
    };
    c <- square[a] + square[b];
};
```

```
h2 <- sum_squares[3,4];
```

After this declaration has been processed, the situation is as in figure 4.13. Figure 4.14 is a snapshot of the environment just after the call to *sum_squares* has been made and the nested declaration for *square* has been processed. There is now a binding in frame f2 for the procedure *square*. The environment pointer of *square* points to f2 since that is the environment in which that procedure was declared. Frame f2 also

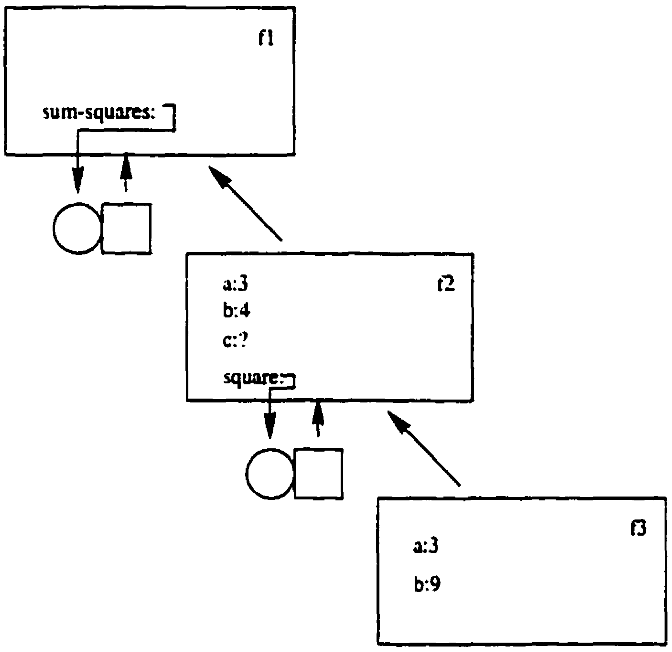


Figure 4.15: Sum of squares, part 3

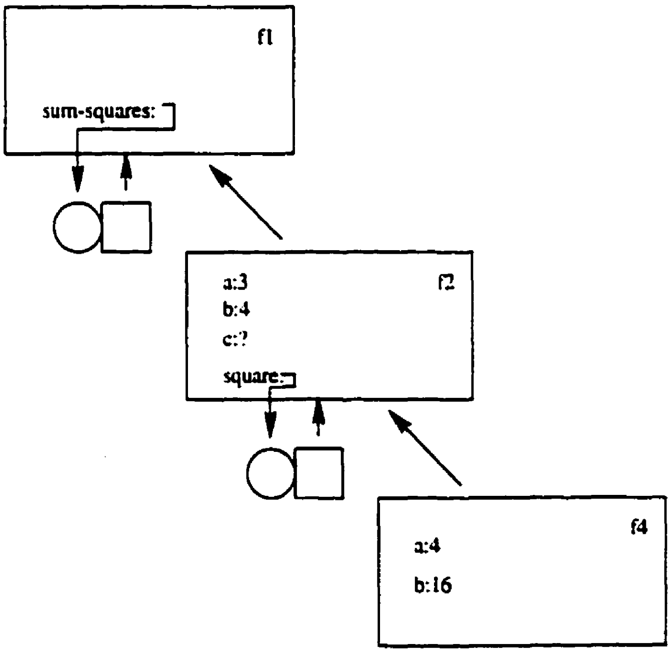


Figure 4.16: Sum of squares, part 4

contains bindings of all the formal input parameters to the values specified by the call. Now when the next line of code in *sum_squares* is executed, two successive invocations of *square* are made. Figure 4.15 contains the state of the environment after the first of these is begun. A new frame f3 has been created in which 'a' has been bound to 3 since that is the value of 'a' in frame f2. Note that the 'a' in f3 shadows the one in f2. When the second call to *square* is processed, the new frame, f4, is created. Frame f3 is gone and the memory used by it will be reclaimed. This is illustrated in figure 4.16. In frame f4, 'a' has been bound to 4, and once again shadows the 'a' in frame f2.

4.3.2 Creating objects with state

In the previous example, the nested procedure *square* was local to *sum_squares*. Some programming languages such as LISP and Relix allow a procedure to be returned as the value of another procedure. In jRelix, for example, the type of an input or output formal parameter can be a computation. Coupling an environment model with this capability of manipulating procedures provides a programming paradigm sufficiently powerful to enable the modeling of objects with local state having proper accessor and mutator methods. The following example shows how this may be done.

Program 4

```
computation make_bank_account(init_balance, deposit, balance)
  local bal intg;
{  bal <- init_balance;
   deposit(amount)
   {  if ( bal+amount >= 0) then
      {  bal <- bal+amount;
      }
      else
      {  print "error: not enough funds for withdrawal";
      };
   };
   balance(b)
   {  b <- bal;
   };
};
```

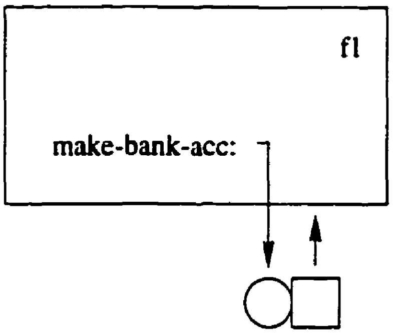


Figure 4.17: Bank account computation, part 1

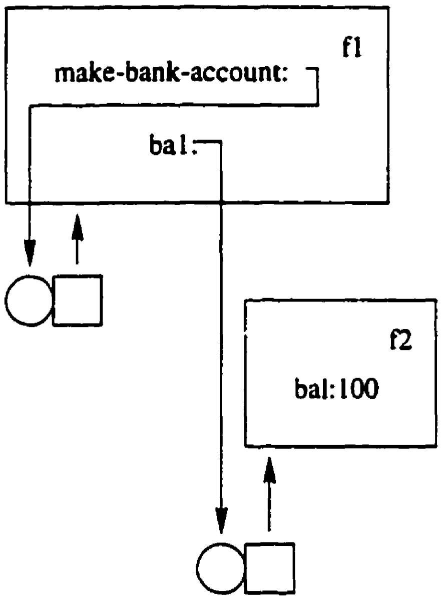


Figure 4.18: Bank account computation, part 2

Figure 4.17 shows the effect of declaring this procedure in the global environment. Figure 4.18 shows the environment after the following invocation.

```
bal <- make_bank_account[100];
```

The procedures *deposit* and *balance*, which are nested inside of *make_bank_account*, are passed out as return values through parameters. The result relation *bal* is defined on the domains *deposit* and *balance*, and has a single tuple which contains references to the *deposit* and *balance* procedure objects just created. These then become available for use outside of *make_bank_account*. Indeed, they can be accessed from *bal* which is bound in the global frame. Because of this, we will say that *deposit* and *balance* are exported procedures, in contrast to the local procedure *square* in the previous

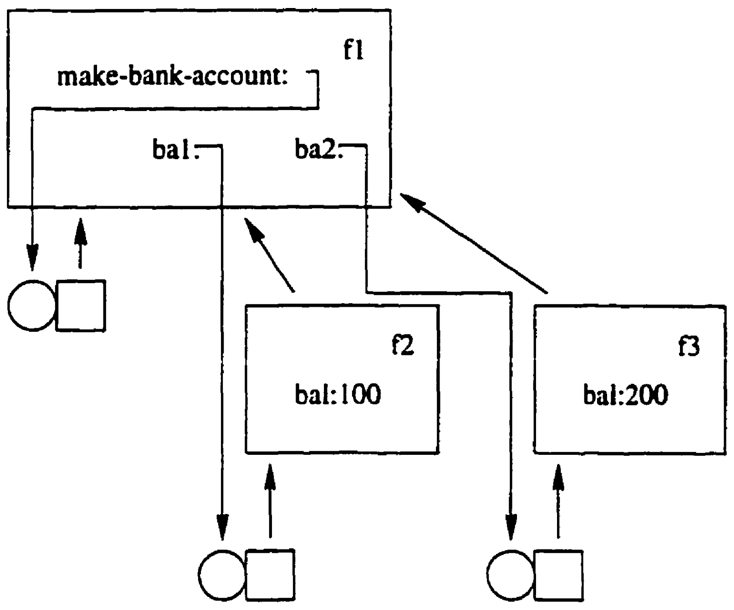


Figure 4.19: Bank account computation, part3

example which remained local to *sum_squares*. Each invocation of *make_bank_account* creates a new bank account which also provides accessor and mutator methods for that new bank account. Figure 4.19 shows the environment after a second bank account is created by the following statement.

```
ba2 <- make_bank_account[200];
```

According to the rules of procedure application stated above, this call produces yet another frame. In this frame, the variable *bal* currently has the value 200. This *bal* is different from the one in frame f2 which has the value 100. What this example has accomplished is the creation of two objects each having their own state variable and access methods. A call to the *deposit* procedure stored in the relation *ba2* will update the *bal* in frame f3 because the environment pointer of its procedure object is set to the environment in which it was declared, namely, frame f3.

4.3.3 The static environment model

In a realistic database application, it would not be unusual if there were thousands of bank accounts. If the approach just outlined was adopted, this would require equally many new frames to be created and to persist. Such an approach would be very expensive in terms of memory usage, or disk space if the environments can be

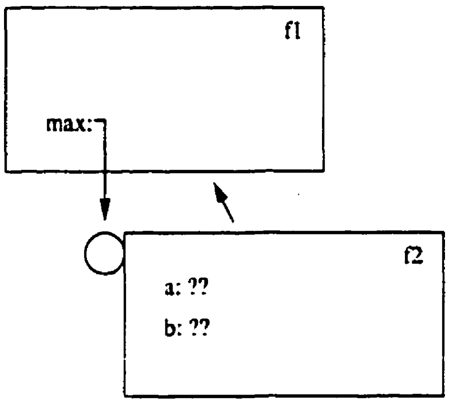


Figure 4.20: Static environment, part 1

swapped to disk. In light of this, a new modified environment model is proposed. The new rules for procedure application and declaration are:

> A procedure is created by evaluating a declaration relative to an environment. The resulting procedure object is a pair consisting of the code of the procedure, as well as a pointer to a new frame. This new frame has as its parent the environment in which the procedure was declared.
>
> A procedure is applied to a set of arguments by binding its formal parameters, to the actual arguments provided by the call, in the environment which is pointed to by the environment portion of the procedure object.

Program 1 on page 74 will be used to illustrate how these rules work. Figure 4.20 shows the situation after the computation has been declared. The computation object does not just contain a pointer to its parent frame, it contains the entire frame f2. This will be used when the computation is called to bind the formal parameters to the values provided by the call.

Figure 4.21 shows the environment just after the following call is begun.

```
C <- max[2,3];
```

The variables 'a' and 'b' have been assigned the values 2 and 3, respectively, in frame f2. As shown in figure 4.22, this frame is reused again when the following call is made.

```
C <- max[4,6];
```

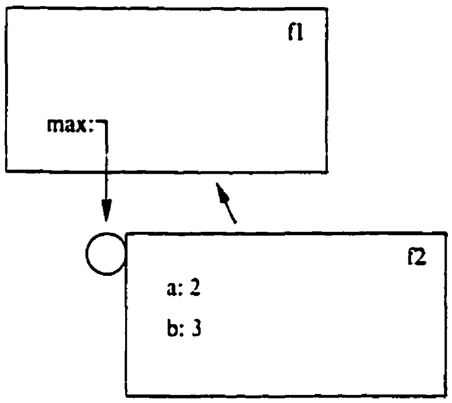


Figure 4.21: Static environment, part 2

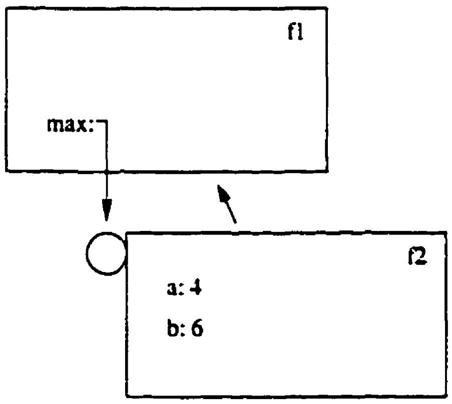


Figure 4.22: Static environment, part 3

The essential difference between this model and the previous ones is that the new frame, which is to be used for binding the formal parameters and as the evaluation context of the procedure, is created when the procedure is declared instead of when it is applied. In this new version a single environment is simply reused each time the procedure is called instead of creating new environments each and every time. For this reason, we call the new approach the static environment model, and the original model will be referred to as dynamic. This is not to be confused with dynamic and static binding. The feature which distinguishes dynamic binding from static binding is how the parent pointer of the new frame is set. The difference between the static and dynamic environment models is completely orthogonal to the issue separating static and dynamic binding. It is whether the new frame is created when a procedure is declared or when it is applied. The two properties can therefore be combined in four different ways to provide four alternative programming paradigms.

If we reconsider the previous examples in terms of this new model, we find that it behaves similarly to the original one except for the *make_bank_account* example. The reason for this is quite simple. It is only in the case where a procedure is exported that a frame which was created due to a procedure call lingers on after the call is completed. This is how stateful objects were manufactured. A weakness in this approach has, then, been uncovered. If we wish our language to provide the ability to model objects with state, and in the case of jRelix we certainly do, then some other mechanism must be provided. The static environment model alone will not suffice. Section 4.6 discusses the approach adopted by jRelix for handling stateful objects.

4.3.4 Implementation details

In this section the implementation of environments is discussed. As was explained in section 4.1.2, relations are stored in a RelTable object and domains in a DomTable object. Each frame of the environment therefore includes both a RelTable and a DomTable object. In order to support computations, each frame contains an additional hash table, called *table*. The purpose of this hash table is to provide storage space for the formal parameters, local variables and state variables of a computation—in other words, whatever the RelTable and DomTable are not designed to handle. The

Object	Description
reltable	A hash table used to store relations, views and computations
domtable	A hash table used to store domains
table	A hash table used to store parameters and local and state variables of computations
parent	pointer to the parent Environment
NRenv	pointer to an NREnvironment
data	column major array of arrays to store data of a relation
row	row number of the relation contained in data currently being processed

Table 4.2: Component structures of the Environment class

fields of the Environment class are summarized in table 4.2.

The existence of five different 'kinds' of variables—relations, domains, computation parameters, and local and state variables of computations—creates a significant amount of complexity for the systems programmer. When interpreting a statement, the need to look up variables in the environment will arise. It is too much trouble for the system programmer to have to consider what the 'kind' of each variable is. Thus, this complexity has been hidden by means of a sufficiently rich set of interface methods belonging to the Environment class.

An abstract base class called IDInfo is created from which the classes ParamInfo, LocalInfo and StateInfo are derived. These three classes are used to store information pertaining to a single computation variable and it is these that are inserted into the hastable. A RelInfo and a DomInfo class, which also derive from IDInfo, have been created so that the RelTable, DomTable and the third hash table all blend together seemlessly. The purpose of the abstract base class is to simplify the manipulation of these 'info' objects. The programmer can simply pass them around as IDInfo objects without having to check their specific type. These classes are summarized in table 4.3 on the following page.

Class	Description
IDInfo	Abstract class with one field called kind
ParamInfo	Stores a name/value binding for a parameter kind = PARAMETER Field storing the type of the parameter Field storing the index of the parameter
LocalInfo	Stores a name/value binding for a local variable kind = LOCAL Field storing the type of the variable Field storing the value of the variable
StateInfo	Stores a name/value binding for a state variable kind = STATE Field storing the type of the variable Field storing the index of the variable
RelInfo	Stores a name/value binding for a relation kind = REL Pointer to a relation object
DomInfo	Stores a name/value binding for a domain kind = DOM Pointer to a domain object

Table 4.3: Classes encapsulating the different kinds of variables

There are a few fields in table 4.2 that have not yet been discussed. The *parent* field contains a pointer to the Environment object which is the parent frame. Any unsuccessful look up of a variable in the current frame is followed by a look up in the parent frame. However, if the pointer happens to be null then, as this is the last frame that may be searched, the variable is not in found in the environment and null is returned. The *NRenv* field is discussed in the next section. The *data* and *row* fields are used when a computation is applied. The data field stores the contents of the input relation in column major format as described in section 4.1.3. The row field keeps track of the row of this relation which is currently being processed.

Now that the implementation of environments has been covered, it remains to explain how a programmer can make use of it. Table 4.4 on the next page lists the interface funtions that are available for this purpose. These fall into three groups. The polymorhic *put* methods are used to insert new bindings into the environment. There is a set of 'lookup' methods for retrieving items. The generic *lookup* method returns an IDInfo object. It is up to the system programmer to check the *kind* field to deduce which specific class the current object belongs to. This must be one of the classes, listed in table 4.3, that inherits from IDInfo. If a Relation object is expected, then the *lookupRel* method may be used instead. This relieves the programmer of dealing with an IDInfo object by simply returning a Relation object. The *lookupDom* and *lookupParam* methods behave similarly, except that they return Domain objects and ParamInfo objects, respectively. The last group consists of interface methods that are used to remove bindings from the environment.

4.3.5 Scopes induced by nested relations

There is a further notion of scope in jRelix which is orthogonal to that which has been discussed so far in this section. Each level in a deeply nested relation is itself made up of nested relations until, eventually, primitive data is reached. Each of these relations is defined on a collection of domains. It is possible to write domain algebra expressions which involve domains at different levels of such a deeply nested relation.

The example in figure 4.23 on page 90 will help to illustrate this. Only the scheme of the relation is given. It may be desired to join PT1 with PRESSURE (for example.

method	Input	Output
Insertion Methods		
put	name of variable	none
	IDInfo object	none
put	Relation object	none
put	Domain object	none
Retrieval Methods		
lookup	name of variable	IDInfo object
lookupRel	name of variable	Relation object
lookupDom	name of variable	Domain object
lookupParam	name of variable	IDInfo object
Removal Methods		
removeRel	name of variable	none
removeDom	name of variable	none

Table 4.4: Interface methods of the Environment class

to compute the pressure at some point on the bridge). Since PRESSURE is higher up in the nested hierarchy than PT1, the data in one relation under PRESSURE is regarded as constant with respect to all relations in that tuple that are stored under PT1.

During interpretation, the names PRESSURE and PT1 need to be resolved so that actions can be carried out. A look up into the environment will be undertaken so as to obtain the data that these vaiables currently represent. A class called NREnvironment has been created for this purpose. It has been embedded into the Environment object so that a programmer need not be aware of this situation, and therefore not have to write extra code. The look up method of the Environment class will search the NREnvironment before looking in the structures described in the preceding section.

When domain algebra is used to join PRESSURE with PT1, the actualizer, which is the class that implements the domain algebra [70], works its way down the hierarchy until it reaches PT1. As its descends the relation, it creates frames of the NREnvironment which record the domains that were seen on the way. For example, the first

```
bridge (OBJECT                    PRESSURE)
        (SEQNO  PT1   PT2  )(X Y P)
                (X Y) (X Y)
```

Figure 4.23: A deeply nested relation

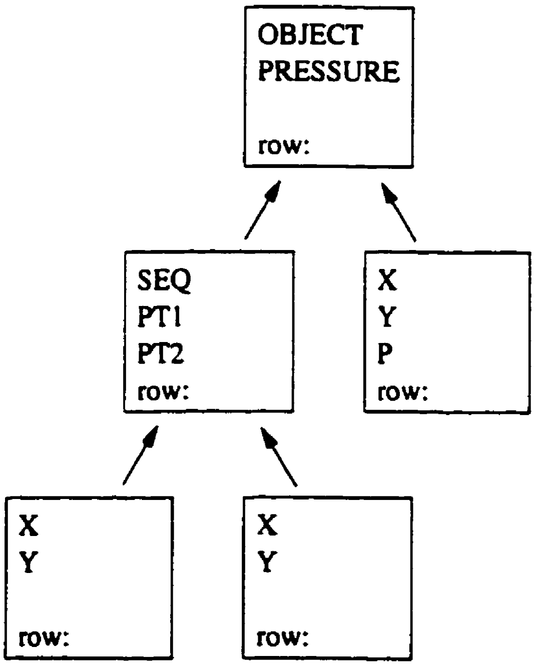


Figure 4.24: Nested-relation environment

NREnvironemt to be created in this case contains the domains OBJECT and PRESSURE. When the actualizer goes deeper into OBJECT, a new frame is created which contains SEQNO, PT1 and PT2. The expression involving the join of PRESSURE with PT1 is evaluated with respect to this environment. Therefore the look up of the variables in the environment will return the bindings found in these NR-frames. As the actualizer descends the relation hierarchy and creates the NR-frames, one more piece of information must be kept track of—the row number. At a given moment in the interpretation it is possible to be in the n^{th} row of the relation PT1 and in the m^{th} row of PRESSURE. Thus many row numbers must be kept track of. In fact one per NR-frame is precisley what is needed. When the system programmer does a look up in the environment, the interface method looks at the row number in the NR-frame and returns the correct information. Thus this aspect is also made transparent to the programmer. Figure 4.24 on the page before illustrates the NR-environment that is created for our example.

4.4 Application of Computations

Computations may be applied in two different ways—by means of an ijoin or by a selection. The algorithms used to implement these are provided below.

Algorithm: Natural join

1. Create and initialize a new environment frame.
2. Set this frame's parent pointer to that stored within the computation.
3. Determine the join attributes.
4. Calculate a block type from the join attributes.
5. Locate the matching 'alt' block. If not found throw an error.
6. Load the data of the input relation into memory.
7. Create a new relation to hold the output.

8. For each tuple in the relation do:
    (a) Execute each statement of the 'alt' block.
    (b) Append the resulting tuple to the output relation.
9. Detach the new environment frame so that it will be garbage collected.
10. Project the output relation on all its attributes to remove duplicate tuples.
11. Return the output relation.

Algorithm: Select

1. Create and initialize a new environment frame.
2. Set this frame's parent pointer to that stored within the computation.
3. Determine the input attributes.
4. Calculate a block type from the input attributes.
5. Locate the matching 'alt' block. If not found throw an error.
6. Create a new relation to hold the output.
7. Execute each statement of the 'alt' block.
8. Append the resulting tuple to the output relation.
9. Detach the new environment frame so that it will be garbage collected.
10. Project the output relation on all its attributes to remove duplicate tuples.
11. Return the output relation.

Algorithm: Stand-alone

1. Create and initialize a new environment frame.
2. Set this frame's parent pointer to that stored within the computation.

3. Calculate a block type from the input attributes.

4. Locate the matching 'alt' block. If not found throw an error.

5. For each input parameter do:

   (a) Create a local variable having the name of the corresponding parameter.

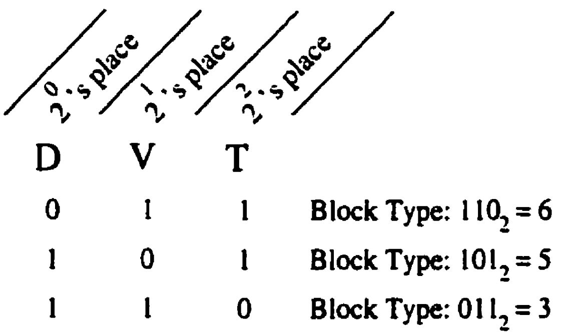
   (b) Initialize this variable with the value supplied for that parameter.

6. Execute each statement of the 'alt' block.

7. Copy the value of each parameter out to the calling environment

8. Detach the new environment frame so that it will be garbage collected.

Each time a computation is invoked, a new frame is created in order to hold data, as specified by the formal parameters. Computations are, however, quite different from the type of procedures normally found in conventional programming languages. In the latter case, when a procedure is called it is implicit that exactly one piece of data will be substituted for each formal parameter. Computations, on the other hand, are intended to work efficiently with large amounts of data and it is expected that each formal parameter will be substituted for many times. Instead of creating a new frame in which to bind the formal parameters to the data each time a block of a computation is executed, which would amount to creating as many frames as there are tuples in the relation being joined with the computation, only a single frame is created which will handle all of the data. The slots in this frame will simply be reused over and over again. This is how jRelix makes use of the static environment model that was discussed in section 4.3.3. Note that this new frame is not created when the computation is defined, but rather, when it is invoked. This may seem like the dynamic environment model. It is best to say that the approach that jRelix takes is somewhere in between. What is to be avoided is the creation and initialization of new frame objects each time a block of a computation is run, which is to say for each tuple of the input relation. The problem with creating the new frame at declaration time is that it precludes the possibility for the language to allow recursive computations. In order to support recursion, each instance of the computation that is currently pending



D	V	T	
0	1	1	Block Type: $110_2 = 6$
1	0	1	Block Type: $101_2 = 5$
1	1	0	Block Type: $011_2 = 3$

Figure 4.25: Block type

must have a frame to store its data. If there were only one frame per computation, and not one frame per invocation, it would not be possible to save the context of each suspended instance of the computation.

Regardless of the method of invocation, a necessary step is to deduce which of the 'alt' blocks to execute. The parameters of the computation are viewed as place holders for binary digits, but instead of using the customary left to right order, the number must be read backwards from right to left. Figure 4.25 illustrates this for the velocity computation which was shown in figure 3.1 on page 47. The parameters, from left to right are *D*, *V* and *T*. The three rows of digits correspond to the three 'alt' blocks of the velocity computation. A '1' indicates that the variable is an input parameter of the block, a '0' that it is an output. The type of the first block is 110 in binary notation, that is 6. This is just the binary digits of the first row read backwards. The digits have to be read backwards because the least significant digits occur to the left instead of to the right, as with our costumary positional numeral system.

4.5 Creation of Computations

The declaration of a computation leads to the instantiation of a Computation object. The Computation class contains methods for processing the declaration and application of computations as well as for responding to commands that apply to computations. The constructor of this class is called by the interpreter to handle a declaration. It requires as inputs a reference to the syntax tree and a pointer to the environment in which the declaration occurs. The steps involved are:

Algorithm: Declaration

1. Store the environment pointer.
2. Parse the syntax tree.
3. Calculate the block type of each 'alt' block.
4. Update the system tables to include an entry for the new computation.

The environment pointer is stored to implement the discipline of static binding. When the computation is called, a new frame will be created. Its parent frame will be set to that indicated by this pointer. The syntax tree is traversed in order to deduce the parameters, local and state variables, 'alt' blocks, as well as the name of the computation. This information is used by the block type algorithm.

The body of a computation consists of a sequence of 'alt' blocks. When the computation is invoked, it must be decided which of these applies. Each 'alt' block corresponds to exactly one set of input parameters. The purpose of the block type algorithm is to assign an integer 'type' to each 'alt' block of a computation. It is stored in the computation object so that it need not be re-evaluated each time the computation is invoked. As described in the previous section, this integer, when expressed in binary notation, is a bit string of 1's and 0's that indicates the input parameters of the block. The block type algorithm marks each variable that appears in a computation as either an input or an output. This decision is based on the way in which the variable is *first* used. Hence, the marking of a variable is permanent. The block type algorithm is complicated by the inclusion of nested computations. To simplify the discussion, we first describe the algorithm for the case where there are none these.

Algorithm: Block Type 1

1. Initialize the markings: All parameters and local and state variables have their marking set to RELIX_VOID.
2. For each statement of the 'alt' block:

(a) Deduce the input parameters of the statement.

(b) Mark these as input parameters if their current marking is RELIX_VOID.

(c) Deduce the ouput parameters of the statement.

(d) Mark these as output parameters if their current marking is RELIX_VOID.

3. Assign a type to the 'alt' block based on the input parameters.

We now explain how the input and output variables may be deduced for each type of statement provided by jRelix.

Assignment < id > '<-' **expr**

Inputs: all identifiers in **expr**

Outputs: < id >

Remarks: There are two exceptions to the rule for inputs. The identifiers of a projection list and of a selection clause are excluded[3].

	Statement	Inputs	Outputs
Examples:	R <- [A,B] in S ijoin T	S, T	R
	R <- where A=2 in S	S	R

Incremental Assignment < id > <+ **expr**

Inputs: all identifiers in the statement

Outputs: none

Remarks: There are two exceptions to the rule for inputs. The identifiers of a projection list and of a selection clause are excluded.

	Statement	Inputs	Outputs
Examples:	R <- [A,B] in S ijoin T	S, T, R	None
	R <- where A=2 in S	S, R	None

Domain Declaration domains < id > **type**

Inputs: none

Outputs: none

[3]The reason for these exceptions is discussed in section 5.2.5 on page 110.

Virtual Domain let < id > be expr

Inputs: none

Outputs: none

Relation Declaration relation < id > (< idList >) <- expr

Inputs: all identifiers in expr

Outputs: none

Examples:

Statement	Inputs	Outputs
relation R(A,B)	None	None
relation R(A,B) <- S	S	None

Relational view < id > is expr

Inputs: all identifiers in expr

Outputs: < id >

Remarks: There are two exceptions to the rule for inputs. The identifiers of a projection list and of a selection clause are excluded.

Examples:

Statement	Inputs	Outputs
R is [A,B] in S ijoin T	S, T	R
R is where A=2 in S	S	R

Update Add,Delete update < id > add expr

Inputs: < id > and all identifiers in expr

Outputs: none

Example:

Statement	Inputs	Outputs
update R add S ijoin T	R, S, T	None

Update Change update < id > change `stmtList` using `JoinOperator` expr

Inputs: < id >, all inputs in `stmtList` and all identifiers in expr

Outputs: none

```
>comp C(X,Y) is
{  comp nested(Z) is
   {  Z <- Y;
   };
   X <- nested[2];
};
```

Figure 4.26: Nested Computation

Remarks: The `stmtList` may contain any jRelix statements. Since the output attributes of these statements are assumed to be domains of < id >, they do not play a role in the block algorithm. The inputs of these statements are considered inputs by the block algorithm.

Example:

Statement	Inputs	Outputs
update R change A <- 2*C using ijoin S	R, C, S	None

We now complete our exposition of the block type algorithm by describing the additional steps required in the presence of nested computations. The declaration of a nested computation is ignored. It is when a nested computation is called within the 'alt' block that extra care must be taken. Consider the example provided in figure 4.26. The variable *Y* is an input parameter to computation *C*, even though it is only used directly by the computation *nested*. If *nested* is never called within *C*, then *Y* is not an input paramater for *C*. This is why the nested computation is only processed when it is called, and not when it is declared.

Algorithm: Block Type 2

1. Initialize the markings: All parameters and local and state variables have their mark set to RELIX_VOID.

2. For each statement of the 'alt' block:

   (a) Deduce the input parameters of the statement.

   (b) Mark these as input parameters if their current marking is RELIX_VOID.

   (c) Deduce the ouput parameters of the statement.

(d) Mark these as output parameters if their current marking is RELIX_VOID.

(e) If a nested computation is invoked, run the block type algorithm on that computation.

3. Assign a type to the 'alt' block based on the input parameters.

There is only one additional step in this algorithm. It is a recursive call to the block type algorithm. In the example of figure 4.26, this occurs when the computation *nested* is called by *C*. The statement in *nested* identifies *Y* as an input and marks it as such.

Just as with relations and views, a computation that has been successfully declared will be added to the system tables. This is accomplished by adding the new Computation object to the RelTable. The class Computation inherits from the Relation class (once again demonstrating the influence of the principle that a computation 'is a' relation on the implementation). Thus, it has a copy of the data dictionary variables. Not all of these fields apply to computations however. The *name* field is set to the name of the computation, the *rvc* field is set to indicate a computation. The *numtuples* and *numsortattrs* fields are always set to 0 as these fields do not apply to computations. The *numattrs* field is set to the number of parameters that the computation is defined on. The syntax tree for the computation declaration is stored in the *tree* field.

4.6 Implementation of Stateful Computations

It has been noted that a shortcoming of the static environment model is its inability to model objects with state. This section explores the approach taken in jRelix to overcome this limitation.

The keyword **state** is introduced into jRelix as a modifier of variables. The purpose of this new syntax is to tell the system that the variable to which it refers is intended to be stateful and, therefore, that special action must be taken. It is only permitted to use this keyword to qualify a local variable of a computation. It is interesting that with the dynamic environment model no such syntax is required. The bank account example of section 4.3.2 will be used to illustrate the special action taken by jRelix when it encounters this keyword. The new version of *make_bank_account* is shown in program 5.

Program 5

```
computation make_bank_account(init_balance, deposit, balance)
  state bal intg;
{  bal <- init_balance;
   deposit(amount)
   {  if ( bal+amount >= 0) then
      {  bal <- bal+amount;
      }
      else
      {  print "error: not enough funds for withdrawal";
      };
   };
   balance(b)
   {  b <- bal;
   };
};
```

This computation may be invoked as it was before.

```
bal <- make_bank_account[100];
```

This time, however, the system will see that the keyword **state** is used instead of **local** as a modifier for the variable *bal*. jRelix will create an extra hidden domain called *#bal*, if one does not already exist, that will be included as one of the domains of *bal*, see figure 4.27 on the following page[4]. When *deposit* is called, the system can find the value of *bal* by checking the entry in the current row and under the column labeled *#bal*.

Several bank accounts may be instantiated at once by means of an ijoin. Here is the required invocation.

```
ba <- [name, accno, deposit, balance] in
           Initial ijoin make_bank_account;
```

[4]The two dots in this figure represent pointers to computations. The way this is implemented is discussed in 4.7.2

```
-------------------------------

deposit      balance      #bal

-------------------------------

..           ..           100

-------------------------------

relation: "bal" has "1" tuple(s)
```

Figure 4.27: Hidden state variables, part 1

The result is shown in figure 4.28 on the next page. In effect, the hidden column labeled *#bal* acts as a storage space for the various values of *bal*. In this respect, it acts as an environment does. So the keyword **state**, in some sense, brings us back to dynamic environments. But the new approach is much more efficient. Each instance of the variable *bal* takes up only the space of an integer. The only overhead is having an extra domain name, but this is independent of the number of bank accounts which are instantiated.

Only one new frame is created when *make_bank_account* is invoked. The declaration of the nested computation *Deposit* will be stored in this new frame exactly once. Therefore, regardless of the number of tuples in the input relation that is joined with *make_bank_account*, the syntax tree corresponding to the Deposit computation is parsed just once. The destination relation will then contain a constant column of computations. That is, all entries under Deposit will point to the same computation object. This approach is highly efficient for database applications which could easily include thousands of bank accounts.

4.7 Storage of Computations

The manner in which top-level computations are stored is different from that in which computations that are nested inside some relation are stored. This is true both in the case of temporary storage in RAM, and of persistent storage on disk. The term 'computations which are nested, or contained, in some relation' is, at the very least, cumbersome, but the temptation to call them nested computations will be resisted.

```
----------------------------------------
name          accno         init_balance
----------------------------------------
Pat           4104          100
Sue           1729          200
----------------------------------------
relation: "Initial" has "2" tuple(s)
```

```
------------------------------------------------------------
name          accno         deposit       balance       #bal
------------------------------------------------------------
Pat           4104          ..            ..            100
Sue           1729          ..            ..            200
------------------------------------------------------------
relation: "ba" has "2" tuple(s)
```

Figure 4.28: Hidden state variables, part 2

The term nested computation is reserved for a computation that is nested inside another computation. (Perhaps, we should not be so fickle, as a computation 'is a' relation!). For brevity, the term 'R-nested computation' will be used when refering to a computation that is contained in some relation.

4.7.1 Top-level computations

Each computation is represented in RAM by a Java Computation object. The Computation class inherits from—or in Java parlance, extends—the Relation class which is used for relations and views [30]. These are inserted into a RelTable object which is essentialy just a hash table containing relations, computations and views, as well as a few handy utility methods. Each frame of the environment contains its own RelTable object so that different relations can belong to different frames.

When the system is shutdown, the Computation objects are not saved to disk in their entirety. Only the syntax tree representing the declaration of the computation is saved by means of the object serialization mechanism which Java provides [23]. The file .expr is used for this purpose. It also stores syntax trees for views and virtual domains as well [70]. When the system is restarted, these syntax trees will be used to create new Computation objects for each top-level computation by calling the constructor of that class.

4.7.2 Computations nested in a relation

Due to the fundamental principle that a computation 'is a' relation, every effort was made to mimic the implementation of nested relations when designing the method for storing R-nested computations. Therefore, whenever a domain of type computation is declared, a relation having the same name but prefixed with a period is automatically created by the system. This new relation is defined on the same attributes as those which make up the type of the newly declared domain, as well as an extra one called '.id'. In figure 4.29 on the following page, the domain *DEPOSIT* is declared. The system will create a new relation called *.DEPOSIT* on the domains *DEP* and *.id*. Whenever a relation is created which has *DEPOSIT* as one of its domains, only surrogates will be stored under that field. These serve as pointers to actual compu-

```
>domain DEP intg;
>domain DEPOSIT comp(DEP);
>pr .DEPOSIT;
+---------------------+-------------+
| .id                 | DEP         |
+---------------------+-------------+
+---------------------+-------------+
relation .DEPOSIT has 0 tuple
>
```

Figure 4.29: Storage of a computation nested in a relation

tations. The surrogates will not be found in the relation *.DEPOSIT*, as would be the case if *DEPOSIT* where a regular nested domain. In fact, the relation *.DEPOSIT* will remain completely empty. It is created solely to mimic the implementation of nested relations so that routines in the interpreter that depend on this implementation will function properly for domains of type computation. The user should simply think of any relation defined on *DEPOSIT* as containing a column of computations. The surrogates that are stored under a domain of type computation are in fact keys for a hash table called CompTable. Since the surrogates are assigned uniquely, there is only a need for one CompTable in the entire system, not one per frame of the environment as might be expected. The actual Computation objects for R-nested computations are stored in the CompTable keyed on the surrogates generated when they were inserted into a relation.

When the system is shut down, all computations in the CompTable are saved to disk. Once again, only their sytax trees are actually serialized to disk. The surrogates for each R-nested computation is also recorded. When the system is restarted, these syntax trees are used to recreate the Computation objects for each R-nested computation. The Computation objects are then reinserted into the CompTable keyed on the same surrogates as in the previous run of the system.

4.8 Recursion

No special effort is required to support recursion. All of the necessary machinery is already provided by the environment model. Consider the *factorial* computation shown in figure 3.12 on page 57. Each time it is called, a new frame is created in which the formal parameters are bound to the values provided by the call. In effect, a stack of new frames is created. Each of these will contain bindings of the formal parameters *M* and *N*. Therefore the different instances of these will not interfere with each other. When the recursion bottoms out, the frames will dissappear (i.e. be marked for garbage collection) in the reverse order of their creation as values are returned from the various incantations of the *factorial* computation.

There is one implementation detail that arises in this context, however, it is not specific to recursive computation. The following statement, taken from the *factorial* computation, illustrates the issue.

```
N <- M * factorial[M-1];
```

Since *N* in an integer, the right hand side of this statement is expected to return an integer. However, the invocation of *factorial* results in a relation. When the system expects an integer, or other primitive type, and obtains a relation instead, an attempt is made to see if an appropriate value can be extracted from the relation. If the relation has exactly one tuple, and is defined on only one domain that is of the type that is expected, then this single value is extracted from the relation. In our example, the result of *factrial[M-1]* is indeed a relation with one tuple that is defined on the single domain *N*, which is of type integer. The value is extracted from the relation and returned to the interpreter as the result of the computation call.

Chapter 5

Conclusion

This chapter begins with a summary of the work that has been accomplished. This is followed by suggestions for possible extensions and enhancements that could be made in the future.

5.1 Summary

This thesis documents the design and implementation of computations, the procedural abstraction mechanism provided by jRelix. The central design principle is that a computation is a special type of relation. A computation is intended to embody a constraint amongst its parameters. It has been explained how this permits a computation to be thought of as a (possibly infinite) relation. Thus, to every computation there corresponds a relation.

One of the qualities of the relational algebra is that its operators may be cascaded. allowing intricate queries to be expressed concisely. The overloading of the relational algebra operators to allow computations as their arguments implies that computations are used syntactically in the same way as relations. Therefore, these queries may consist of computations as well as relations, yielding a high degree of flexibility.

The operators that have been overloaded are the selection and the natural join. These are used to invoke computations. To understand how these work, it is best to consider the relation that corresponds to a computation. The selection and join operators behave with this relation as they would with regular relations.

```
>comp no_constraint(X,Y) is
{ X <- 2 * Y;
} alt
{ Y <- X * X;
};
>
```

Figure 5.1: Computation that does not embody a constraint

Computations also may be used to instantiate objects with state. The model of instantiation is designed to deal with large numbers of objects efficiently. Although behaviour may be encapsulated within the objects, these are not true 'objects' in the object-oriented sense. Many OO properties are not included in this implementation. The following section discusses how some of these may be added in the future.

5.2 Future Work

5.2.1 A note on the commutivity of the natural join

It has been claimed that the natural join operator remains commutative after having been generalized to work with computations. As has been discussed, there is a relation which corresponds to each computation that embodies a constraint. The join of a relation with a computation is like the join of a relation with the relation corresponding to a computation. Thus, the commutivity, as well as the associativity, of the generalized join follows from that of the regular join.

However, if the user creates a computation that does not embody a constraint, then it may not correspond to any relation. In this case, the generalized join operator will most likely not be commutative. An example of such a computation is provided in figure 5.1. Note that the two 'alt' blocks do not enforce the same relationship amongst the parameters of the computations. Therefore, this computation does not embody a constraint.

5.2.2 Natural join of two computations

The join of two relations and the join of a relation with a computation have been discussed. It is natural next to inquire what the join of two computations should yield. The meaning is in fact forced upon us if we wish to preserve the associativity of the join operators. Consider the expression

$$Comp1 \bowtie Comp2 \bowtie Rel1 \tag{5.1}$$

This can be evaluated in two different ways depending on whether the ijoin operator is left or right associative[1]. Regardless, we would like the results to be the same in each case. Evaluating the expression from right to left presents no trouble as the result of joining *Comp2* with *Rel1* is a relation, which can then be joined with *Comp1*. Thus the join of two computations must behave like function composition. The join of *Comp1* with *Comp2* results in a new computation which, when it is applied to a relation, *Rel1*, has the same effect as joining *Comp1* with *Rel1* and then joining the result of this with *Comp2*.

This suggests an implementation strategy for the join of two computations. A computation can actually be a doubly linked list of Computation objects. Computations such as *Comp1* would be a linked list with the Computation object of *Comp1* as its single item. Similarly, *Comp2* would be a linked list containing only the Computation object of *Comp2*. When these are joined together, the resulting object would be a computation that we will call *Comp3*. *Comp3* would actually be a doubly linked list with two entries, one of these being the Computation object for *Comp1* and the other that for *Comp2*. to apply *Comp3*, all that must be done is to apply the computations stored in the linked list in turn. If the relation that is joined with *Comp3* is to the left of the join operator, then we begin applying the computations starting at the head of the linked list. If the relation is to the right of the join operator, then we start with the computations at the tail of the linked list, and work towards the head.

This implementation, however, has the unfortunate side effect of breaking the commutivity of the join operator. Suppose the computation *Comp2* is defined on the parameters X and Y and encapsulates the constraint $Y = X + 2$, and that *Comp1*

[1]The join operators are in fact right associative.

is defined on Y and Z and represents the constraint $Z = Y^2$. Then, if the relation *Rel1* is defined on X, the result of equation 5.1 is essentially to calculate the function $Z = (X+2)^2$ However, the expression

$$Comp2 \bowtie Comp1 \bowtie Rel1 \qquad (5.2)$$

does not even make much sense since *Comp1* and *Rel1* have no domains in common. A join of two relations having no domains in common yields a cartesian product. If we imagine that it is possible to take the cartesian product of the infinite relation corresponding to *Comp1* with *Rel1*, then the resulting (infinite) relation, let us call it *Temp*, could indeed be joined with *Comp2*. *Temp* would be defined on the domains X, Y, and Z. Joining *Temp* with *Comp2* would amount to picking out those tuples of *Temp* satisfying $Y = X + 2$. The finiteness of the relation *Rel1* together with the constraints guarantees that the final result will be finite[2].

The difficulty with the preceding is in how to implement the cartesian product of a computation with a relation, as the result is infinite. Overcoming this problem would restore the the commutivity of the natural join operator.

5.2.3 Overloading other relational algebra operators

It would be nice if all of the relational algebra operators were overloaded to function with computations. So far, this has only been done for the selection and natural join operators. Unfortunately, it is not immediately clear what the appropriate generalizations of the other operators (projection, μ-join other than the natural, σ-joins, and update) should be. Examples applications would be necessary to motivate and justify any extensions.

5.2.4 Events and triggers

In Relix, the predecessor to jRelix, procedures were used to implement event handlers [26]. Due to time limitations, including this facility within computations was not an

[2]Indeed, the constraint $Y = X + 2$ implies that for each value of X there corresponds a finite number of values for Y. The constraint $Z = Y^2$ in turn implies that there can only be a finite number of values for Z. In this example, the 'finite number' just mentioned is in fact one. In general, however, because of the **also** statement this could be any finite number.

Domains of S	Block type
A,X	3
A	1
X	2
None	0

Table 5.1: Ambiguous block types

objective of the current project. This feature could be added at a later date.

5.2.5 Block algorithm: select clause

As explained in section 4.5, the select clause is ignored by the block algorithm. The reason is that the algorithm is used at declaration time, but it can not be known until run-time what identifiers in the select clause belong to the source relation. Consider the following example.

```
comp C(A,X) is
{ R <- where A=X in S;
};
```

It is impossible to tell, at declaration time, whether A, X or both are domains of the relation S. The problem is that the block type depends on this information. Table 5.1 gives all the possible block types for the example.

Since the implementation would be more complicated and also because it would be less efficient to try to deduce this information at run-time, the decision was made to ignore the selection clause in the block algorithm. Further experience with the system may lead to a better solution. Note that the ambiguity concerning which identifiers belong to the source relation results from the practice of labeling the columns of a relation. No such ambiguity arises with a positional notation, such as the array syntax.

For the time being, the user can force a particular meaning by using local variables. For example, if the intent is that X is a member of S and A is an input parameter. then the following code could be used.

```
comp C(A,X) is
 local l intg;
{ l <- A;
  R <- where l=X in S;
};
```

The first statement makes A an input parameter. The relation R will consist of those tuples of S that have a value of l in the column headed by X. Since the value of l is set to A, the desired result is achieved.

The same probem arises with the use of domain algebra in a computation.

```
comp C(A,R) is
{ let X be A+B;
  R <- [X] in S;
};
```

It can not be deduced until run-time whether or not A is a domain of S. Thus, the block type is ambiguous. Once again, a local variable may be used to force A to be an input parameter.

5.2.6 Storing a view in a relation

jRelix supports nested relations—relations that contain relations. It has also been seen that relations may contain a column of computations. This is consistent because computations are a special from of relation. Views are also a special form of relation. Therefore it is natural to inquire about the possibility of placing views in a relation. Consider the following code fragment.

```
comp viewComp(S,T,R) is
{ R is S ijoin T;
};
Rel <- viewComp ijoin InputRel;
```

The invocation of the computation *viewComp* causes a view to be placed in the relation *Rel*.

The implementation of this is essentially the same as for columns of computations. A ViewTable class, analogous to the CompTable class described in section 4.7.2, stores the views in a hash table keyed on surrogate values. A column of views actually consists of a column of surrogates which which are pointers to syntax trees stored in a ViewTable object.

Unfortunately, storing views in a relation violates one of the principles used in the design of nested relations in jRelix. As explained in section 2.5, there is a restriction that applies to the structure of a nested relation—all relations stored in a column must have the same schema. This schema is well known because it is the type of the domain, and the domain must be declared before any relation can be defined on it. In contrast, a view does not have a fixed schema. The domains of R, in the example, depend on the domains of S and T. Since the values of S and T may change over time, the schema of R can not be known until it is time to evaluate the view. From the perspective of the implementation, however, this does not pose a problem. The current proposal is experimental. Time spent with the system will tell if this is a useful feature.

5.2.7 Object-oriented features

The inclusion of object-oriented features in Relix has been investigated in [49]. However, the approach taken there, being based on relations, is somewhat incompatible with the instantiation mechanism developed in this thesis, which is based on computations. We now explore ways of including additional OO features in jRelix. We base the discussion on the points outlined in the *Object-Oriented Database System Manifesto*. These were listed in table 1.2 on page 12.

Classes

Classes in jRelix correspond to stateful computations, which are computations that have at least one state variable. The bank account computation of figure 3.9 on page 55 illustrates this. It has a stateful variable called *bal*.

Methods

Methods in jRelix are computations that are nested inside a stateful computation. The bank account example illustrates this as well. That class has a *DEPOSIT* method. for making either deposits or withdrawals, and a *BALANCE* method, for printing out the current balance in the account.

Instantiation

Objects of a class may be instantiated by calling a stateful computation. There is no need for new syntax. In particular, jRelix has no need for a **new** operator to instantiate objects, as with Relix and many OO languages.

Visibility

In [9], it is discussed how first-class procedures can be used to distinguish private and public data. The approach can be adopted by jRelix. Public variables and methods are those that appear in the parameter list of the stateful computation. We say that these have been exported through the parameter list by the stateful computation. Private variables and methods are those that do not appear in the parameter list.

Complex object

As a stateful computation can contain any number of stateful variables, support for complex objects is provided.

Object identity

Relational systems are inherently value-oriented. This means that a relation will never contain two identical tuples. This is a consequence of the definition of a relation as a set, instead of a bag[3]. Thus a relation called *people* with attributes *NAME* and *CITY* could only contain one tuple with *NAME* set to "John Smith" and *CITY* set to "Montreal" even though there very well could be, and probably are, at least two John Smith's in Montreal. It is possible to use an extra surrogate domain, perhaps

[3]A bag is similar to a set, except that it may contain duplicates.

someMscns	(init_name	init_city	init_inst)
	J. Medeski	New York	Wurlitzer
	B. Martin	New York	Percussion
	C. Wood	New York	Bass

Figure 5.2: Some musicians

called *.oid*, that could serve as a unique object identifier. This could be left to the user's discretion or it could be maintained by the system. See [49] for a discussion of the issues that this raises.

Encapsulation

As the methods of a class are contained within the class declaration (i.e. the stateful computation), encapsulation is provided by jRelix.

Inheritance

A model of inheritance could be built on top of the implementation of the join of two computations suggested above. To motivate this, consider a *musician* class that inherits from a *person* class. The class definitions are as follows:

```
computation person(init_name,init_city) is
state name string;
state city string;
{ name <- init_name;
  city <- init_city;
  ...
};
```

```
computation mscn(init_inst) is
state instrument string;
{ instrument <- init_inst
  ...
};
```

The *musician* class is a *mscn* class that inherits from *person* by means of a natural join.

```
musician <- mscn ijoin person;
```

So, starting with the relation shown in figure 5.2, we may instantiate *musician* objects in the usual way.

```
-------------------------------------
#name         #city        #instrument
-------------------------------------
J. Medeski    New York      Wurlitzer
B. Martin     New York      Percussion
C. Wood       New York      Bass
-------------------------------------
relation MMW has 3 tuples
```

Figure 5.3: Musician objects

```
MMW <- musician ijoin someMscns;
```

The result is shown in figure 5.3. There are three private stateful variables.

In order to make the use of inheritance less esoteric, the keyword **isa**, first introduced in [49], may be included in jRelix. The previous example becomes

```
mscn isa person;
MMW <- mscn ijoin someMscns;
```

The first statement indicates that the *mscn* is a subclass of *person*. The next statement performs the instantiation.

Appendix A

Backus-Naur Form for jRelix Commands

This appendix describes jRelix grammar/syntax in the Backus-Naur Form (BNF) format. The convention of this BNF definition is explained 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 specification. Therefore the grammar looks different from that of the former Relix which is intended for the bottom-up parser generator Yacc.

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.

```
Start := Command ";" | Statement ";" | ";" | <EOF>

Command := "help" (<IDENTIFIER>)?
    | "quit" | "input" FilePath | "debug" | "batch" | "expert"
    | "time" | "deld" IDList | "delr" IDList | "pr" Expression
    | "sd" (<IDENTIFIER>)? | "sr" (<IDENTIFIER>)? | "srd"
    | "ssd" | "ssr" | "print" <STRING_LITERAL>

Statement := SequentialStatement

SequentialStatement := ParallelStatement ("--" ParallelStatement)*

ParallelStatement := ChoiceStatement ("||" ChoiceStatement)*

ChoiceStatement := PrimaryStatement ("??" PrimaryStatement)*

PrimaryStatement := Declaration | Assignment | Update
    | ComputationCall | Conditional | ForLoop | WhileLoop
    | Exit | DeadLock | Exec | StatementBlock

StatementBlock := "{" Statement (";" Statement)* (";")? "}"

Conditional := "if" Expression "then" Statement ("else" Statement)?

ForLoop := ("for" Identifier)? ("from" Expression)?
           ("to" Expression)? ("by" Expression)?
           ("do" | "loop") Statement

WhileLoop := "while" Expression ("do" | "loop") 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 ("initial" Expression)? "be" Expression
    | ("computation" | "comp") Identifier
```

```
      "(" (ParameterList)? ")" "is" ComputationBody

Initialization := "<-" ("{" ConstantTupleList "}" | Identifier)

ConstantTupleList := ConstantTuple ("," ConstantTuple)*

ConstantTuple := "(" Constant ("," Constant)* ")"

Constant := Literal | "{" ConstantTupleList "}"

Identifier := <IDENTIFIER>

FilePath := <STRING_LITERAL>

Assignment := Identifier
    ( ("<-" | "<+") Expression
      | "[" IDList ("<-" | "<+") ExpressionList "]" Expression
    )

Update := "update" Identifier
    ( ("add" | "delete") Expression
      | "change" (StatementList)? (UsingClause)?
      | "[" IDList ("add" | "delete") ExpressionList "]" Expression
    )

StatementList := Statement ("," Statement)*

UsingClause := "using"
    ( 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 (("min" | "max") Summation)*

Summation := JoinExpression (("+" | "-") JoinExpression)*

JoinExpression := Projection
    ( JoinOperator Projection
      | "[" ExpressionList ":" JoinOperator (":")?
      ExpressionList "]" Projection
    )*

Projection := Projector (("in" | "from") Projection | 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 (("*" | "/" | "mod") Factor)*

Factor := ("+" | "-" | "not" | "!") Factor | Power

Power := Primary ("**" Power)*

Primary := Literal | QuantifierOperator | ArrayElement
    | PositionalRename | Identifier | Cast | "(" Expression ")"
    | Pick | Eval | Function | IfThenElseExpression | VerticalExpression

ArrayElement := Identifier "[" ArrayIndexList "]"
```

```
ArrayIndexList := (Expression)? ("," (Expression)?)*

PositionalRename := Identifier "(" (IDList)? ")"

Cast := "(" Type ")" Primary

Pick := "pick" Selection

Eval := "eval" Expression

Function := FunctionOperator "(" Expression ")"

Literal := "null" | "dc" | "dk" | "true" | "false"
    | ("+" | "-")? (<INTEGER_LITERAL> | <FLOAT_LITERAL>)
    | <STRING_LITERAL>

IfThenElseExpression := "if" Expression "then" Expression
                        "else" Expression

VerticalExpression := "red" AssoCommuOperator "of" Expression
    | "equiv" AssoCommuOperator "of" Expression
      "by" ExpressionList
    | "fun" OrderedOperator "of" Expression
      "order" ExpressionList
    | "par" OrderedOperator "of" Expression
      ( "order" ExpressionList "by" ExpressionList
        | "by" ExpressionList "order" ExpressionList
      )

Type := ("boolean" | "bool") | "short"
    | ("integer" | "intg") | "long"
    | ("float" | "real") | "double"
    | ("string" | "strg") | "text"
    | ("statement" | "stmt")
    | ("expression" | "expr")
    | ("computation" | "comp") "(" IDList ")"
    | "(" IDList ")"

AssoCommuOperator := ("or" | "|")
    | ("and" | "&") | "min" | "max" | "+" | "*"
    | ("ijoin" | "natjoin") | "ujoin" | "sjoin"

OrderedOperator := AssoCommuOperator
    | "cat" | "-" | "/" | "mod" | "**" | "pred" | "succ"
```

```
ComparativeOperator := "substr" | "=" | "!=" | ">" | "<" | ">=" | "<="

JoinOperator := "nop" | MuJoin
    | (("not" | "!"))? SigmaJoin

MuJoin := ("ijoin" | "natjoin")
    | "ujoin" | "sjoin" | "ljoin" | "rjoin"
    | ("dljoin" | "djoin") | "drjoin"

SigmaJoin := ("icomp" | "natcomp") | "eqjoin"
    | ("gejoin" | "sup" | "div") | "ltjoin"
    | ("lejoin" | "sub") | ("iejoin" | "sep")

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 := ComputationDeclaration
                   ComputationBlock ("alt" ComputationBlock)*

ComputationBlock := "{" ComputationStatements "}"

ComputationDeclaration := ( LocalVariableDeclaration
                            | StateVariableDeclaration
                          )*

LocalVariableDeclaration := "local" IDList Type ";"

StateVariableDeclaration := "state" IDList Type ";"

ComputationStatements := Statement (";" Statement |
                         "also" Statement)* (";")?

ComputationCall := Identifier "(" (CallParameterList)? ")"

CallParameterList := CallParameter ("," CallParameter)*

CallParameter := ("in" | "out") <IDENTIFIER>
```

Bibliography

[1] Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Dan Friedman, Robert Halstead, Chris Hanson, Chris Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Bill Rozas, Gerald Jay Sussman, and Mitchell Wand. The revised revised report on scheme or the uncommon lisp. Technical Memo AIM-848, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1985.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[3] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):32–41, June 1993.

[4] E. Allman, G. Held, and M. Stonebraker. Embedding a data manipulation language in a general purpose programming language. In *Proceedings of the Conference on Data Abstraction, Definition, and Structure*, pages 25–35, Salt Lake City, UT, March 1976. ACM-SIGPLAN-SIGMOD, ACM.

[5] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, K. P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[6] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. PS-algol: A language for persistent programming. In *10th Australian National Computer Conference*, pages 70–79, Melbourne, Australia, 1983. This document is available at http://www-ppg.dcs.st-and.ac.uk/Publications/.

[7] M. P. Atkinson, W. P. Cockshott, P. Bailey, K. J. Chisholm, and R. Morrison. PS-algol reference manual. Technical Report PPR-4-83, Departments of Computer Science, Universities of Edinburgh and St. Andrews, January 1984.

[8] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[9] M. P. Atkinson and R. Morrison. Persistent first class procedures are enough. In M. Joseph, R Shyamasundar, and J Hartmanis, editors, *Lecture Notes in Computer Science 181*, pages 223–240. Springer-Verlag, 1984.

[10] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The object-oriented database system manifesto. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, page 395, Atlantic City, NJ, 23–25 May 1990.

[11] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[12] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System.* Morgan Kaufman, August 1991.

[13] François Bancilhon. A classification of object oriented database systems. In *Database Programming languages: bulk types and persistent data*, pages 3–6, San Mateo, CA, August 1991. Proceedings of the Third International Workshop, Morgan-Kaufmann Publishers Inc.

[14] J. Bocca. EDUCE: A marriage of convenience: Prolog and a relational DBMS. In *Proceedings of the International Symposium on Logic Programming*, pages 36–45. IEEE Computer Society,, The Computer Society Press, September 1986.

[15] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):225–236, June 1990.

[16] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient database access from Prolog. *tose*, 15(2):153–164, February 1989.

[17] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[18] C. L. Chang and A. Walker. PROSQL: A Prolog programming interface with SQL/DS. In L. Kerschberg, editor, *Expert Database Sys.*, page 233. Benjamin/Cummings, Menlo Park, CA, 1986.

[19] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. In *IEEE Transactions on Knowledge and Data Engineering*, volume 2, March 1990.

[20] Hong-Tai Chou, David J. DeWitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the Wisconsin Storage System. *Software— Practice and Experience*, 15(10):943–962, October 1985.

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

[22] A. J. Cole and R. Morrison. *An Introduction to Programming with S-algol.* Cambridge University Press, Cambridge, England, 1982.

[23] Gary Cornell and Cay S. Horstmann. *Core Java.* SunSoft Press, second edition, 1997.

[24] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.

[25] A. Dearle, R. C. H. Connor, A. L. Brown, and R. Morrison. Napier88 - A database programming language? In R. Hull, R Morrison, and D Stemple, editors, *2nd International Workshop on Database Programming Languages*, pages 179–195, Salishan Lodge, Oregon, 1989. Morgan Kaufmann. This document is available at http://www-ppg.dcs.st-and.ac.uk/Publications/.

[26] A.Z. El-Kays. Implementing event handlers in a database programming language. Master's thesis, McGill University, Montreal, Canada, 1996.

[27] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–49. October 1991.

[28] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.

[29] D. H. Fishman et al. Overview of the Iris DBMS. Technical report, HP Labs, DB Techno. Department of, June 1988. Also published in/as: In 'Object-Oriented Concepts, Languages, and Applications', Edited by W.Kim, F.H.Lochovsky, pages 219–150, AW, 1989.

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

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

[32] Y. E. Ioannidis and M. M. Tsangaris. The design, implementation, and performance evaluation of BERMUDA. *IEEE Transactions on Knowledge and Data Eng.*, 6(1):38, February 1994.

[33] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report (third edition).* Springer-Verlag, New York, N.Y., 1985. Revised to the ISO Standard by Andrew B. Mickel and James F. Miner.

[34] S.C. Johnson. Yacc: Yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[35] W. Kim. Features of the ORION object-oriented programming database system. In *Object oriented Concepts, Applications, and Database*, pages 251–282. Addison Wesley, March 1989.

[36] Won Kim. *Introduction to Object-Oriented Databases.* MIT Press, Cambridge, 1990.

[37] Won Kim, Jorge F. Garza, Nat Ballou, and Darrell Woelk. Architecture of the ORION next-generation database system. In *IEEE Transactions on Knowledge and Data Engineering*, volume 2, pages 109–124. 1990.

[38] Normand Laliberté. Design and implementation of a primary memory version of ALDAT including recursive relations. Master's thesis, McGill University, Montreal, Canada, 1986.

[39] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Object-Store database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[40] M.E. Lesk. Lex: a lexical generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[41] J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1984.

[42] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to starburst: Objects, types, functions, and rules. *Communications of the ACM, Special Section on Next-Generation Database Systems*, 34(10):94, October 1991.

[43] Rebecca Lui. Implementation of procedures in a database programming language. Master's thesis, McGill University, Montreal, Canada, 1996.

[44] F. Matthes, A. Rudloff, J. W. Schmidt, and K. Subieta. The database programming language DBPL - user and system manual. FIDE Technical Report Series FIDE/92/47, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992. This document is available at http://www.sts.tu-harburg.de/papers/1992/MRSS92.

[45] F. Matthes and J. W. Schmidt. The type system of DBPL. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, pages 255–260, June 1989. This document is available at http://www.sts.tu-harburg.de/papers/1989/MaSc89.

[46] F. Matthes and J. W. Schmidt. Bulk types: Built-in or add-on? In M. P. Atkinson, editor, *Fully Integrated Data Environments.* Springer-Verlag, 1995. This document is available at http://www.sts.tu-harburg.de/papers/1995/MaSc95b.

[47] T.H. Merrett. *Relational Information Systems.* Reston Publishing Company, Reston, Virginia, 1984.

[48] T.H. Merrett. Computations: Constraint programming with the relational algebra. *International Symposium on Next Generation Database Systems and their Applications*, September 1993.

[49] Ellen Mnushkin. Inheritance in a relational object-oriented system. Master's thesis, McGill University, Montreal, Canada, 1992.

[50] Katherine Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the NAIL! system. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 554–568, London, 1986. Springer-Verlag.

[51] R. Morrison. S-algol language reference manual. Technical Report CS/79/1, University of St Andrews, 1979.

[52] R. Morrison. PS-algol reference manual. Technical Report 12, University of St. Andrews, St. Andrews, Scotland, February 1988.

[53] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, G. N. C. Kirby, and D. S. Munro. *Napier88 Reference Manual (Release 2.2.1)*. University of St Andrews, 1996. This document is available at http://www-ppg.dcs.st-and.ac.uk/Publications/.

[54] R. Morrison, R. C. H. Connor, G. N. C. Kirby, and D. S. Munro. Can Java persist? In *1st International Workshop on Persistence for Java*, Glasgow, 1996. This document is available at http://www-ppg.dcs.st-and.ac.uk/Publications/.

[55] Allen Otis, Paul Butterworth, and Jacob Stein. The GemStone object database management systems. *Communications of the ACM*, 34(10):64–77, October 1991.

[56] Lawrence A. Rowe and Kurt A. Shoens. Data abstractions, views and updates in RIGEL. In *Proceedings of the ACM-SIGMOD Conference on Management of Data, Boston, Mass.*, pages 71–81, May 1979.

[57] Sriram Sankar, Rob Duncan, and Sreenivasa Viswanadha. Java compiler compiler (javacc)—the java parser generator, 1996. JavaCC web site at: http://www.suntest.com/JavaCC/. The web site contains documentation, FAQs, newsgroups, and software for JavaCC and JJTree.

[58] J. W. Schmidt and F. Matthes. DBPL language and system manual. Esprit Project 892 MAP 2.3, Fachbereich Informatik, Universität Hamburg, Germany, April 1990. This document is available at http://www.sts.tu-harburg.de/papers/1990/ScMa90b.

[59] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[60] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and O_2. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 21(1):93–105, March 1992.

[61] Guy L. Steele, Jr. *Common LISP: The Langugage.* Digital Press, Pennsauken, New Jersey, 1984.

[62] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[63] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In C. Zaniolo, editor, *sigmod*, pages 340–355, Washington, DC, May 1986. acm.

[64] M. R. Stonebraker, E. Wong, P. Kreps, and G. D. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[65] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[66] Nattavut Sutyanyong. Implementation of domain algebra in jRelix. Master's thesis, McGill University, Montreal, Canada, 1998.

[67] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Bystems*, volume 1. Computer Science Press, Rockville, Maryland, 1989.

[68] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Bystems*, volume 2. Computer Science Press, Rockville, Maryland, 1989.

[69] Niklaus Wirth. *Programming in Modula-2 (3rd corrected edition).* Springer-Verlag, New York, N.Y., 1985.

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