Procedural abstraction in a relational database programming language

Nattavut Sutyanyong

School of Computer Science
McGill University, Montreal

June 1994

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

Name SUTYANYONG, NATTAVUT

Dissertation Abstracts International is arranged by broad, general subject categories Please select the one subject which most nearly describes the content of your dissertation Enter the corresponding four-digit code in the spaces provided

Computer Science	0 9 8 4
SUBJECT TERM	SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

Abstract

This thesis introduces the notion of procedural abstraction in a relational database system. Procedures are treated as special forms of relations, and called *computations.*

Like relations, a computation is defined over a set of attributes. Subset of attributes can be defined as input attributes and the remaining attributes are output. Beyond the notion of procedures that a procedure can have only one set of input and output parameters, computations are symmetric: a computation may have a number of different subsets of input attributes.

Computations can be recursive and called by other computations.

States are introduced so that computations can remember values from previous evaluation and use them in next invocations. Stateful computations may be instantiated to have new sets of states.

This thesis contains the design and implementation of a parser for compiling computations as well as operations to evaluate them. All operations are coincident with relational algebra, a set of operations for manipulating relations.

Résumé

Cette thèse introduit la notion d'abstraction procédurale dans un système de base de données relationelle. Les procédures sont traitées comme des formes spéciales de relations et elles sont nommées *computations.*

De même que les relations, une computation est définie sur un ensemble d'attributs. Un sous-ensemble d'attributs peut être défini comme des attributs d'entrée et les attributs qui restent répresentent la sortie. Au delà de la notion de procédures, une procédure ne pouvant avoir qu'un ensemble de paramètres d'entrée et de sortie, les computations sont symétriques: une computation peut avoir un nombre différents de sous-ensembles d'attributs d'entrée.

Les computations peuvent être récursives et appelées par d'autres computations.

Les états sont introduits pour que les computations puissent se rappeler les valeurs des évaluations précédentes et les utiliser dans des invocations ultérieures. Les computations étatiques peuvent être instantisées pour avoir des nouveaux ensembles d'états.

Le contenu de cette thèse inclus la conception et l'implantation d'un analyseur lexical pour la compilation des computations ainsi que des opérations pour leur évaluation. Toutes les opérations coincident avec l'algèbre relationel, un ensemble d'opérations pour la manipulation des relations.

Acknowledgements

I would like to thank my supervisor, Professor T.H. Merrett, for his invaluable advice throughout the period of doing this thesis.

I would also like to thank the system coordinator of our ALDAT lab, Heping Shang, who provided much assistance on the usage of facilities in the lab; Luminita C. Stancu and Sandro Mazzucato who translated the abstract into the résumé in French; Jared A. Chiama who proofread the whole thesis; S. Alan Ezust who provided me with the LaTeX script for typesetting the thesis.

A special thanks goes to the Canadian International Development Agency (CIDA). Without their financial support throughout my study program, I would not have been able to complete my research.

Finally, I would like to dedicate this thesis to my parents, who have been awaiting my return home for almost two years. I will be back soon.

Contents

Chapter 1

Introduction

This thesis documents the implementation of *computation*, a notion of procedural abstraction, in a relational database system [37]. The work was done by extending *Relix*, a relational database system developed at McGill.

Computations provide a mechanism of storing procedures or pieces of programs in a database. One can then use computations to evaluate data in the database. Additionally, computations are treated as special forms of relations. Operations on computations are designed to coincide with the operations on relations in Relix.

1.1 An overview of the relational model

After E. F. Codd first introduced the relational database model in 1970 [10], much research in the area of relational database systems has thrived broadly and rapidly. Research has been done in the context of:

- database design [9, 19, 20];
- implementation techniques [46, 44, 36];
- database theory [31];
- query languages [48, 26];
- concurrency control [18, 27];

- distributed databases [8];
- database machines [23].

Before we go into details, we shall clarify some terminology here.

The word *domain* means *set of values*. For example, the domain of student identifiers is the set of all possible student identifiers.

Suppose D_1, D_2, ..., D_n are domains. R is a *relation* on these n domains if it is a set of n-component data, the first component of which is a value on D_1, the second component from D_2, and so on. In other words, the relation R is a subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$. We said that the *degree* of R is n.

A name is given to each of the n domains of a relation to release users from remembering the domain ordering of the relation. These names are called the *attributes* of the relation. One can refer to any domain of a relation by its attribute.

Each element of the set of n-component data is said to be a *tuple* of the relation. A tuple can be thought of as the mapping of a set of attributes to a set of values.

A relation must retain the following properties:

- Each attribute is unique among all attributes of the relation.
- All tuples are distinct from one another.
- The ordering of tuples is unimportant.

For example, suppose we want to keep the information of students who are registered in a course. Student identifiers and names, which are divided into two fields lastname and firstname, are recorded. A relation, *Student*, of the student records may be represented in table form as in Table 1.1.

We may perceive a relation as a table. An attribute is equivalent to the title of a column. A tuple is equivalent to a row or record. Here, *stu_id, lastname*, and *firstname* are attributes of the relation *Student*.

(stu_id:"9300435", lastname:"Smith", firstname:"Roy")

is a tuple of the relation *Student*.

stu_id	lastname	firstname
9300435	Smith	Roy
9213651	White	Adam
9204560	Ford	Watson
9102214	Lee	Sandra
9380009	Warrant	Roger
9011137	Pinn	Michael

Figure 1.1: An example of a relation

In the relational model, relations are the only data structures that users perceive. The work of organising data in storage is taken care of by database management systems (DBMS).

1.1.1 Operations on relations

In order to manipulate data, we need a set of operations. In the relational model, this is called *relational algebra*. Since Codd's original publication in [11], a number of variations of relational algebra have been proposed; for example, in [35].

Codd originally defined two groups of four operations each. The first four operations are traditional set operations—*union, intersection, difference*, and *Cartesian product*. The second ones are special relational operations—*select, project, join*, and *divide*. All operations take relations as operands and return a relation as result.

Union takes two relations which are of the same degree (say n) and for every i (i=1,2,...,n) the i^{th} domains of the two relations are the same; and returns a relation that is created by copying tuples from its operands, and eliminating duplicate tuples.

Intersection takes two compatible relations, as in the union operator; and returns a relation containing only those tuples that appear in both operands.

Difference takes two compatible relations, as in the union and intersection operators; and returns a relation containing those tuples that appear in the first operand but not in the second operand.

Cartesian product takes two relations; and returns the relation that contains all tuples generated by combining every pair of tuples, one from each of the two operands.

Selection takes a relation and a condition; and returns a subset of the relation contains those tuples of the operand that satisfy the condition.

Projection takes a relation and a list of attribute(s); and returns a relation of the specified attribute(s) that contains all tuples of the operand with duplicate tuples removed.

If we think of a relation as a table, a selection is an extraction of the relation in the horizontal way and projection is an extraction in the vertical way (with duplicate rows removed).

Join takes two relations and a condition; and returns a relation consists of all possible concatenated pairs of tuples, one from each of the two operands, such that the condition is true. Join is equivalent to the selection on the condition of the Cartesian product of the two operands.

Division takes two relations. One is a dividend of m+n attributes. The other is a divisor of n attributes, which are also defined on the dividend. The operator compares sets of the n attributes of the dividend on the same value of the m attributes to the set of all tuples of the divisor; and returns a relation of tuples of the m attributes whose set of the n attributes cover all tuples of the divisor.

Details on relational algebra can be found in [11, 36, 15, 16, 14].

1.2 Research trends in database models

The first paper on the relational model was introduced by Codd in 1970 [10]. In 1979, Codd published the extended relational model RM/T [12]. He wrote a book describing what he called Version 2 of the model in 1990 [14]. The features in the relational model have been increased from "nine structural features, three integrity features, and eighteen manipulative features" in the RM/T model (referred in [17]) to "18 classes of features" in the Version 2 [14].

The use of database systems has grown rapidly. Nowadays people require database systems to store and manipulate not only tableau format of data; but also parts of text documents, programs, maps, or diagrams [7]. Applications of databases vary from business applications to science/engineering applications such as computer-aided design (CAD), computer-aided manufacturing (CAM); office information systems (OIS); geographical information systems (GIS); and hypertext/hypermedia. One of the needs of the latter types of applications is the ability to store and manipulate complex data structures. The relational model, unfortunately, because of its simple data structure, has been claimed to not support complex objects. Furthermore, in most commercial relational database systems, data is retrieved or updated through database languages, such as SQL. [1] Database languages, particularly SQL, lacks the *computational completeness* [4]. That means the languages cannot perform all computations, such as recursion and looping that can be done in ordinary programming languages. To counter this problem, the database languages are *embedded* in *host* languages, such as Pascal, PL/1, C, etc. When a program reaches a database command, it passes the command to the database system for evaluation and the result is returned back to variables in the main program. Zdonik and Maier criticised this approach led to "the *impedance mismatch* between the data manipulation language (DML) of the database and the general-purpose language in which the rest of the application is written" [1].

In the following two sections, we look at two approaches of database models that have been proposed to solve the above problem. We then discuss our approach in the last section.

1.2.1 Object-Oriented models

Recently, the concept of object-orientation has been flourishing in many fields of computer science. Among them are object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programmming (OOP), and object-oriented database management system (ODBMS). ODBMSs are, more or less, influenced by the concept of OOP. We briefly discuss the concept of OOP here.

A programming language is considered an OOP language if it embodies the following concepts:

[1]SQL, stands for Structured Query Language, originally pronounced se-quel, was first defined by Chamberlin and others at the IBM Research Laboratory in San Jose, California (cited from [15] p.95.)

- Classes and objects

  A class contains data and operations on data. Operations are called *methods* in OOP terminology. One can use a class by declaring a name belonging to that class. This process is called *instantiation*. The name becomes an *object* of the class. Theoretically, one can access the data of an object only by calling methods provided by the class of the object.

- Class hierarchies

  A class can *derive* data and operations from other classes. For example, persons and students have the following common properties: name, age, and sex. However, students may have some additional specific properties, such as student id and major. In OOP, classes such as persons are called *parent classes*; and students are called *subclasses*. Subclasses may derive not only common data elements but also operations on data from parent classes. This mechanism is called *inheritance*. A class can be a subclass of its parent class and, at the same time, a parent of other classes. The relationships then form as hierarchical trees of classes. The benefit is we can reuse data and operations of parent classes without rewriting them in subclasses. Any changes on operations are made in the parent classes only, instead of iteratively making changes to the same operations in several subclasses.

- Overloading and late binding

  In fact, the concept of overloading is not new and has been used in ordinary programming languages for a long time. The example is the operator plus (+). When we write two statements, say, 3 + 4 and 3.5 + 4.2, the language distinguishes the plus operators of these two statements. The first plus operator is an "integer" plus but the latter is a "real" plus that are totally different from the implementation point of view. However, users do not have to realise that. In OOP, the concept is applied more generally. We can have two methods with the same name. For example, we can have a print() method in class *integer* and the same name in class *real*; compared with a print_int() for printing integers and another print_real() for reals in ordinary programming languages.

Generally, ODBMSs are the database systems that allow data to store beyond tableau format of the relational model. It can deal with complex data structures as in programming languages. Another possible way of thinking of ODBMSs is as an OOP with persistent data, in the sense that data in the programs lives beyond the life of programs. The ability to manipulate data and perform computations within one single system is the strong point that is claimed to solve the problem of the impedance mismatch between data manipulation languages (e.g., SQL) in the relational model and ordinary programming languages.

Although ODBMSs have some advantages over relational database management systems (RDBMSs), no standard specification in ODBMSs has been defined, as Codd's original paper [10] contributed to RDBMSs. Varieties of research prototypes and commercial products are done or underway. Among these are Encore [49], EXODUS [6], Gemstone [5], Iris [21], O_2 [4], ObjectStore [29], Versant [47].

1.2.2 Extended relational models

Another direction of the research in database models is to extend the relational database systems to support complex objects. We discuss an example of this approach.

POSTGRES is an extended relational database system developed at University of California, Berkeley by Stonebraker and his team. The system was extended from an original relational database system called INGRES. Some of the goals of POSTGRES were [42]:

- Support complex objects
- Allow new data types, new operators, and new access methods to be included in the DBMS
- Support triggers and rules
- Make as few changes to the relational model as possible

The query language used in POSTGRES is POSTQUEL. It is a modification of

QUEL,[2] the original query language used in INGRES.

POSTGRES allows users to use C functions in query commands in order to cope with complex queries. For example, a query "Get the names of students whose final grade is A" may be written in POSTQUEL [41] as follows:

```
RETRIEVE (Student.Name)
WHERE grade(Student.midterm, Student.final) = 'A'
```

Here, grade() is a C function that requires two parameters, midterm mark and final mark; and returns the grade.

This may solve the problem of computational incompleteness of the relational model. In addition, since POSTGRES is still based on the concept of the relational model, it implies that one who has been familiar with the relational model may capture the idea in a shorter time than start learning on a new model.

1.2.3 Procedures as relations

The approach in this thesis generalises the notion of procedural abstraction in ordinary programming languages as a special form of relations and adapts the existing operations on relations to simulate the ones on procedures. We shall first introduce the concept of domain algebra, which sparks our approach.

Domain algebra is a set of operations that manipulate the attributes of relations. The mechanism of domain algebra allows one to perform operations among attributes of the same tuple and on the same attributes over a set of tuples. For example, a relation *Mark* has three attributes -*stu_id, midterm*, and *final*. Suppose we want to compute the total mark of each student, which is the summation of the midterm mark and the final mark. We may write down the formula of the total mark as below:

$$total = midterm + final$$

PRTV [45] treated the above formula as a relation of infinite tuples on three attributes, *total, midterm*, and *final*. We might then use the natural join of the

[2]QUEL, derived from QUEry Language, is an implementation of the relational calculus, a counterpart of the relation algebra. ([15] p.209)

relation *Mark* and the above relation to perform the computation of the attribute *total*.

Obviously, we cannot store a relation of infinite tuples explicitly. It must be a *virtual relation*. Merrett suggested that, instead of forming those three attributes as a virtual relation, we may consider *total* as a *virtual attribute* defined on the two attributes, *midterm* and *final*. The value of *total* can be *actualised* on any relation containing the attributes *midterm* and *final* by naming the attribute name in a projection. [36]

There are two types of operations in the domain algebra—horizontal and vertical operations. The horizontal operations are used to describe the relationship within a tuple. The relationship can be any arithmetic expressions, as the example of *total* above, boolean expressions, or conditional expression (if-then-else).

The summation might be a simple example to demonstrate vertical operations of domain algebra. Suppose we want to sum the final mark of all students. The formula may be written as:

$$sum\_final = \textbf{red} + \textbf{of} \, final$$

where **red** indicates a reduction operation and the operator + is one of many operators that can be specified. When *sum_final* is actualised, its value is equal to the result from the + operation of the values in the attribute *final* of all tuples.

Although the domain algebra allows computations over attributes of relations, it does not provide the iteration mechanism such as do-while statement. Our approach is to move back to the original idea of virtual relations in PRTV. We describe computations among attributes as special relations, called *computations*. Attributes of relations perform as parameters of computations. We simulate the iteration mechanism by allowing computations to call themselves recursively. Moreover, we extend operations on relations (e.g., selection, projection, join) to computations to simulate the calling operation.

We also introduce states, a notion of persistent data element in ODBMS, into computations. States are inaccessible but can be updated via operations on computations. Moreover, we borrow the concept of instantiation in OOP/ODBMS to generate new copies of states. This leads to a limited form of object identity [25] to distinguish instantiations.

1.3 Thesis outline

This thesis is divided into five chapters. Chapter 1 provides an overview of the relational model as well as research trends in other database models. We end with the motivation of our approach. Chapter 2 introduces the general concept of Relix. Chapter 3 is the user's manual, which shows the syntax and semantics of computations, operations on computations, as well as examples. Chapter 4 discusses the implementation details. Chapter 5 is the conclusions. Some proposals for future work are also presented in the last chapter.

Chapter 2

An overview of Relix

Relix is briefly described in this chapter. The purpose is to provide enough background for readers to be able to understand the rest of the thesis. Therefore, we will present only a subset of Relix commands which relate to this work. Section 2.1 introduces the concept of Relix and some basic commands. Section 2.2 describes the relational algebra. Section 2.3 describes the domain algebra. Readers should bear in mind that there is no intention of presenting all the details of Relix in this thesis. The complete reference of Relix can be found in [28].

To avoid any confusion, the following convention is applied throughout the rest of the thesis:

- The **boldface** style is used for reserved words.
- The `typewriter` font is used for program output, examples, or commands that one types in.
- Anything enclosed in a pair of angle brackets (<...>) is a label which one has to substitute with some sequence of characters.
- Anything enclosed in a pair of single quotes ('...') is to be typed as it is.
- Anything enclosed in a pair of square brackets ([...]) is an optional word or phrase.

- The symbol '|' is an alternation notation. One can use the words/phrases on either side of '|'.

2.1 What is Relix?

Relix, which stands for **Rel**ational database on Un**ix**, is the interpreter that accepts relational algebra and domain algebra statements as described in [36]. It has been developed under the ALDAT project at McGill University since 1986. The main purpose of developing Relix is for use as an experimental version to demonstrate the concept of the relational database model.

The language used in Relix is divided into two main categories domain algebra and relational algebra. As implied by their names, domain algebra is a set of operations on domains; and relational algebra is on relations. In the interactive mode, Relix gives the prompt '>' whenever it is ready to receive input from users.

2.1.1 Domains and relations

In order to create a relation in Relix, one has to explicitly create the domains that the relation is defined on. Domains can be any of these five atomic types boolean; two types of integer, short and long; one type of floating point, real; and string.

To create a domain, use the following syntax:

domain <dom-name> <dom-type>;

For example,

```
> domain stu_id intg;
> domain lastname strg;
```

The first statement tells Relix to create a domain `stu_id` of type integer. The second one is to create a domain `lastname` of type string. However, if the domain already exists and has another type, there are two possible actions:-

1. An error message is generated if the domain is used by any relations and/or other domains, or,

2. An old domain is overwritten by the new one.

To create a relation, use the following syntax:

relation <rel-name>(<dom-list>) ;

For example,

```
> relation Student(stu_id, lastname, firstname);
```

Relix will create a relation `Student` that has three attributes, namely, **`stu_id`**, **`lastname`**, and **`firstname`**. All the above attributes have to be created as domains before the relation is created.

We shall note that, in Relix, any relation of degree 1 is called *scalar relation.* We can perform scalar operations, such as addition, on scalar relations. [38] A *singleton* scalar relation, which contains only one tuple, may be viewed as a single value. This concept permits us to use relations in arithmetic and other expressions, which we will exploit in chapter 3.

2.1.2 Basic commands in Relix

- **show domains or a domain**

  Syntax

  sd!

  or

  sd!!<dom-name>

  Description

  Relix will show the name, type, and other information associated with all domains in the system or a particular domain.

  Example

  ```
  >sd!!lastname
  ```

- **show relations or a relation**

  Syntax

  sr!

  or

  sr!![<rel-name>]

Description

Relix will show the name, degree, and other related information of all relations in the system or a particular relation.

Examples

```
> sr!!Student
```

- **show details of relations or a relation**

  Syntax

  srd!

  or

  srd!![<rel-name>]

  Description

  Relix will show all the domains defined on all relations in the system or a particular relation.

  Examples

  ```
  > srd!!Student
  ```

- **print a relation**

  Syntax

  pr!!<rel-name>

  Description

  Relix will print all data in the specified relation.

  Examples

  ```
  > pr!!Student
  ```

- **delete a domain**

  Syntax

  dd!!<dom-name>

Description

Relix will delete the specified domain.

Examples

```
> dd!!stu_id
```

- **delete a relation**

  Syntax

  dr!!<rel-name>

  Description

  Relix will delete the specified relation.

  Examples

  ```
  > dr!!Student
  ```

- **quit**

  Syntax

  q!

2.2 Relational algebra

Relational algebra is a set of operations on relations. Operands and results of relational algebra are always relations. The relational algebra in Relix is an extension set of Codd's original relational algebra [10, 11], proposed by Merrett [35].

Two relations are used as examples throughout this chapter:

`Student(stu_id, lastname, firstname)`

stu_id	lastname	firstname
9300435	Smith	John
9213651	White	Adam
9204560	Ford	Watson
9102214	Lee	Sandra
9380009	Warrant	Roger
9011137	Pinn	Michael

`Mark(stu_id, assignment, midterm, final)`

stu_id	assignment	midterm	final
9011137	25	18	45
9102214	17	13	32
9204560	23	10	20
9213652	21	18	40
9380099	28	17	35
9300345	27	16	44

Some relational algebra operations are:

- **assignment**

  Syntax

  <rel-name> <- <rel-expression>;

  Description

  Relix will creates a new relation of the name on the left hand side by copying all the attributes and data from the relation on the right hand side.

  Examples

  ```
  > New_mark <- Mark;
  ```

- **increment**

  Syntax

<rel-name> <+ <rel-expression>;

Description

Relix will appends all but the duplicated tuples of the relation on the right hand side to the relation on the left hand side. Both relations must be defined on the same set of attributes.

Examples

Suppose there is a relation **Student1** that has the following tuples:

```
Student1(stu_id, lastname, firstname)
         ------  --------  ---------
         8905412 Roy       Patrick
         9213651 White     Adam
         9102214 Lee       Sandra
         8730027 Muller    Kirk
```

The statement

```
> Student1 <+ Student;
```

will give the following result to the relation **Student1**.

stu_id	lastname	firstname
8730027	Muller	Kirk
8905412	Roy	Patrick
9011137	Pinn	Michael
9102214	Lee	Sandra
9204560	Ford	Watson
9213651	White	Adam
9300435	Smith	John
9380009	Warrant	Roger

```
relation:  "Student1" has "8" tuple(s)
```

- **selection**

  Syntax

  where <selection-clause> in <rel-expression>

  Description

  Selection is an operation that selects the tuples in the relation that satisfy the condition in the selection clause. The result of the selection operation is a relation that can be used in any operation where a relation expression is required as an operand, e.g., in the assignment statement.

  Examples

  ```
  > smith <- where lastname = "Smith" in Student;
  ```

  The result is

stu_id	lastname	firstname
9300435	Smith	John

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

- **projection**

  Syntax

  '['<projection-list>']' in <rel-expression>

  Description

  Projection is an operation that extracts the subset of attributes specified in the projection list of the relation. The result is a relation that can be placed wherever a relation expression is required as an operand.

  Examples

  ```
  > stu_name <- [firstname, lastname] in Student;
  ```

  The result is

```
firstname  lastname
Adam       White
John       Smith
Michael    Pinn
Roger      Warrant
Sandra     Lee
Watson     Ford
relation:  "stu_name" has "6" tuple(s)
```

- **T-selector**

  Syntax

  '['<projection-list>']' **where** <selection-clause> **in** <rel-expression>

  or

  <rel-name>'{' <value>',' <value>',' ... '}'

  Description

  T-selector is a combination of selection and projection.

  The second syntax is called *positional notation.* One can supply values for attributes as if the attributes were ordered in the same way as when the relation was declared. Any positions of attributes that left blank are output. Compared to the original T-selector syntax, the unsupplied attributes are equivalent to the projection list; and the supplied attributes are the selection clause. The notation is quite limited because it allows users to express selection clauses which use the equal sign and the AND operator only.

  Examples

  ```
  > 3rd_year <- [lastname] where stu_id < 9200000 in Student;
  ```

  The example shows that the relation `3rd_year` contains all tuples of the attribute `lastname` with `stu_id` less than 9200000.

  ```
  > Lee's_id <- Student{,"Lee","Sandra"}
  ```

The above example shows the command that supplies the value "Lee" and "Sandra" for the attributes **lastname** and **firstname** respectively of the relation **Student**. The position of the first attribute, **stu_id** is left blank to designate the output attribute. The equivalent version of the example is

```
> Lee's_id <- [stu_id] where lastname = "Lee" and
                                    firstname = "Sandra" in Student
```

- **μ-join**

  Syntax

  <rel-expression> '[' [<dom-list>] <μ-join-op>
  [<dom-list>] ']' <rel-expression>

  Description

  μ-join is a family of join operations that combine two relations by operations on sets, e.g., union, intersection, and difference. Two of them are mentioned:

  - natural join is an operation that combines tuples of the two relations that have equal values on the join attributes. We used the keyword **ijoin**, which comes from *intersection join*, or **natjoin** to represent natural join. Formally, we can define the natural join of R(U,V) and S(X,Y) on the join attributes U and X as

  $$R\,[U\; ijoin\; X]\; S = \{(u,v,x,y) \mid (u,v) \in R\; and\; (x,y) \in S\; and\; u = x\}$$

  - union join is an operation that is a union of the set of tuples from the natural join, with the tuples from the relations of both sides that are not equal to each other in the join attributes, expanded the missing attributes by the DC *null value.*[1] Union join is represented by the keyword **ujoin**. We formally define the union join of R(U,V) and S(X,Y) as

  $$R\,[U\; ujoin\; X]\; S = R\,[U\; ijoin\; X]\; S \cup left\ wing \cup right\ wing$$

  $$left\ wing = \{(u,v,DC,DC) \mid (u,v) \in R\; and\, \forall y((u,y) \notin S)\}$$

  $$right\ wing = \{(DC,DC,x,y) \mid (x,y) \in S\; and\, \forall v((x,v) \notin R)\}$$

[1]DC stands for "don't care", a special value that describes irrelevant information [36]

Examples

```
> RijoinS <- Student ijoin Mark;
```

The result is

stu_id	lastname	firstname	assignment	midterm	final
9011137	Pinn	Michael	25	18	45
9102214	Lee	Sandra	17	13	32
9204560	Ford	Watson	23	10	20

```
relation:  "RijoinS" has "3" tuple(s)
```

```
> RujoinS <- Student ujoin Mark;
```

The result is

stu_id	lastname	firstname	assignment	midterm	final
9011137	Pinn	Michael	25	18	45
9102214	Lee	Sandra	17	13	32
9204560	Ford	Watson	23	10	20
9213651	White	Adam	DC	DC	DC
9213652	DC	DC	21	18	40
9300345	DC	DC	27	16	44
9300435	Smith	Roy	DC	DC	DC
9380009	Warrant	Roger	DC	DC	DC
9380099	DC	DC	28	17	35

```
relation:  "RujoinS" has "9" tuple(s)
```

- **σ-join**

Syntax

<rel-expression> '[' [<dom-list>] <σ-join-op>
[<dom-list>] ']' <rel-expression>

Description

σ-join is a family of join operations that combine two relations by set comparisons. Three of them are discussed here:

- inclusion (**sup** or **div**) of R(U,V) in S(V) is an operation that gives a relation RsupS(U) such that the sets of V of the same value u in R are a superset of the set of V in S.

- natural composition (**icomp**) of R(U,V) and S(V) is an operation that gives a relation RicompS(U) such that some elements of the sets of V of the same value u in R are elements of the set of V in S. In fact, it is a relation that comprises all attributes but the join attributes of *R* [*ijoin*] *S*.

- separation (**sep**) of R(U,V) in S(V) is an operation that gives a relation RsepS(U) such that the sets of V of the same value u in R have no element that belongs to the set of V in S.

Examples

To demonstrate the σ-join, two relations are given.

Register(student,	course)	CompSci(course)
-------	------	------
Smith	CS100	CS100
Smith	CS202	CS202
White	CS100	CS208
White	CS202	
White	CS208	
White	MA100	
Ford	MA100	
Lee	CS202	
Lee	MA100	

Suppose `Register` is a relation of the courses that students are registered in, and `CompSci` is a relation of all the courses given by Computer Science Department. The question "Find students who are registered in *all* the courses given by Computer Science Department?" is written as:

```
> RsupS <- Register sup CompSci;
```

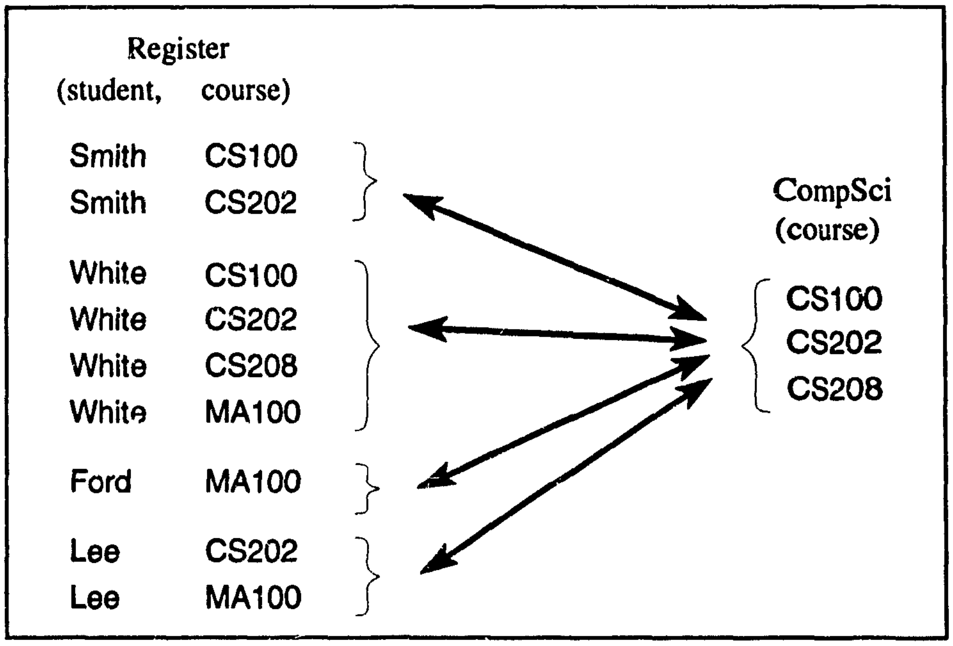

Figure 2.1: Set comparisons between two relations

In Figure 2.1, the four sets on the left hand side are tuples of the relation `Register` grouped on the same value of attribute `student`. The set on the right hand side contains tuples of the relation `CompSci`. The inclusion operation picks the students whose set of courses are a superset of the set of courses on the right hand side.

The result is:

```
student
White
relation:  "RsupS" has "1" tuple(s)
```

The question "Find students who are registered in *some* of the courses given by Computer Science Department?" is written as:

```
> RicompS <- Register icomp CompSci;
```

The natural composition operation does the intersection between the four sets and the one on the right hand side and picks the students whose result from the intersection are not empty.

The result is:

```
student
-------
Lee
Smith
White
-------
relation:  "RicompS" has "3" tuple(s)
```

The question "Find students who are registered in *none* of the courses given by Computer Science Department?" is written as:

```
> RsepS <- Register sep CompSci;
```

The separation operation does the intersection and picks the students whose results from the intersection are empty.

The result is:

```
student
-------
Ford
-------
relation:  "RsepS" has "1" tuple(s)
```

In general, the second operand may have some attributes that are not the attributes of the first operand. The σ-join between R(U,V) and S(V,Y) is done by grouping sets of V of the same value u in R (say $\mathcal{U}$) and sets of V of the same value y in S (say $\mathcal{Y}$); comparing each of sets in $\mathcal{U}$ to each of sets in $\mathcal{Y}$ according to the condition of the σ-join (e.g., superset, non-empty intersection, empty intersection); and returning a relation on attributes (U,Y) of the tuples that satisfy the above condition.

2.3 Domain algebra

Besides creating a domain by declaring its type as in section 2.1.1, one can build a new domain by expressing the domain as operations on existing domains in the system. This mechanism is called *domain algebra* [36].

The syntax of the command is

let <dom-name> **be** <dom-expression>;

Relix will create a domain name associated with the domain expression. Domain expressions can be written in one of the following types:

- scalar operations

  This is the simplest type of operations. Basically, the expressions deal with arithmetic operations; the conditional expression—if-then-else; and constants. Below are some examples.

  - constant domain

    ```
    let one be 1;
    let myname be "Joe";
    let TRUE be true;
    ```

    Note that `true` is a reserved word representing the boolean value "true". The capitalised `TRUE` is a domain name.

  - unary operators

    ```
    let FALSE be not TRUE;
    let debt be -income;
    ```

  - binary operators

    ```
    let mark be assignment + midterm + final;
    let total_interest be principle * time * interest_rate;
    ```

  - conditional expression

    ```
    let grade be if mark > 50 then "Pass" else "Fail";
    ```

  - predefined functions

    ```
    let area_triangle be sqrt(a**2 + b**2 + c**2) / 2;
    ```

    Note the area of a triangle if the length of the three sides of the triangle are a,b and c is a half of the square root of the summation of the square of each side. Here `sqrt` is a predefined function.

- type casting

  ```
  let l_intg be (long salary);
  ```

  Here, **salary** is a domain of type real. The new domain, **l_intg** is defined to be a long integer that has a value equivalent to **salary**.

- reduction

  Suppose we want to find a summation of an attribute in a relation. We need a mechanism to process values across tuples of the relation. Merrett called this mechanism the *vertical* operations [36].

  Simple reduction produces a single result from the values of all tuples of a single attribute in the relation. The syntax is

  let <dom> **be red** <opr> **of** <dom>;

  For example,

  ```
  let sum be red + of mark;
  let cnt be red + of 1;
  ```

  The first statement expresses the domain **sum** as a summation of domain **mark**. The second is the summation of the constant 1.

- equivalence

  Reduction can also produces a number of different results from different sets of tuples in a relation. Instead of calculating from all tuples, we group tuples into different sets by specifying the "grouping" attributes. The syntax is

  let <dom> **be equiv** <opr> **of** <dom> **by** <dom-list>;

  For example,

  ```
  let sum_by_type be equiv + of product by type;
  ```

  The statement gives the summation of the domain **product** of the same value of the domain **type**.

- functional mapping

  Functional mapping performs the accumulation of the operator specified by the "ordering" attributes. The syntax is

  let <dom> **be fun** <opr> **of** <dom> **order** <dom-list>;

  For example,

  ```
  let accum_prod be fun + of product order year;
  ```

- partial functional mapping

  Partial functional mapping is a functional mapping controlled by the "grouping" attributes. The syntax is

  let <dom> **be fun** <opr> **of** <dom> **order** <dom-list> **by** <dom-list>;

  For example,

  ```
  let acc_prod_type be fun + of product order year by type;
  ```

Chapter 3

User's manual

We devote this chapter to the detail of computations. Section 3.1 explains the concept of computations. Section 3.2 introduces a special type of computations that have hidden states. Then we propose the operations on computations in section 3.3 and formally describe the grammar and semantics of computations in section 3.4. We end the chapter by showing some examples.

3.1 Basic concept of computations

The notion of computations covers three aspects:

- it represents procedural abstraction;
- it is symmetric;
- therefore, it is close to the notion of constraints as in [43].

For example, a constraint $vel = dist/time$ expresses a relationship between velocity, distance, and time, a fundamental physic law. The computation is

```
comp Displacement(dist, time, vel) is
def vel = dist / time;
vel <- dist / time
alt
dist <- vel * time
alt
time <- vel * dist;
```

Here, `Displacement` is the name of the computation. `Dist`, `time`, and `vel` are attributes.

The body of a computation begins after the keyword **is**. We say that computations represent procedural abstraction because each computation contains algorithms for evaluating the values of subsets of attributes, when values are supplied for the rest of the attributes. These algorithms, which we call *blocks*, are alternative forms that express any attribute(s) to be derived from the rest of attributes. We do not propose mechanisms to enforce consistency of those blocks: it is up to users to write code for each block and to relate them to one another.

A special type of blocks called *predicate clause* defines the constraint of a computation. It begins with the keyword **def** followed by a boolean expression, which is used when one supplies values for all attributes of the computation. The output value is the result of evaluating the boolean expression, which is a boolean value.

In this example, the statement

```
def vel = dist / time;
```

is the predicate clause of the computation `Displacement`. When one supplies values for all the attributes, the above statement is evaluated and a result of boolean value is returned.

Ordinary blocks are separated from one another by the keyword **alt**. Blocks may contain one statement or a sequence of statements.

In this example, each block of the computation `Displacement` contains only one statement. The first block is used when one supplies values for the attributes `dist` and `time`. Then, the value of the attribute `vel` is computed by the statement in the block and returned as an output attribute. In the same manner, if one supplies values for the attributes `vel` and `time`, then the value of the attribute `dist` is computed, by the statement in the second block, and returned. The third block is used when one supplies values for the attributes `vel` and `dist`.

Computations are symmetric. A computation can has more than one set of input attributes. This concept makes computations go beyond the notion of procedures or functions that each procedure has only one set of input and output parameters.

We can represent the type of the above computation as the union of the type of the predicate clause and the types of the blocks as follows:

$$\begin{aligned}&(\mathit{dist{:}real} \times \mathit{time{:}real} \times \mathit{vel{:}real}) \rightarrow \mathit{boolean} \;\mid\\ &(\mathit{dist{:}real} \times \mathit{time{:}real}) \rightarrow \mathit{vel{:}real} \;\mid\\ &(\mathit{time{:}real} \times \mathit{vel{:}real}) \rightarrow \mathit{dist{:}real} \;\mid\\ &(\mathit{dist{:}real} \times \mathit{vel{:}real}) \rightarrow \mathit{time{:}real}\end{aligned}$$

Technically, when one supplies values for attributes, the system searches for the corresponding type, picks up the block of that type and evaluates the predicate clause or the statements in the block. For example, suppose one supplies values for `dist` and `time`. The system will find the type

$$(\mathit{dist{:}real} \times \mathit{time{:}real}) \rightarrow \mathit{vel{:}real}$$

and evaluate the value of `vel`. Likewise, one may supply values for `time` and `vel`. This time, the type

$$(\mathit{time{:}real} \times \mathit{vel{:}real}) \rightarrow \mathit{dist{:}real}$$

is searched and the value of `dist` is evaluated using the statement in the block corresponding to the type. In contrast, if one supplies only a value for `dist`, the system cannot evaluate because the type

$$\mathit{dist{:}real} \rightarrow (\mathit{time{:}real} \times \mathit{vel{:}real})$$

is not defined in the computation and an error message is returned.

Merrett suggested that we can think of computations as compressed forms of relations [37]. For the purpose of illustration, let us assume the domains of the three attributes of the above example are positive integers. Then we may write the computation `Displacement` as a relation of infinite tuples as below:

```
Displacement(dist, time, vel)
             ----  ----  ---
              1     1     1
              2     1     2
              3     1     3
              ⋮     ⋮     ⋮
              2     2     1
              4     2     2
              6     2     3
              ⋮     ⋮     ⋮
```

From the above format, we may simply imagine that if one supplies values for any two attributes, the process of getting the answer is the table lookup algorithm. In Relix, it is the T-selector. For example, the statement

```
> R <- [vel] where dist=6 and time=2 in Displacement;
```

will return a scalar relation of one attribute, vel, that has one tuple of value 3 to the name R.

This implies that the T-selector can be applied to computations as well. Furthermore, Relix provides the positional notation, which is somehow closer to the notation of calling a procedure. Its syntax may be easier to understand than the general T-selector syntax when applied to computations. For example, the statement

```
> R <- Displacement{6,2,};
```

is equivalent to the above T-selector example. Likewise, the syntax allows the last comma to be omitted whenever the last attribute is an output attribute. Therefore, we can also write as below:

```
> R <- Displacement{6,2};
```

The statement

```
> R <- Displacement{,2,3};
```

will cause the computation `Displacement` to use the block of the statement `dist <- vel * time`. The result relation from the evaluation that has a tuple of the attribute `dist` of the value 6 will be assigned to the relation `R`.

3.2 Stateful computations

Readers may notice that computations discussed in the previous section are purely functional. In other words, a computation will give the same answer if one supplies the same values for the same set of input attributes no matter how many times s/he evaluates it. In this section, we introduce another type of computations called *stateful computations.* They are computations that have hidden states. These states are persistent and affect the output in next invocations.

For example, a simple stateful computation, an accumulator

```
> comp Accum(no#:seq, in, total) is
total <- old total initial 0 + in;
```

The **seq** keyword, as in `no#:seq`, states that the attribute `no#` is a *sequencing attribute.* Its function is to designate the order of tuples in relations joining with stateful computations. As a matter of fact, we do not need sequencing attributes in any operations on computations except the natural join operation. At this moment, readers are asked to think of any sequencing attribute as an extra unused attribute in stateful computations.

The **old** keyword, as in `old total`, introduces a stateful variable. It is automatically updated by copying the value from the variable of the same name *at the end* of the evaluation.[1]

The keyword **initial** sets the following constant as an initial value to the stateful

[1]This mode of evaluation is correspondent to the *simultaneous mode* in [40], which all states are updated in parallel at the end of the computation. Another possible mode of evaluation is the *successive mode*, which the value of a variable is updated to its stateful variable immediately after change (by the assignment statement). These may apply to methods of finding solution of linear equations: the simultaneous mode applies to the Jacobi iterative method and the successive mode to the Gauss-Seidel iterative method.

variable name that places before it. (The above expression will be parsed as `(old total initial 0) + in`.)

Below is an example of an evaluation of the computation `Accum`.

```
> R <- Accum{300};
```

will give a tuple of the attribute `total` of the value 300 to the relation `R`.

Repeatedly, we enter the command again,

```
> R <- Accum{300};
```

Now, the value is 600.

The result is 600, instead of 300, because the output attribute `total` depends on not only the input attribute `in` but also the stateful variable `old total`, which has the value 300 at the time before the second execution.

3.3 Operations on computations

Operations on computations are designed to be similar to the ones on relations. We even reuse two commands on relations: delete and print.

The commands on computations are:

- **delete**

  Syntax

  dr!!<comp-name>

  Examples

  ```
  > dr!!Displacement
  ```

  Description

  The command deletes a computation. Any computation that has been *called with instantiation* by other computations cannot be deleted until its callers have been deleted. We will discuss more on the calling mechanism in the next section.

- **print**

  Syntax

  pr!!<comp-name>

  Examples

  ```
  > pr!!Accum
  comp (no#:seq, in, total) is
  total <- old total initial 0 + in;
  ```

  Description

  The command prints the body of a computation

- **show**

  Syntax

  sc!

  or

  sc!!<comp-name>

  Examples

  ```
  > sc!
  Displacement(dist:real, time:real, vel:real)
  [dist time vel ] -> []
  [dist time ] -> [vel ]
  [time vel ] -> [dist ]
  [dist vel ] -> [time ]
  Accum(no#:long,seq, in:long, total:long)
  [no#,in ] -> [total ]
  ```

```
> sc!!Accum

Accum(no#:long,seq, in:long, total:long)

[no#,in ] -> [total ]
```

Description

The command shows the types of all computations in the system or a particular computation.

The statements on computations are:

- **create**

  We discuss the syntax and semantics of the computation in the next section.

- **assignment**

  Syntax

  <new-comp-name> <- <comp-expression>;

  Examples

  ```
  > NewDisp <- Displacement;
  ```

  Description

  An assignment creates a new computation with a name on the left hand side (LHS) by copying the body and hidden states, if any, from the computation on the right hand side (RHS). The operation fails and an error message is given when the name on LHS has already been used. This is different from the assignment on relations. The LHS name will be overridden in the assignment statement on relations whether it has already been used or not.[2]

- **instantiation**

  Syntax

[2]The difference of the assignment operator on computations and relations is a design decision. We protect the case that one may lose his/her work when a computation is accidentally overridden. The cost we pay is the flexibility of assigning to any name freely, as in the case of relations. However, one might assign a computation to an already used name by deleting the name first.

new <comp-name>

Examples

```
new Accum
```

Description

An instantiation takes a computation as an operand and creates a new computation by copying the body and hidden states, if any, from the operand. However, the hidden states of the result computation are reset to their initial values. An instantiation can be placed wherever a computation expression is required, e.g., in the assignment statement.

```
NewAccum <- new Accum;
```

- **T-selector**

  Syntax

  '['<projection-list>']' **where** <selection-clause> in <comp-expression>

  or

  <comp-expression>'{' <value> ',' <value>',' ...'}'

  Description

  T-selector is a mechanism of evaluating a computation by supplying values for a subset of attributes and receiving the values of the rest as results, which are always relations. When using the first syntax, it is allowed to use only the equal sign (=) and AND operator in the selection clause.

- **natural join**

  Syntax

  <comp-name> (**ijoin** | **natjoin**) <rel-expression>

  or

  <comp-name> '[' <attribute-list> (**ijoin** | **natjoin**)
  <dom-list> ']' <rel-expression>

Description

We only consider the natural join operation between a computation and a relation. The operation behaves as if the relation were fed in tuple by tuple to be evaluated in the computation. The output is a relation whose attributes are the union of the attributes of the computation and the attributes of the input relation.

In stateful computations, different orders of tuples fed in may lead to different outputs. We solve the problem by introducing a set of attributes used as sequencing attributes, for example, the attribute `no#` in the computation `Accum` in the previous section. Then, before the evaluation step, we sort the input relation on the sequencing attributes. In the case where there is more than one tuple on the same sequencing attribute(s) only the first tuple is evaluated and the rest are ignored.

3.4 Formal syntax

We illustrate the formal syntax of computations in this section. The syntax will be given in BNF (Backus-Naur Form). We give the description before the syntax and then show some examples. The convention of the BNF is briefly summarised below.

BNF contains a set of rules. A rule comprises its name enclosed in a pair of angle bracket (<..>), an assignment (::=), and its definition. A definition can be written as the combination of the following syntax:

- A rule's name, which refers to its definition.
- Symbols enclosed in a pair of single quotes ('..'). This means the symbols are to be typed as they are.
- Symbols enclosed in a pair of square brackets ([..]). This means any one of the symbols is used.
- Symbols enclosed in a pair of curly brackets ({..}). This means the symbols are optional.

- The metasymbol '|', which is an alternation notation. One can use the symbol on either side of '|'.
- A pair of round brackets ((..)), which is a grouping notation.
- The metasymbol '*', which means the symbol before it may appear zero or more times.
- The metasymbol '+', which means the symbol before it may appear one or more times.
- The metasymbol '-' appearing in a pair of square brackets. This indicates continuous ASCII-ordering symbols, e.g., [0-9] is equivalent to [0123456789].

3.4.1 Identifiers

An identifier is a combination of any length of characters, digits or the symbols underscore (_), number sign (#) and single quote ('). However, in the implementation, the length of any identifier is limited to 80 characters.
Syntax
    <identifier> ::= ([a-z] | [A-Z] | [0-9] | [_#'])+
Example
    a, Anne, dist, 0eq34, a_book, a', c#, seqno#, _name

3.4.2 Types of variables

There are five types of variables or constants in computations.
Syntax

    <type> ::= 'bool' | 'boolean' | 'short' | 'intg' | 'integer' | 'long'
                | 'real' | 'float' | 'strg' | 'string'

'bool', 'boolean' are used to designate boolean type.

'intg', 'integer', 'long' are for long integer, which is in the range of −2147483648 - 2147483647.

'real', 'float' are for floating point.

'short' is for a short integer, which is in the range of −32768 − 32767.

'strg', 'string' is for a string variable or constant.

3.4.3 Constants

A constant can be

- a boolean constant, 'true' or 'false';
- a null value constant, 'dc' (don't care) or 'dk' (don't know); [3]
- an integer number;
- a floating point number;
- a string, any sequence of characters within a pair of double quotes (" ").

A floating point number can be written in the exponential form (the exponential part after the character 'e' or 'E' attaches to a base part). We can also specify the type of constants by attaching the type after the constants separated by a colon (':'). Lastly, an integer number is always considered type 'long' unless any other type is specified.

Syntax

```
<constant>   ::= <bool_const> { ':' <type>}
               | <null_const> { ':' <type>}
               | <integer> { ':' <type>}
               | <floating> { ':' <type>}
               | <string> { ':' <type>}
<bool_const> ::= 'true' | 'false'
<null_const> ::= 'dc' | 'dk'
<sign>       ::= [−+]
<integer>    ::= {<sign>} ([0-9])+
<floating>   ::= {<sign>} <real>
<real>       ::= ([0-9])+ '.' ([0-9])*
               | ([0-9])* '.' ([0-9])+
               | [0-9] '.' ([0-9])* <expon>
```

[3]"don't care" and "don't know" are special values that describes the status of irrelevant information and missing data respectively. [36]

<expon> ::= [eE] <integer>

Examples

boolean constants:	true, false
null value constants:	dc, dk
long constants:	12, −34, 56789022
floating point constants:	1.2, −3.5, 10., 0.5434, 1.2e2, 0.5E−3
string constants:	"Tom", "Hello world!", "#$@!*&%"
short constants:	134:short, −470:short

3.4.4 Stateful variables

Stateful variables are defined by stating the keyword **old** before identifiers. Both attributes and local variables can be stateful. Stateful variables may be initialised. If there is more than one initial value defined, only the first initial value is used and a warning message is issued. Stateful variables without initial values are set to default initial values ('false' for boolean type, 0 for numeric type, and null string for string type) with a warning message.
Syntax

<state-var> ::= 'old' <identifier> { <initial> <constant>}
<initial> ::= 'init' | 'initial'

3.4.5 Expressions

An expression can be recursively defined as
- a constant;
- an identifier;
- a stateful identifier;
- any of unary operators following by an expression;
- any of binary operators with two expressions;
- conditional expression (if-then-else);
- grouping expression with round bracket (...);

- predefined functions with an expression;
- type casting;
- computation calling.

Syntax

```
<expression>   ::= <constant>
                 | <identifier>
                 | <state-var>
                 | <unary-op> <expression>
                 | <expression> <binary-op> <expression>
                 | 'if' <expression> 'then' <expression> 'else' <expression>
                 | '(' <expression> ')'
                 | <function> '(' <expression> ')'
                 | '(' <type> <expression> ')'
                 | <computation-call>
<unary-op>     ::= '+' | '−' | '!' | 'not'
<binary-op>    ::= <compare-op> | <arith-op> | <logical-op>
                 | 'min' | 'max' | 'cat' | 'also'
<compare-op>   ::= '=' | '!=' | '~=' | '<=' | '<' | '> ' | '>='
<arith-op>     ::= '+' | '−' | '*' |'/' | '**' | 'mod'
<logical-op>   ::= 'and' | '&' | 'or' | '|'
<function>     ::= 'abs' | 'isknown' | 'round' | 'ceil' | 'floor' | 'sqrt'
                 | 'ln' | 'log' | 'log10' | 'acos' | 'asin' | 'atan'
                 | 'cos' | 'sin' | 'tan' | 'cosh' | 'sinh' | 'tanh'
```

The unary operators are plus sign ('+'), minus sign ('−'), and negator ('not' or '!'). Any expression which follows the plus or minus sign must be number type (integer or floating point). One which follows the negator must be boolean type.

The binary operators are

- Comparison operators : '=', '~=' or '!=', '<', '<=', '>', '>='.
- Arithmetic operators : '+', '−', '*', '/', '**' (exponentiation), 'mod' (modulus operator). The operands must be number type (integer or floating point).

- Logical operators : 'and' or '&', 'or' or '|'. The operands must be boolean type.
- Concatenation operator ('cat'). The operator takes two expressions of string type and concatenates them together.
- Maximum operator ('max'). The operator compares two expressions and returns the greater one. The expressions can be type real, long, short or string.
- Minimum operator ('min'). The operator does the same as the maximum operator but returns the lesser one.
- Multi-valued operator ('also'). This operator returns both expressions. Used in the assignment statement, it returns multiple values to the name on the left hand side of the assignment operator. For example, an inverse of the function abs() is

  ```
  > comp InvAbs(x,iabs) is
  iabs <- x also -x;
  ```

  The variable `iabs` will return two values, x and −x.

To avoid ambiguity in the order of the operators, we define the precedence and associativity in Table 3.1.

The higher the position of the operator in the table, the higher its order of precedence is. For example, an expression $3 + 5 * 4 ** 3 = 34 \& 15 < 5$ will be interpreted as $((3 + ((5 * (4 ** 3)))) = 34) \& (15 < 5)$.

The conditional expression (if-then-else) requires three expressions. if-expression must be boolean type. then-expression and else-expression must be compatible types.

There are 17 predefined functions:

- abs() returns an absolute value of the expression.
- isknown() returns 'true' when the expression is any value but not 'dk'.
- round() returns the greatest integer that less than or equal to the expression.
- ceil() returns the least integer that greater than or equal to the expression.
- floor() returns the greatest integer that less than or equal to the expression.
- sqrt() returns the positive square root of the expression.
- ln() or log() returns the logarithm to the natural base of the expression.

Operator	Associativity
'old'	(unary-op)
'initial'	non-associative
'not', '!'	(unary-op)
'**'	right-associative
'*' '/' 'mod'	left-associative
'+' '−'	left-associative
'max' 'min'	left-associative
'cat'	left-associative
'=' '!=' '~=' '<' '<=' '>' '>='	non-associative
'&' 'and'	left-associative
'|' 'or'	left-associative
'also'	non-associative

Table 3.1: Precedence and associativity of operators

- log10() returns the logarithm to base 10 of the expression.

- acos(), asin(), atan() return inverse cosine, inverse sine, and inverse tangent functions of the expression.

- cos(), sin(), tan() return cosine, sine, and tangent functions of the expression.

- cosh(), sinh(), tanh() return hyperbolic cosine, hyperbolic sine, and hyperbolic tangent functions of the expression.

3.4.6 Calling a computation

As with functions and procedures in traditional programming languages, computations can be called by other computations, In fact, we use the positional notation to embody computation calling. The return value must be a singleton scalar relation, which is implicitly treated as a variable in computations. A computation can recursively call itself as well.

Syntax

<computation-call> ::= { 'new' } <identifier> '{' <actual-params> '}'
<actual-params> ::= <empty> | <expression> | <actual-params> ','
| <actual-params> ',' <expression>
<empty> ::= (' ')*

For example, a computation that computes the factorial.

```
> comp Factorial(n,fac) is
fac <- if n=0 then 1 else n * Factorial{n-1};
```

Suppose one enters the command

```
> R <- Factorial{3};
```

The computation **Factorial** will assign the value 3 to **n** and try to evaluate the value of **fac**. The process of evaluation may look like a downward calculation in the table below.

n	fac
3	3 * Factorial{2}
2	2 * Factorial{1}
1	1 * Factorial{0}
0	1

Finally, the computation **Factorial** stops at n=0 and returns the value of **fac** upward to its caller. The final result is a relation of an attribute **fac**, which has one tuple of value 6.

One should be careful when calling stateful computations inside other computations. Suppose we want to use the computation **Accum** in section 3.2, which is a stateful computation, to represent a storage with a capacity of 100. A computation **IsFull** is written to report the status of the storage whether it is full or not.

```
> comp IsFull(no#, in, mesg) is
mesg <- if new Accum{no#,in} < 100 then "I'm full!"
        else "I need more.";
```

Here, the computation **Accum** is said to be *called with instantiation.* With the keyword **new**, the computation **IsFull** will instantiate a hidden state from the one in **Accum** and reset to the initial value when it is declared. After that, we run the statement

```
> Mesg <- IsFull{0, 68};
```

The relation **Mesg** will have a tuple of the attribute **mesg** of the value "I need more.". Note that the value supplied to the attribute **no#** in the above T-selector statement can be any value and there is no effect on the result since it plays the role of sequencing attribute only in the natural join operation.

In contrast, if the keyword **new** is missing in the declaration of the computation **IsFull** as below:

```
> comp IsFull(no#, in, mesg) is
mesg <- if Accum{no#,in} < 100 then "I'm full!"
          else "I need more.";
```

Every time we perform any evaluation (either T-selector or natural join) on the computation **IsFull**, the original state in **Accum** determines the result of the expression **Accum{no#,in}**. For example, we run the statement

```
> R <- Accum{0, 40};
```

Assume that the stateful variable **in** is originally 0. After the above statement, the stateful variable **in** in **Accum** is 40. Then we run the statement on the latter version of **IsFull** as exactly as the above one.

```
> Mesg <- IsFull{0, 68};
```

The statement updates the stateful variable **in** in **Accum** to 108 and returns the value "I'm full!". This might not the expected answer.

Forward reference is allowed in calling computations. In other words, computations can be called by other computations before their declaration. The called names are kept without checking their existence during the compilation process. They must be created, however, before the evaluation (T-selector or natural join) of the calling computations, or an error is reported.

However, the principle of forward reference does not apply to the case of calling with instantiation. Because we need to instantiate all the states from the called computations in the compilation process, we require the called computations to have been declared.

3.4.7 Declaration of computations

A computation can be created by using the following syntax:

<computation>	::= ‘comp’ <new-comp-name> <attributes> ‘is’ <body>
<new-comp-name>	::= <identifier>
<attributes>	::= ‘(’ <identifier> {‘:’ ‘seq’} (‘,’ <identifier> {‘:’ ‘seq’})* ‘)’
<body>	::= {<predicate-clause> } <block> (‘alt’ <block>)*
<predicate-clause>	::= ‘def’ <expression> ‘;’
<block>	::= <single-stmt> | <compound-stmt>
<single-stmt>	::= <identifier> **<-** < expression>
<compound-stmt>	::= ‘{’ {<local-vars> } (<single-stmt> ‘;’)+ ‘}’
<local-vars>	::= ‘local’ <vars-type-list> (‘,’ <vars-type-list>)* ‘;’
<vars-type-list>	::= <variable-list> ‘.’ <type>
<variable-list>	::= <identifier> (‘,’ <identifier>)*

<new-comp-name> must be unique among the set of relation and computation names in the system.

<attributes> is a list of domain names representing the attributes of the computation. The optional keyword **seq** is used to show that the designated attribute is a sequencing attribute.

<body> contains at most one predicate clause, which is designated by the keyword **def** followed by a boolean expression and a semicolon (‘;’), and at least one block. In the case of more than one block, the blocks are separated by the keyword **alt**.

<block> can be one statement or sequences of statements.

<single-stmt> is an identifier followed by the assignment operator (‘**<-**’) and an expression.

<compound-stmt> comprises an optional <local-vars> which is the declaration part of local variables, and sequences of statements; within the curly bracket.

The declaration of local variables starts with the keyword **local** followed by sets of variable names, a colon (':'), and a type with each set separated by a comma (','); and ends with a semicolon (';'). Variables used in the statements of any block must be either local variables of that block or attributes of the computation in which the variables belong to.

3.5 Examples

Note that we assume all attributes' names used in the following computations have already been declared as domain names with proper types before the declaration of computations (e.g., `domain pi real;`).

3.5.1 Constant computation

Here is an example of declaring a constant computation, **Pi**. Using the positional notation without attribute, the computation `Pi` returns a constant.

```
> comp Pi(pi) is
pi <- 3.141593;
> result <- Pi{};
> pr!!result
--------
 pi
--------
 3.141593
--------
relation:  "result" has "1" tuple(s)
```

Constant computations may be called in other computations. Here is an example of a computation to compute the area of circles that uses the computation `Pi` as a constant.

```
> comp CirArea(radius, area) is
area <- Pi{} * radius * radius; << use the computation Pi >>
```

3.5.2 Integer division

The example below shows a computation for integer division. The purpose is to show that computations may have more than one output attribute.

The computation `IntDiv` takes two input attributes, `dividend` and `divider`; and returns two output attributes, `quotient` and `remainder`.

```
> comp IntDiv(dividend, divider, quotient, remainder) is
{
quotient <- dividend / divider;
remainder <- dividend mod divider;
};
> result <- IntDiv{34, 3};
> pr!!result
-----------------------
 quotient  remainder
-----------------------
 11        1
-----------------------
relation:  "result" has "1" tuple(s)
```

3.5.3 Multi-valued computation

Computations may return more than one value to the same attribute.[4] For example, the computation `SqRoot` returns both positive and negative root.

```
> comp SqRoot(n,root) is
root <- sqrt(n) also -sqrt(n);
> result <- SqRoot{24};
> pr!!result
------------
 root
------------
 -4.898979
 4.898979
------------
relation:  "result" has "2" tuple(s)
```

[4]Some limitations applied. See details in section 4.5.2

iteration	1	2	3	4	5	6	...	n
a	1	1	2	3	5	8	...	Fib(n)
b	0	1	1	2	3	5	...	Fib(n-1)

Table 3.2: The values of a and b

3.5.4 Recursive computation

To demonstrate recursive computations, we consider finding the sequence of Fibonacci numbers, in which each number is the sum of the preceding two starting with 0 and 1:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

The Fibonacci numbers can be defined as follows:

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(\text{n-1}) + Fib(\text{n-2}) & \text{otherwise} \end{cases}$$

We may compute the Fibonacci numbers by an iterative method. Let us use a pair of integer a and b, initialised to 1 and 0 respectively. If we repeatly apply the parallel assignment

$a \leftarrow a + b$

$b \leftarrow a$

The values of a and b will be as in Table 3.2. [2]

We apply this method to the computation `FibIter`. The computation calls itself recursively until it reaches the stopping condition and returns the value back to its parent.

```
> comp Fib(n,fib) is
fib <- FibIter{1,0,n};
> comp FibIter(a,b,count,sum) is
sum <- if (count=0) then b else FibIter{a+b,a,count-1};
```

Suppose **Num** is a relation of the numbers 1 to 10. We may generate the first ten Fibonacci numbers by joining **Fib** with **Num**.

```
> pr!!Num

 ──
 n
 ──
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 ──
relation:  "Num" has "10" tuple(s)

> FirstTenFib <- Fib ijoin Num;
> pr!!FirstTenFib

 ──────
 n  fib
 ──────
 0   0
 1   1
 2   1
 3   2
 4   3
 5   5
 6   8
 7  13
 8  21
 9  34
 ──────
relation:  "FirstTenFib" has "10" tuple(s)
```

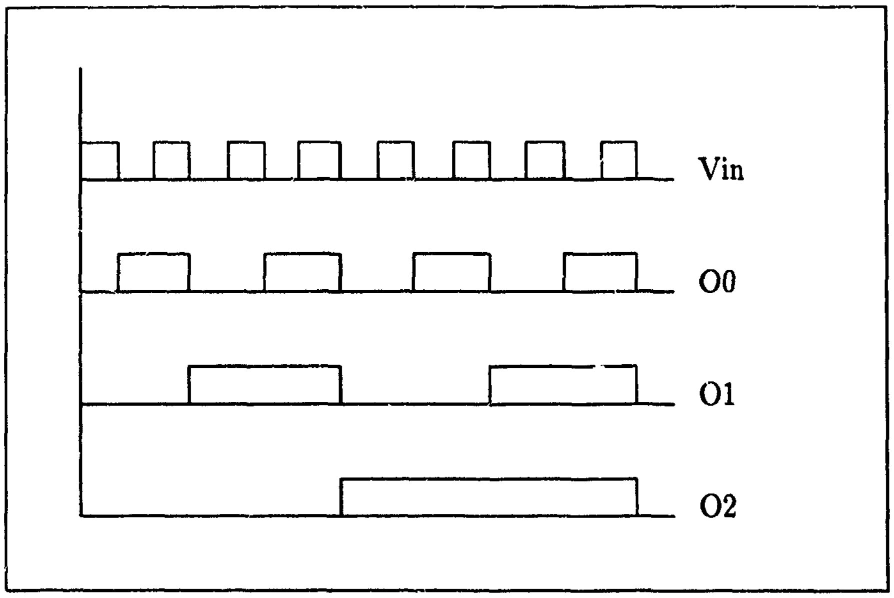

Figure 3.1: Wave forms of the 3-bit counter

3.5.5 Stateful computation

We use the frequency divider as an example of stateful computation. The output toggles whenever the input changes its state from 1(true) to 0(false). In Figure 3.1, Vin and O0 are the input and output signal respectively of the frequency divider circuit.

We can build a 3-bit counter using three frequency dividers as in Figure 3.2. The computation forms of the frequency divider and the 3-bit counter are

```
> comp FreqDiv(time:seq, vin, vout) is
vout <- if (old vin initial true=true and vin=false) then
         not old vout initial false else old vout;
> comp Counter(time:seq, vin, o0, o1, o2) is
{
o0 <- new FreqDiv{0, vin};
o1 <- new FreqDiv{0, o0};
o2 <- new FreqDiv{0, o1};
};
```

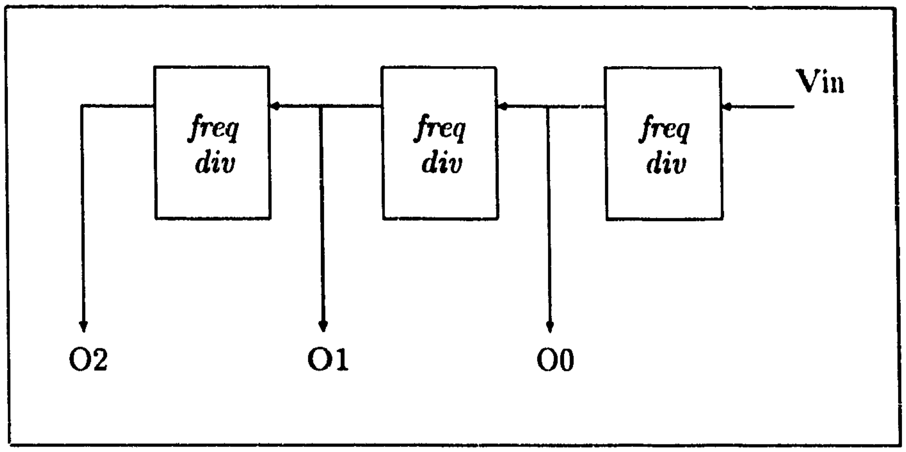

Figure 3.2: Diagram of the 3-bit counter

Suppose we have a relation `Pulse` which represents the wave form as in Vin in Figure 3.1. The natural join between the relation `Pulse` and the computation `Counter` gives the output attributes O0, O1, and O3 corresponding to the wave form in Figure 3.1. The following is an output in the form of a relation.

```
> result <- Pulse ijoin Counter;
> pr!!result
```

```
time  vin    o0     o1     o2
0     true   false  false  false
1     false  true   false  false
2     true   true   false  false
3     false  false  true   false
4     true   false  true   false
5     false  true   true   false
6     true   true   true   false
7     false  false  false  true
8     true   false  false  true
9     false  true   false  true
10    true   true   false  true
11    false  false  true   true
12    true   false  true   true
13    false  true   true   true
14    true   true   true   true
relation:  "result" has "15" tuple(s)
```

To ensure that the computation **Counter** begins counting from zero. We should use the statement

```
> result <- Pulse ijoin new Counter;
```

The keyword **new** will instantiate a new counter in which all states are reset to their initial values, regardless of any states remaining from the previous execution of the computation **Counter**.

3.5.6 The role of instantiation

Instantiation plays a very important role on stateful computations. Statements with and without the keyword **new** may lead to totally different results. Examples are two following T-selector statements on the stateful computation **Accum** in section 3.2. One with an instantiation as

```
> R <- new Accum{92};
```

will always give the result 92 to the relation R because the T-selector is done by executing a new instance of the computation `Accum`, which has its own state reset to the initial value of the corresponding state in `Accum`. The other without an instantiation as

```
> R <- Accum{92};
```

will depend on the last status of the hidden state in `Accum`.

Chapter 4

Implementation

This chapter is about the implementation of computations. Section 4.1 overviews the implementation of Relix. Section 4.2 describes how computations are represented. Algorithms for each operation on computations are given in section 4.3. Then, in section 4.4, we explain the functions of the parser, which performs the syntactic and semantic checking. Section 4.5 is the detail of the executor used in the evaluation of computations in the T-selector and the natural join operations.

Section 4.6 presents the mechanism of concurrency control.

4.1 Implementation of Relix

Relix consists of two main modules: a parser, generated by Lex [30] and Yacc [24]; and an interpreter, which is a C program. The parser performs syntax checking and generates intermediate codes. The interpreter branches to particular C functions corresponding to the intermediate codes.

Each relation is stored in a UNIX file whose name corresponding to the name of the relation. Due to UNIX limitations, relation names must be unique in the first fourteen characters, even though users may name relations up to 80 characters.

Databases, which are collections of relations, are equivalent to UNIX directories. Each database has its own data dictionary. Therefore, relations of the same name in different databases are not related to each other and may represent different things.

Data dictionary, information about domains and relations, is kept in the form of relations. We call these relations *system relations.* Data dictionary contains three system relations: *.dom*, *.rel*, and *.rd*,[1] which store information of domains, relations, and relationship between relations and domains respectively. We discuss only the system relations *.rel* since it is related to the implementation of computations. The relation *.rel* is defined as follows:

.rel (.rel_name:strg, .sort_status:short, .rank:short, .ntuples:long)

Each relation defined on Relix has a tuple in relation *.rel*, including *.rel* itself. Attribute *.rel_name* stores the relation names. *.Sort_status* and *.rank* tell Relix that whether the relations are sorted or not, and on how many attributes. *.Ntuples* indicates how many tuples are in the relations.

When we start Relix, all system relations are loaded into data structures in memory. Relix performs all operations directly on the data structures and updates the data back to the system relations whenever the data is changed.

The main flow of Relix can be summarised as the following algorithm:

1. Load system relations into data structures.

2. Wait for input from users.

3. Parse the input and generate intermediate codes.

4. Execute the intermediate codes.

5. Update system relations if necessary.

6. Go to step 2.

4.2 Representation of computations

Computations are implemented as special forms of relations. A computation is represented as three components:

[1]In Relix convention, names begin with period (.) are system names.

1. *A source file*—a file of the same name as the computation contains the source code of computation.

2. *An i-code file*—a file of the name ".<comp-name>.ic", which <comp-name> is replaced by the actual computation name, containing intermediate codes generated in the executable format in Relix.

3. *Interface information*—its declaration, type, local and stateful variables are kept in the form of tuples in system relations.

An example of a computation `Rpar` that computes the resistance `r_par` of two resistors `r1` and `r2` connected in parallel is used for the purpose of illustration throughout this section. The computation `Rpar` is written as below:

```
comp Rpar(r1,r2,r_par) is
def 1/r_par = 1/r1 + 1/r2;
{
local x:real;
x <- r1 + r2;
r_par <- if x=0 then 0 else r1 * r2 / x;
}
alt
{
local x:real;
x <- r2 - r_par;
r1 <- if x=0 then 0 else r2 * r_par / x;
}
alt
{
local x:real;
x <- r1 - r_par;
r2 <- if x=0 then 0 else r1 * r_par / x;
};
```

The computation Rpar contains four blocks: the predicate clause, which is the boolean expression after the keyword **def**, and three sequences of statements grouped within the brackets separated by the keyword **alt**. Note that there need be no bracket if a block contains only one statement.

4.2.1 Source file

Source files are kept only for the purpose of human reading. They do nothing in the execution process of computations. We generate a source file by intercepting all characters that users type in when they create a computation. Then we eliminate the name of the computation in the source file to make it anonymous. The source file of the computation Rpar will look like this:

```
comp (r1,r2,r_par) is
def 1/r_par = 1/r1 + 1/r2;
...
x <- r1 - r_par;
r2 <- if x=0 then 0 else r1 * r_par / x;
};
```

Note that the name Rpar in the first line was eliminated. The advantage of the anonymity, which we will see later, is we can simply copy the source file from the original computation to the new computation in the assignment and the instantiation operations.

4.2.2 I-code file

I-code files play a certain role in the execution of computations. Conceptually, they are sequences of mnemonic codes for a stack machine. For example, the i-code `push-name` tells Relix to push its operand onto the stack. The i-code `plus` tells Relix to pop two elements from the stack, do a plus operation, and push the result back to the stack. We discuss the details of i-code in section 4.4 and 4.5.

4.2.3 System relations

Each computation has its entry in relation *.rel.* The *.sort_status* of any computation is set to a constant CMPTN_STATUS (equals 16 in the current version) to be distinct from other types of relations. The *.rank* and *.ntuples* are set to RELIX_VOID (the status indicates the unused state, which equals −3).

In addition to the relation *.rel*, five system relations are designed to keep information related to computations. Table 4.1 illustrates the relations' names, their attributes and their functions.

The system relations contain information of the computation Rpar are shown:

```
.comp(.comp_name, .dom_pos, .dom_name, .type, .seq_attr)
      ----------  --------  ---------  -----  ---------

      Rpar        0         r1         1      0
      Rpar        1         r2         1      0
      Rpar        2         r_par      1      0

.comp_type(.comp_name, .block, .block_type, .code_offset)
           ----------  ------  -----------  ------------

           Rpar        0       7            0
           Rpar        1       3            154
           Rpar        2       6            360
           Rpar        3       5            570

.comp_local(.comp_name, .block, .var_name, .type)
            ----------  ------  ---------  -----

            Rpar        1       x          1
            Rpar        2       x          1
            Rpar        3       x          1
```

Relation *.comp* contains information of attributes, their positions, and their types. The type is represented as a short integer which can be deciphered as in Table 4.2.

Relation *.comp_type* contains information on types of computations, which is different from the types of attributes above. To avoid any confusion, we call the types of

Relation	Attribute name	Function
.comp		Attribute information
	(.comp_name : strg,	computation name
	.dom_pos : short,	position of attribute, begins at 0
	.dom_name : strg,	attribute name
	.type : short	type of attribute
	.seq_attr : short)	sequencing attribute status
.comp_type		Type of computation
	(.comp_name : strg,	computation name
	.block : short,	sequence number of block, begins at 0
	.block_type : long,	type of block
	.code_offset : short)	offset of intermediate code in the .ic file
.comp_local		Local variable
	(.comp_name : strg,	computation name
	.block : short,	sequence number of block
	.var_name : strg,	local variable name
	.type : short)	type of variable
.comp_state		State variable
	(.comp_name : strg,	computation name
	.block : short,	sequence number of block
	.instance : short,	sequence number of instance
	.var_name : strg,	state variable name
	.type : short,	type of variable
	.init_val : strg,	initial value in ASCII sortable format
	.act_val : strg)	current value in ASCII sortable format
.comp_depend		Dependency of stateful computation
	(.calling : strg,	caller
	.called : strg)	receiver

Table 4.1: System relations of computations

Type	Code
bool	0
real	1
string	2
short	3
long	4

Table 4.2: The codes representing types of variables

computations as *computation types*. We number each block of code sequentially starting from zero. The predicate clause is optional. It must be the first block (block 0), however, if it exists. Computations that have no predicate clause will start running the block number of ordinary blocks from 0, not 1. In the case of the computation `Rpar`, the predicate clause is considered block 0, The code for computing the attribute `r_par` block 1, and so on. It is not difficult to see that the computation type of `Rpar` is

block 0 : *(r1:real* × *r2:real* × *r_par:real)* → *boolean*

block 1 : *(r1:real* × *r2:real)* → *r_par:real*

block 2 : *(r2:real* × *r_par:real)* → *r1:real*

block 3 : *(r1:real* × *r_par:real)* → *r2:real*

We represent the computation type information in attribute *.block_type* as a long integer (32 bits) and mark the bit corresponding to the position of input attributes. Output attributes are the complement of input attributes. For example, the posi'ions of **r1**, r2, and **r_par** are 0, 1, and 2 respectively. The computation type of block 0 is $2^0 + 2^1 + 2^2 = 7$, of block 1 is $2^0 + 2^1 = 3$, and so on. With this method, we limit the number of attributes in any computation to 30. Any constant computation will have the computation type equals 0 (no input attribute).[2]

The attribute *.code_offset* indicates the offset of intermediate codes in the i-code file. For example, the code of block 2 begins at byte 360 of the i-code file.

[2]See an example in section 3.5.1

Relation *.comp_local* keeps the information of local variables of each block of computations. As shown above, each block of the computation `R_par` has its own local variable. Even though the variable names, `x`, are the same; they represent different positions of storage.

Another example is raised to demonstrate the content in the two remaining system relations, *.comp_state* and *.comp_depend.* Here is a computation represents a bank account.

```
comp Acct(no#:seq, dep, bal) is
bal <- old bal initial 0 + dep;
```

`No#` is a sequencing attribute. The result of `bal` is calculated by adding the deposit (`dep`) to the balance (**old** `bal`) A new account is defined by instantiating the computation `Acct`.

```
comp NewAcct(no#:seq, dep, bal) is
bal <- new Acct{no#,dep};
```

The system relation *.comp_state* that keeps the information of stateful variables of both computations is shown:

```
.comp_state
(.comp_name,.block,.instance,.var_name,.type,.init_val,.act_val)
 ---------- ------ --------- --------- ----- --------- --------
 Acct       -1     0         bal       3     -00000    -00000
 NewAcct    0      1         bal       3     -00000    -00000
```

Stateful variables are recorded in the relation *.comp_state.* The computation `Acct` has one stateful variable, `bal`. The value −1 in the attribute *.block* means the name is a stateful variable of an attribute. The value 0 in the attribute *.instance* means the variable is declared directly in `Acct`. The attribute *.init_val* and *.act_val* keep the initial and actual values of `bal`, represented in *ASCII sortable format* (the format used in Relix that transforms numbers into fixed-length strings so that they can be sorted by using ordinary string sorting function). Both values are set to the value after the keyword **initial**, which is zero. Initial values are not changed after the declaration of computations but actual values may be updated when computations are executed using the T-selector or the natural join operation.

The computation *NewAcct* does not declare any stateful variable explicitly. However, it *instantiates* a state by calling `Acct` with the keyword **new**, which means `NewAcct` will create a new set of stateful variable(s) by copying from the one(s) of `Acct` and resetting the actual value(s) to the initial value(s). We also say that `Acct` is an instance of the first block (*.block*=0) of `NewAcct`. A tuple in relation *.comp_state* records the stateful variable `bal` belongs to the first block of the computation `NewAcct`. The value 1 in attribute *.instance* means the state is derived from the first instance of the block.

Here is another example. Suppose we have a fixed-size buffer of 2048 bytes. We want to write a computation that returns a status TRUE if it successfully adds certain bytes of data into the buffer and does not overflow; and returns FALSE otherwise. The computation `NotOverflow` is

```
comp NotOverflow(no#:seq, bytes, result)
{
local size:intg;
size <- old size initial 2048 - bytes;
result <- if size < 0 then false else true;
<< Comment:set the value of "size" back if not successful >>
size <- if size < 0 then old size else size;
};
```

The information of stateful variables of the computation `NotOverflow` will be:

```
.comp_state
(.comp_name ,.block,.instance,.var_name,.type,.init_val,.act_val)
 ---------- ------ ---------- --------- ----- ---------- --------
 NotOverflow 0      0          size      3     -02048     -02048
```

The value in the attribute *.block* is 0 because the stateful variable `size` belongs to the first block. The value in the attribute *.instance* is 0 because `size` is directly

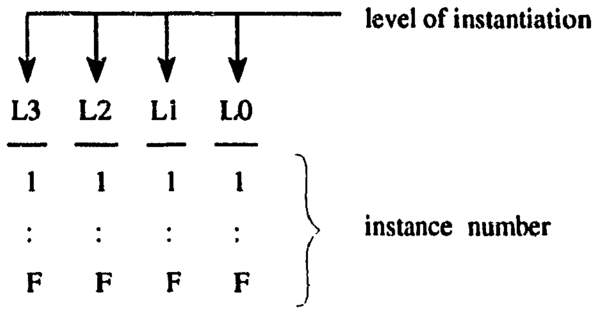

Figure 4.1: The detail of the attribute *.instance*

declared in the block, not derived from instantiating other computations as in the case of the computation `NewAcct`.

Formally, the attribute *.block* indicates the scope of stateful variables. It equals -1 when the name is a state of an attribute. Otherwise, it equals the block number of the block in which the name is a local state.

The attribute *.instance* equals 0 if the stateful variable is declared in the block (with the keyword **old**); or indicates *the level of instantiation* and *the instance number*, which is the number (starting from 1) indicating the order of appearance of the instance within the block. In this case, the four half-bytes of *.instance* support four levels of instantiation, each with instance numbers from 1 to 15(F) - see Figure 4.1.

In the example of frequency dividers and a counter in section 3.5, the computation `FreqDiv` is stateful. The computation `Counter`, which is a 3-bit counter, uses three frequency dividers. The instance numbers of those three `FreqDivs` are 1, 2 and 3 respectively according to the order of their appearances in the code. The level of instantiation is L0 above because the states come from the computation `FreqDivs` which were instantiated.

Suppose we write another computation `TwoCounter` which uses two 3-bit counters. Similarly, the instance numbers of those two `Counters` are 1 and 2 respectively. The level of instantiation, however, is L1 above since the states are not from the `Counters` but the `FreqDivs` which were instantiated by the `Counters`.

The system relation *.comp_state* of the above computations are (the values of the attribute *.instance* are shown in hexadecimal):

```
.comp_state
(.comp_name,.block,.instance,.var_name,.type,.init_val,.act_val)
---------- ------ --------- --------- ----- --------- --------
 FreqDiv    -1     0         vin       0     1         1
 FreqDiv    -1     0         vout      0     0         0
 Counter    0      0001      vin       0     1         1
 Counter    0      0001      vout      0     0         0
 Counter    0      0002      vin       0     1         1
 Counter    0      0002      vout      0     0         0
 Counter    0      0003      vin       0     1         1
 Counter    0      0003      vout      0     0         0
 TwoCounter 0      0011      vin       0     1         1
 TwoCounter 0      0011      vout      0     0         0
 TwoCounter 0      0012      vin       0     1         1
 TwoCounter 0      0012      vout      0     0         0
 TwoCounter 0      0013      vin       0     1         1
 TwoCounter 0      0013      vout      0     0         0
 TwoCounter 0      0021      vin       0     1         1
 TwoCounter 0      0021      vout      0     0         0
 TwoCounter 0      0022      vin       0     1         1
 TwoCounter 0      0022      vout      0     0         0
 TwoCounter 0      0023      vin       0     1         1
 TwoCounter 0      0023      vout      0     0         0
```

The values 0001, 0002, and 0003 of the attribute *.instance* in the tuples where *.comp_name*= "Counter" refer to the first, second and third instance in the code of the computation `Counter` respectively. The values 0011, 0012, and 0013 (in hexadecimal) in the tuples where *.comp_name*= "TwoCounter" refer to all instances instantiated by the first instance in the code of the computation `TwoCounter`. The values 0021, 0022, and 0023, similarly, refer to all instances from the second instance of the computation

TwoCounter.

The last system relation, *.comp_depend*, stores dependencies between stateful computations. The attribute *.calling* and *.called* store the name of the calling computations, or *caller*, and the computations being called, or *receiver* respectively. The purpose is to keep consistency of states and i-codes between callers and receivers.

The receivers cannot be deleted because all states of the caller were generated based on the states and i-codes of the receiver found at the time the caller was compiled. If the states and i-codes of the receiver were changed, or disappeared after this, it might cause inconsistency in the states of the caller when executing the i-codes in the receiver.

The situation is different in non-stateful computations. As they do not have hidden states, we need not to keep the dependency as in the case of stateful computations. Therefore, we allow non-stateful computations to be deleted freely, without considering the dependencies between computations. This implies that the principle of forward reference is applied to non-stateful computations but not stateful computations as we have mentioned in section 3.4.6.

Below is the data in relation *.comp_depend* for the above examples.

```
.comp_depend(.calling,   .called)
             --------    -------
             NewAcct     Acct
             Counter     FreqDiv
             TwoCounter  Counter
```

4.2.4 Data structures

For reasons of efficiency, we load the information in the system relations to memory whenever we start Relix. Below are the data structures used for storing the information (written in C with comments bracketed by /* and */) and their descriptions:

```
typedef struct _sym { /* corresponding to .comp and .comp_local */
  char          name[MAX_ID]; /* .dom_name or .var_name */
  short         type;         /* .type */
  short         status;       /* .seq_attr, not used in .comp_local */
  short         offset;       /* .dom_pos, not used in .comp_local */
  struct _sym   *next;        /* ptr to next element */
} SYM_TYPE;
```

The structure SYM_TYPE is used to store computation attributes or local variables in a single-linked list. Name and type are for the name and type of attributes or variables. Status is currently used to indicate the sequencing attribute if set to 1, otherwise 0. Offset indicates the position of attributes as well as their address of storage when it is being executed. *Next points to the next element in the list.

```
typedef struct _state { /* corresponding to .comp_state */
  unsigned short instance;      /* .instance */
  char           name[MAX_ID];  /* .var_name */
  short          type;          /* .type */
  ACT_VAL_TYPE   *init_val,     /* .init_val */
                 *actual_val;   /* .actual_val */
  struct _state  *next;         /* ptr to next stateful var */
} STATE_TYPE;
```

The structure STATE_TYPE is for storing stateful variables. It is a single-linked list like SYM_TYPE. *Init_val and *actual_val are for initial and actual values which are stored in the ASCII sortable format.

```
typedef struct _block { /* corresponding to .comp_type */
  long          btype;        /* .block_type */
  short         code_offset;  /* .code_offset */
  SYM_TYPE      *local;       /* ptr to a list of local var(s) */
  STATE_TYPE    *state;       /* ptr to a list of stateful var(s) */
  struct _block *next;        /* ptr to next block */
} BLOCK_TYPE;
```

The structure BLOCK_TYPE stores information related to blocks. Btype stores the type of the block while code_offset stores the offset of i-codes in the i-code file. *Local and *state are pointers to lists of local variables and stateful local variables respectively.

```
typedef struct _cmp { /* parent of the whole structure */
   char         name[MAX_ID];     /* .comp_name */
   SYM_TYPE     *fparam;          /* ptr to a list of comp attr(s) */
   STATE_TYPE   *state;           /* ptr to a list of stateful attr(s) */
   BLOCK_TYPE   *block;           /* ptr to a list of block(s) */
   boolean      completed;        /* compilation flag */
   struct _cmp  *next_in_bucket,  /* ptr to next comp in the bucket */
                *next_in_table;   /* ptr to next computation */
} COMP_TYPE;
```

The structure COMP_TYPE is the parent of the whole structures. Name is the name of computations. *Fparam and *state are pointers to lists of attributes and their states respectively. *Block points to a list of blocks. Completed is used in the compilation process and set to TRUE when the computation is successfully compiled.

A global hashing table and a global linked list are provided for computations. Every computation in Relix must have an entry in the hashing table and link to the linked list. When two or more computations have the same hash value, we solve the collision by forming them as a single-linked list with the pointer *next_in_bucket. *Next_in_table links all the computations in Relix together as a list. We will refer to the hashing table and the linked list as *computation pool* and *computation list* respectively.

The computation pool is used when searching for a computation by its name and the computation list for sequential tracking from the beginning of the list to the end when writing the information of all computations back to the system relations.

```
typedef struct _dp { /* corresponding to .comp_depend */
  char         calling[MAX_ID],/* .calling */
               called[MAX_ID]; /* .called */
  struct _dp   *next_calling,  /* pointer to the next caller */
               *prev_calling,  /* pointer to the previous caller */
               *next_called,   /* pointer to the next receiver */
               *prev_called;   /* pointer to the previous receiver */
} DEPEND_TYPE;
```

The structure DEPEND_TYPE stores the dependency information in the system relation *.comp_depend.* Calling and called are the names of callers and receivers respectively. *Next_calling and *prev_calling form a double-linked list on the dependencies of the same hashing key on their callers. So are *next_called and *prev_called on the receivers.

Two global hash tables are implemented, one for the callers, referred to as the *caller pool* and the other for the receivers, the *receiver pool*. We can search by hashing any receiver's name when we want to check before the deletion whether the receiver is called by other computations. Similarly, we can search by hashing the caller's name when we want to delete all the caller's dependencies, one of the tasks while the caller is being deleted.

The reason we implement the structure DEPEND_TYPE as double-linked lists instead of single-linked lists is, when we update a pointer in one list, we need the backward link of the other list to update the pointer in that list.

4.3 Implementation of operations

We implement operations on computations the same as the program flow we have mentioned at the end of section 4.1. Commands and statements are translated into sequences of intermediate codes. Then, the interpreter executes the intermediate codes. The algorithm for each operation is shown below:

- **create**

    1. If the name is already used, the operation fails.
    2. Create an entry in the relation pool (the hash table for relations).
    3. Create an entry in the computation pool.
    4. Link the entry to the computation list.
    5. Create a file with the same name as the computation, used as a source file.
    6. Create an i-code file.
    7. Perform the compilation of the intermediate codes (details in section 4.4).
    8. Write the generated intermediate codes into the i-code file.
    9. Write the source program into the source file.

- **show**

    1. If users do not supply the name then set pointer `ptr` to the beginning of the computation list, otherwise search an entry in the computation pool that contains the computaiton name.
    2. Set pointer `ptr` to that entry, and set flag `onlyone` to TRUE.
    3. Display the following information of the computation pointed by `ptr`
        - the name of the computation
        - the name, type, and sequencing status of the attributes
        - the computation type of each block
    4. If `ptr` is null or `onlyone`=TRUE then stop, otherwise move `ptr` to next entry and go to step 2.

- **print**

    1. Search an entry in the computation pool that has the same name as the computation.
    2. Print the contents of the file of the same name.

- **delete**

  1. If the name appears in the receiver pool then the deletion fails.
  2. Remove the entries that the name is the caller from the caller pool.
  3. Unlink the corresponding pointers in the receiver pool.
  4. Remove the entry from the relation pool.
  5. Delete the source file and the i-code file.
  6. Search for an entry with the same name in the computation pool.
  7. Free `*fparam`, `*state`, and `*block` associated to the entry.
  8. Set the name to null string.
  9. Unlink the entry from the computation list.
  10. Remove the entry from the computation pool.

- **assignment**

  Note that we call the names on the left and right hand side of the assignment operator as LHS and RHS respectively.

  1. Search RHS in the computation pool. If the name is not found then the assignment fails.
  2. Search LHS in the relation pool. If LHS is found (has been named to a relation or a computation) then the assignment fails.
  3. Create an entry of LHS in the computation pool.
  4. Link the entry to the computation list.
  5. Copy `*fparam`, `*state`, and `*block` from RHS to the ones in LHS.
  6. Copy the source file and i-code file from RHS to LHS.
  7. If RHS is a caller in the caller pool, generate the same dependency information with the name of RHS changes to the name of LHS.

- **instantiation**

  Instantiation does the same thing as the assignment but copying the initial values to the actual values in all stateful variables.

- **T-selector**

We implement the computation executor as a stack machine. There are three types of storage in the stack machine: Code Segment for storing code, Data Segment for storing data, and Stack Segment as the stack of the machine (details in section 4.5). T-selector is one of the two operations that use the executor (The other is the natural join).

Algorithm

1. Determine the computation type from input attributes.
2. Search the block corresponding to the computation type.
3. Save current status of the stack machine.
4. Load the intermediate codes of the block into Code Segment.
5. Allocate storage for all the local variables in Data Segment.
6. Create an entry for the output relation.
7. Set all storages in Data Segment to the uninitialized status (by setting the type to RELIX_VOID).
8. Load the value of the input attributes into Data Segment.
9. Perform the execution.
10. Append the output attributes from Data Segment as a tuple of the output relation.
11. Free storage of all the local variables in Data Segment.
12. Restore current status.
13. In the case where the output relation has more than one tuple (some may be generated by the i-code `mult-val` from the multi-valued expression), project the relation over its domains to eliminate the possible duplicate tuples.
14. Return the name of the output relation.

- **natural join**

  1. Determine the join attributes.
  2. If the computation is stateful then determine the sequencing attributes of the relation.
  3. Search the block corresponding to the type of the join attributes.
  4. Swap the sequencing attributes, if any, to be the first elements of the join attributes.
  5. Project the relation over the join attributes (by swapping the sequencing attributes to the first, the projection will implicitly sort the output relation on the sequencing attributes).
  6. Save current status of the stack machine.
  7. Load the intermediate code of the block into Code Segment.
  8. Allocate storage for all the local variables in Data Segment.
  9. Create an entry for the output relation
  10. Repeat step 11-17 until no more tuples.
  11. Fetch a tuple from the input relation.
  12. If the computation is stateful and the sequencing attribute is the same as the previous tuple then go to step 17.
  13. Set all storages in Data Segment to the uninitialized status (by setting the type to RELIX_VOID).
  14. Load the value from the tuple into Data Segment.
  15. Perform the execution.
  16. Append the output attributes from Data Segment as a tuple of the output relation.
  17. Move to next tuple.
  18. Free storage of all the local variables in Data Segment.
  19. Restore current status.

20. Project the output relation to eliminate the possible duplicate tuples.
21. U-join the projected output relation with the original input relation.
22. Return the output relation from the join operation.

4.4 The parser

When users declare a computation, Relix will check the syntax of the computation and perform semantic analysis. The syntax analyser is done by modifying the current grammar of Relix, which is generated by Lex [30] and Yacc [24], using the grammar we described in section 3.3 and 3.4. In this section we mainly discuss the semantic analyser of the parser.

The functions of the semantic analyser are:

1. Variable declaration—attributes, local variables, stateful variables.
2. Type checking—check the correctness of type of expressions.
3. Computation type inference—infer the computation type of each block from the expression within the block.
4. Generate intermediate codes and store in an i-code file for execution.
5. Determine the beginning position of intermediate codes of each block in the i-code file.
6. Determine the dependency of the caller and the receiver in the case of stateful computations.

If a computation is syntactically correct a sequence of intermediate codes is generated. We follow the principle of the *postfix* notation intermediate code in Relix [28]. For example, an expression

```
a <- b + c
```

will give a sequence of intermediate codes as follows:

```
push-name a
varaddr
push-name c
var
push-name b
var
plus
assign
```

From the above intermediate codes, we do the type checking, computation type inference, and generate a more compact version of intermediate codes to be kept in an i-code file for the purpose of execution (Within this section, we call this type of intermediate codes as *run-time i-codes* to be distinct from the ones created by the syntactic analyser).

Assume that *a* and *c* are attributes, *b* is a local variable. The run-time i-codes generated by the semantic analyser are:

```
push-addp a
push-para c
push-locl b
plus
assign
```

We augment six kinds of "push" to designate the type of the name after it. `Push-para` and `push-locl`, as in the above example, mean the following name is an attribute and a local variable respectively. `Push-spar` and `push-sloc` mean a stateful version of both types. `Push-addl` and `push-addp`, used in the assignment statement, tell the executor to push the address of the variable instead of value.

We omit the discussion of the type checking since we reuse the code for the type checking in domain algebra [28].

The computation type of blocks, which is represented as a long integer marking bits corresponding to the position of input attributes, is determined as follows:

- the computation type of the predicate clause is a bit string marking all positions of attributes (all attributes are considered input).

- the computation types of other blocks are iteratively inferred from the dependency (DP) value of variables in statements in the blocks. The algorithm is

  1. At the beginning of the block, set DP value of all attributes and local variables to RELIX_VOID (unused status).
  2. In each statement, set the DP value of the variable on the LHS of the assignment operator to the bit operation "or" of the weight, which is defined below, of all operands of the expression on the RHS.
     - The weight of any constant is 0.
     - The weight of any stateful variable is 0.
     - The weight of any attribute is $2^{(position)}$ when *position* is the order of the attribute in the computation if its DP value is RELIX_VOID or its DP value otherwise.
     - The weight of any local variable is its DP value. If the DP value is still RELIX_VOID then error.
  3. At the end of each statement, if there is any attribute that occurs on both LHS and RHS of the assignment operator then error.
  4. At the end of the block, do the bit operation "or" of all the DP values of all attributes. The result is the computation type of the block.
  5. If the computation is stateful, mark the bit of the computation type corresponding to the position of the sequencing attributes.

The examples below show how the above algorithm works. Note that we use the symbol '|' to represent the bit operator "or".

`comp Ex1(a,b,c) is`

`def a=b+c` $\Longleftarrow$ The type is $2^0 \mid 2^1 \mid 2^2 = 7$ (all attributes).

`a <- b+c;` $\Longleftarrow$ The DP value of a is 2^1 (position of `b`) $\mid 2^2$ (position of `c`) $= 6$.
The type is the DP value of a $= 6$.

`comp Ex2(a,b,c) is`	
`{`	
`local temp:intg;`	
`temp <- b+1;`	$\Leftarrow$ The DP value of **temp** is 2^1 (position of b)
	$\mid$ 0 (constant) = 2.
`a <- temp+c`	$\Leftarrow$ The DP value of a is 2 (the DP value of **temp**)
	$\mid$ 2^2 (position of c)
`  also b-c;`	$\mid$ 2^1 (position of b) $\mid$ 2^2 (position of c) = 6.
`};`	The type is the DP value of a = 6.

`comp Ex3(a:seq,b,c) is`	
`{`	
`b <- old b initial 0 - c;`	$\Leftarrow$ The DP value of b is 0 (stateful variable)
	$\mid$ 2^2 (position of c) = 4.
`};`	The type is 4 (the DP value of b)
	$\mid$ 2^0 (position of a, a sequencing attribute) = 5.

`comp Ex4(a,b,c) is`	
`{`	
`local temp:intg;`	
`a <- b+1;`	$\Leftarrow$ The DP value of a is 2^1 (position of b)
	$\mid$ 0 (constant) = 2.
`c <- temp+a;`	$\Leftarrow$ The DP value of c is the DP value of temp
`};`	(RELIX_VOID—unintialised local variable). Error!!!

`comp Ex5(a,b,c,d) is`	
`{`	
`a <- c*10;`	$\Leftarrow$ The DP value of a is 2^2 (position of c)
	$\mid$ 0 (constant) = 4.
`b <- d/2;`	$\Leftarrow$ The DP value of b is 2^3 (position of d)
	$\mid$ 0 (constant) = 8.
`};`	The type is 4 (the DP value of a)
	$\mid$ 8 (the DP value of b) = 12.

```
comp Ex6(a,b,c,d) is
{
a <- c**2;          <== The DP value of a is 2^2 (position of c)
                          | 0 (constant) = 4.
b <- a-d*2;         <== The DP value of b is 4 (the DP value of a)
                          | 2^3 (position of d)
                          | 0 (constant) = 8.
};                      The type is 4 (the DP value of a)
                          | 12 (the DP value of b) = 12.

comp Ex7(a,b) is
a <- a+b;       <== The DP value of a is 2^0 (position of a)
                      | 2^1 (position of b) = 3.
                    Error!!! a occurs in both LHS and RHS of
                      the assignment operator.
```

4.5 The executor

The executor is used for evaluating computations in the T-selector and the natural join operations. It comprises storages, an engine, and four pointers. There are three types of storages: Code Segment, Data Segment and Stack Segment. The four pointers are CS, DS, SP and IP. CS and DS point to the beginning of intermediate codes in Code Segment; and of attributes and local variables storage in Data Segment respectively. SP points to the top of stack in Stack Segment. IP points to the position of intermediate code which is being executed in Code Segment. When the executor starts, the engine sequentially fetches the intermediate code at the position where IP points, advances IP to the next intermediate code, performs the algorithm according to the intermediate code until it reaches the stop command.

4.5.1 The storages

Code Segment is implemented as an array of characters. Data Segment and Stack Segment are implemented as arrays of ACT_VAL_TYPE, the structure that keeps any type of data as a single element. The advantage is we do not have to deal with the problem of varying lengths of different types of data, i.e. 1 byte for boolean, 2 bytes for short, 4 bytes for long. The disadvantage is we pay the cost of spending more memory. The ACT_VAL_TYPE structure is written in C as below:[3]

```
typedef struct {
  short    v_type;    /* type of data */
  short    v_short;   /* storage for short */
  long     v_long;    /* storage for long */
  real     v_real;    /* storage for real */
  char     *v_string; /* pointer to string storage */
  short    v_buffer;  /* don't use */
  booleanv_bool;      /* storage for boolean */
} ACT_VAL_TYPE;
```

4.5.2 The engine

Here are the actions the engine performs for each i-code:

- `push-para` and `push-locl`
  - Read the next string from Code Segment.
  - Treat it as an attribute's or a local variable's name.
  - Find its address.
  - Push its value from Data Segment onto Stack Segment.
- `push-spar` and `push-sloc`
  - Read the next string from Code Segment.
  - Treat it as a stateful attribute's or a stateful local variable's name.
  - Find its actual value from the data dictionary.

[3]Actually, the structure has already been used in Relix for other purposes. We adapt it to use in the executor.

  - Push the value onto Stack Segment.

- **push-adrp** and **push-adrl**
  - Read the next string from Code Segment.
  - Treat it as an attribute's or a local variable's name.
  - Find its address and push onto Stack Segment.
- **push-name** (already used in Relix)
  - Read the next string from Code Segment.
  - Push the name as a string onto Stack Segment.
- **push-count** (already used in Relix)
  - Read the next string from Code Segment.
  - Translate to a short integer and push onto Stack Segment.
- unary operators, type casting, functions
  - Pop one value from Stack Segment.
  - Perform the evaluation according to the operator on the value.
  - Push the result onto Stack Segment.
- binary operators (comparison, logical, arithmetic, concatenation, maximum, minimum operators)
  - Pop two values from Stack Segment.
  - Perform the evaluation according to the operator on the values.
  - Push the result onto Stack Segment.
- ternary operators or if-then-else
  - Pop if-value from Stack Segment.
  - If if-value is TRUE, execute the i-code of then-clause and then skip the i-code of else-clause.

- Otherwise, skip the i-code of then-clause and execute the i-code of else-clause.

- constant operators

  When users use any constant value, Relix will generate a string form of that constant following by a constant operator. For example, a constant 3 will be translated into

  ```
  push-name -00003
  long
  ```

  A constant operator can be any of the following operators, **`boolean, short, long, real`** and **`string`**. The actions of the operators are:

  - Pop a constant in the form of character string from Stack Segment.
  - Perform the conversion of the constant to the type specifed by the operator.
  - Push the result back to Stack Segment.

- `assign`

  - Pop a value from Stack Segment.
  - Pop an address.
  - Set the value to the storage in Data Segment pointed by the address.

- `mult-val`

  This command is generated from the multi-valued expıession. Since we represent any variable as an element in Data Segment, it is impossible that a variable can hold multiple values. What we do is write the value to the output relation after the evaluation of each value. This method of implementation leads to a few limitations on the multi-valued expression:-

  1. No local variable can be assigned from any multi-valued expression.
  2. Only one multi-valued expression is allowed in any block and must be in the last statement of the block.

3. The computation with multi-valued expression cannot be called inside other computations. It must be executed at the Relix prompt only.
4. No multi-valued expression is used in the conditional expression (if-then-else).

The actions are:

- Pop a value from Stack Segment.
- Pop an address.
- Set the value to the storage in Data Segment pointed by the address.
- Push the address back to Stack Segment.
- If the execution is not from Relix, error.
- If the address refers to a local variable, error.
- Otherwise, create a tuple of all the output attributes and append to the output relation.

For example, the statement

```
x <- sqrt(y) also -sqrt(y);
```

will have the intermediate codes as

```
push-adrp x
push-para y
sqrt
mult-val        ⟸ Write the value to the storage in Data Segment;
push-para y       push the address back;
sqrt              write all the output attributes as a tuple.
unary-minus
assign          ⟸ Write the value to the storage in Data Segment.
halt            ⟸ Write the output attributes as another tuple.
```

- `filler` (used in right-partial-fit to indicate an output attribute)
  - Push an unintialised data onto Stack Segment.
- `new`
  - Set the "new" flag to TRUE (to be used in right-partial-fit).
- `right-partial-fit`

  This command indicates calling a computation. The actions are:
  - Pop a short integer indicating the numbe of attributes.
  - Pop values as many times as indicated by the number of attributes. Assign the values to the corresponding addresses in Data Segment.
  - Pop the called computation's name.
  - Save current status by pushing the CS, DS and IP pointers onto Stack Segment.
  - Set CS to the end of current code, DS to the end of current data, and IP to CS.
  - Load the called computation's intermediate codes.
  - Perform the execution on the intermediate codes.
  - Restore previous status by popping the old IP, DS and CS back from Stack Segment.
  - Push the result onto Stack Segment.

  The different between the execution when "new" flag is TRUE and FALSE is all the stateful variables that are referred to within the execution are the calling computation's when the flag is TRUE and are the called computation's otherwise. [4]

  We do not support nested calling. For example,

  ```
  a <- COMP1{..., COMP2{...} };
  ```

[4]As in examples in section 3.5.6

is not allowed. Alternatively we create a local variable to hold the result of COMP2 and then use the variable as a parameter in COMP1.

```
temp <- COMP2{...};
a <- COMP1{..., temp };
```

- **halt**
  - Update stateful variables from its non-stateful variables of the same name.
  - if the execution is from Relix prompt, write the output attributes to the output relation.
  - Stop the execution.

4.6 Concurrency control

Relix was designed to be a multi-user database system. Therefore, concurrency control was one of the functions that Relix encountered. Computations, to be built as parts of Relix, certainly have to follow its scheme.

When a user starts Relix, data dictionary in the form of system relations are copied into data structures in memory. At the same time, if another user starts Relix, another copy of the data dictionary is brought up. To control the consistency of several copies of data dictionary in memory and the system relations, which are ordinary UNIX files, Relix propagates changes in data dictionary to the *differential files.* Every time any Relix process updates the data dictionary it writes the updating contents into the differential files. Before executing a command, every Relix process checks the differential files and updates its own data dictionary in memory from the differential files. To cover the concurrency control of computations, the modification of the original differential files is done to support the change of the data dictionary of computations.

In any critical section, like writing to the differential files, Relix implemented a semaphore as a UNIX file. Any Relix process that wants to get into a critical section has to delete the semaphore file corresponding to the critical section and create the same file back when it leaves.

The problem of locking is also considered. There are two kinds of locking—exculsive lock (X-lock) and shared lock (S-lock). Exclusive lock on any object guarantees that no other process can access the object except the one which has locked the object. Shared lock allows other processes to access but not update the object. For example, in Relix, the relation being deleted is locked exclusively so that no other process can manipulate the relation. Relix implemented the locking mechanism using a write queue and a read queue.

To request an exclusive lock on any object:

- Repeat waiting until the object is not in either the write or read queue.
- Put the object in the write queue.

To request a shared lock on any object:

- Repeat waiting until the object is not in the write queue.
- Put the object in the read queue.

To release an exclusive or shared lock:

- Take the object out of the write or read queue.

To implement the locking mechanism for computations, we reuse the one in Relix since computations are special cases of relations. However, we create a new set of read and write queues for stateful variables. The locking of these variables is performed separately since, in the same operation, they may require the type of locking different from the computation they belong to. Table 4.3 shows the type of locking required in each operation.

We also implement the concurrency control of data from relation *.comp_depend* (dependency information between stateful computations). As mentioned before, the data is loaded into two global hash tables, caller pool and receiver pool. We consider these two pools as a single critical section using the semaphore mechanism. Only one process can enter the critical section at a time. We also propagate the changes of dependency information to a differential file. For example, after the deletion or creation of any stateful computation, we write the changes of the dependency information, if any, to the differential file. Then, other Relix processes will update their own dependency information (in memory) from the differential file.

Operation	**Type of locking**
Declaration	X-lock
Print	S-lock
Delete	X-lock
Assignment/ Instantiation	X-lock on the new computation, S-lock on the computation
T-selector	
non-stateful	S-lock
stateful	S-lock on the computation but X-lock on all stateful variables
Join	
non-stateful	S-lock
stateful	S-lock on the computation but X-lock on all stateful variables

Table 4.3: Types of locking on operations

Chapter 5

Conclusions

5.1 Conclusions

Computations have been built to represent procedural abstraction in Relix. They are special forms of relations. Operations on relations can apply to computations as well. Two of the operations in relational algebra [36], T-selector and natural join, have been implemented.

With the T-selector operator, we supply values for a subset of attributes and get the complement of the attributes as a result. The natural join between a computation and a relation is thought of as supplying a set of tuples from the relation for input attributes of the computation. Each tuple is equivalent to each set of values we supply for in the T-selector. The result of the natural join is a set of tuples in which each is equivalent to the output tuple returns from the T-selector.

Furthermore, we introduce hidden states in computations. They are inaccessible states associated with attributes or local variables, which are automatically updated from their counterpart at the end of the execution of computations. The mechanism of states allows us to remember previous states and reuse them in the next round of executions, for example, the balance in a bank account, or previous solutions of the iteration method in solving linear equation.

Stateful computations can also be instantiated to give new stateful computations with all states reset to their initial values. These states are independent of original computations' states.

In the natural join between stateful computations and relations, we found that the order of tuples in the relation affected the result of states. Therefore, we introduced *sequencing attributes* to impose an order on tuples. Every stateful computation must have some attributes which serve as sequencing attributes to designate the order.

This work is an experimental system of building computational objects in a relational database system. We believe that relational database systems can be extended to cover the problem of computational incompleteness, one of many points that are claimed as a weakness in [4].

The strong point of the relational algebra is it gives a higher abstraction of operations on data. When we use the relation algebra with computations, we retain this property without losing the power of the operations.

5.2 Future work

There are many features to be further explored in the context of computations. Some of them are presented here.

5.2.1 Natural join between computations

Suppose we have two computations, *C1(a:intg,b:intg,c:intg)* and *C2(d:intg,e:intg,b:intg)*. The computation types of *C1* and *C2* are:

$$(a{:}intg \times b{:}intg) \rightarrow c{:}intg$$

and

$$(d{:}intg \times e{:}intg) \rightarrow b{:}intg$$

respectively. *C1 natjoin C2* might have the attributes *(a:intg,b:intg,c:intg,d:intg,e:intg)* and the following computation type:

$$\begin{aligned}&(a{:}intg \times b{:}intg) \rightarrow c{:}intg \mid \\ &(d{:}intg \times e{:}intg) \rightarrow b{:}intg \mid \\ &(a{:}intg \times d{:}intg \times e{:}intg) \rightarrow c{:}intg\end{aligned}$$

The last part of the above type comes from the fact that we can derive the attribute *b* from the attributes *d* and *e*. That means, in the new computation resulting from *C1 natjoin C2*, we can input either *a* and *b*; or *d* and *e*; or *a,d* and *e* to get the rest of the attributes as the output.

This interpretation seems more complicate when either *C1* or *C2* is the union of more than one computation type. For example, suppose the computation type of *C1* has been changed to:

(a:intg × *b:intg)* ⟶ *c:intg* |
(a:intg × *c:intg)* ⟶ *b:intg*

Can we do the substitution of the first part of the computation type of *C1* with the computation type of *C2* and leave the second part as it is? Then the computation type of *C1 natjoin C2* will be:

(a:intg × *b:intg)* ⟶ *c:intg* |
(d:intg × *e:intg)* ⟶ *b:intg* |
(a:intg × *c:intg)* ⟶ *b:intg* |
(a:intg × *d:intg* × *e:intg)* ⟶ *c:intg*

Moreover, how do we interpret the natural join of two stateful computations? Here is one alternative.

Suppose there are hidden states in the joining computations.

- If no state at the common attributes, then copy all states to the result computation.

- If there are states at the join attributes and the values of all states are equal, copy those values into the result computation.

- If there are states at the join attributes but the value of any state is not equal, the operation fails.

5.2.2 Other join operations

We may think of the natural composition between two computations as a functional composition. If *f* and *g* are functions that $f: a \rightarrow b$ and $g: c \rightarrow d$ then the functional composition $g \circ f : a \rightarrow d$ is applicable if and only if the domain b is equal to the domain c. For example, the computations *C1* and *C2* are:

```
comp C1(a,b,c) is  c <- a+b;
comp C2(d,e,b) is  b <- d*e;
```

The computation *C3*, which is *C1 icomp C2*, will be:

```
comp C3(a,c,d,e)  is
{
local b:intg;
b <- d*e;
c <- a+b;
}
```

Then, the computation type of *C3* will be:

$$(a{:}intg \times d{:}intg \times e{:}intg) \rightarrow c{:}intg$$

We have not yet found any useful interpretation for other join operations, such as union join, different join, or other type of σ-joins in the context of computations. However, Merrett did give some interesting interpretations of these operators in [37].

5.2.3 Successive mode in stateful computations

As we have mentioned in chapters 3 and 4, states in stateful computations can be updated in two modes: simultaneous mode, which has been implemented, and successive mode. States are updated simultaneously from their non-stateful variables of the same names at the end of the evaluation of computations in the simultaneous mode. Alternatively, in the successive mode, states are updated immediately at their change after the execution of assignment statements.

An implementation of the successive mode may be done at two levels:

1. At the syntactic level, we provide a new command to toggle the running mode of stateful computations, e.g, `sim` and `suc` for the simultaneous and successive mode respectively. We may also set the simultaneous mode as the default mode.

2. At the implementation level, instead of updating all stateful variables at the i-code `halt`,[1] we update the state variable of the changed value at the i-code `assign`.

5.2.4 Shared states in computations

Suppose we have a computation which represents a storage or a memory cell. A stateful variable is used in the computation to be the storage to keep a value, say, an integer. There are two operations on this computation: read the value in the storage and write a value to the storage. This introduces a problem of sharing a stateful variable by two operations, which cannot be solved by simply declaring two stateful computations for the operations, *read* and *write*.

We may solve the above problem by extending computations as follows:

1. Computations are treated as ordinary type. a computation can be passed as a parameter.

2. Stateful variables can be explicitly declared.

Then, the storage may be written as below:

```
domain i integer;
domain c comp;
comp storage(i, c) is
{
local old i initial 0 :  intg; << declare a stateful var >>
old i <- c{i}
alt
i <- c{old i};
}
comp write(j,i) is
j <- i;
comp read(i,j) is
i <- j;
```

[1]See section 4.5.2

When we supply a value and a computation name for the attributes `i` and `c` respectively, the computation `storage` use the block

```
old i <- c{i}
```

From the above statement, we can infer that the type of computation `c` is

$$intg \rightarrow intg$$

Then, the type of the block is

$$(i{:}intg \times c{:}(intg \rightarrow intg) \rightarrow \phi)$$

The symbol ϕ represents no returning value because we keep the value of `old i` within the computation.

In the same manner, the block

```
i <- c{old i}
```

will have the type

$$c{:}(intg \rightarrow intg) \rightarrow i{:}intg$$

To write a value, say 10, to a storage, we type

```
st1 <- new storage{10,"write"};
```

To read the value of the storage `st1`, we type

```
i <- st1{,"read"};
```

5.2.5 Array of instances

Suppose we need to instantiate a hundred of bank account from the computation `Accum`[2] at the same time. It is a tedious task to do them one by one. A way to do that is to have computations as an attribute in a relation. The relation may have other attributes used as an identifier of a computation in each tuple.

For example, a statement

[2]See page 32

```
Acct_arr <- new Accum on acct_no[101..200];
```

may generate a relation Acct_arr that has two attributes: (acct_no:intg, Accum:comp) with each tuple is an instance of the computation Accum identified by the value 101 to 200 on the attribute acct_no.

Suppose we deposit $1000 to a particular account, say acct_no 150, it can be done by typing

```
bal <- Acct_arr{150}{0,1000};
```

In fact, we may have more than one identifier attribute and the values on identifier attributes may not be continuous. For example, we may create a relation of 6 instances of computation by the following syntax:

```
inst_arr <- new StateComp on Id1[1..3], Id2[4,89];
```

In this case, the identifiers will be the combination of the values from the attributes Id1 and Id2, i.e., (1,4), (2,4), (3,4), (1,89), (2,89), and (3,89).

Bibliography

[1] ________. Fundamentals of Object-Oriented Databases. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1–32. Morgan Kaufmann, San Mateo, CA, 1990.

[2] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1990.

[4] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The story of O_2.* Morgan Kaufmann, San Mateo, CA, 1992.

[5] P. Butterworth, A. Otis, and J Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10), October 1991.

[6] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, CA, 1990

[7] R.G.G. Cattell. *Object Data Management, Object-Oriented and Extended Relational Database Systems (Revised Edition).* Addison-Wesley, Reading, MA, 1994.

[8] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, New York, 1984.

[9] P.P. Chen. The Entity-Relationship Model Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), March 1976.

[10] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377-87, June 1970.

[11] E.F. Codd. Relational Completeness of Data Base Sublanguages. *Data Base Systems, Courant Computer Science Symposium 6*, 1972.

[12] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4), December 1979.

[13] E.F. Codd. Relational Database, A Practical Foundation for Productivity. *Communications of the ACM*, 25(2):109 17, February 1982.

[14] E.F. Codd. *The Relational Model for Database Management: version 2* Addison-Wesley, Reading, MA, 1990.

[15] C.J. Date. *An introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, fourth edition, 1986.

[16] C.J. Date. *An introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, fifth edition, 1990.

[17] C.J. Date and H. Darwen. *Relational Database Writings 1989-1991*. Addison-Wesley, Reading, MA, 1992.

[18] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Data Base System. *Communications of the ACM*, 19(11), November 1976.

[19] R. Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems*, 2(3), September 1977.

[20] R. Fagin. Normal Forms and Relational Database Operators. In *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, 1979.

[21] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngback, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Chan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[22] C E. Fröberg. *Numerical Mathematics· Theory and Computer Applications.* Benjamin/Cummings Publishing, Menlo Park, CA, 1985.

[23] D.K. Hsiao, editor. *Advanced Database Machine Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[24] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[25] S N. Khoshafian and G.P. Copeland. Object Identity. In *OOPSLA '86 Conference Proceedings*, Portland, OR, September 1986.

[26] W. Kim, D.S. Reiner, and D.S. Batory, editors. *Query Processing in Database Systems.* Springer-Verlag, New York, 1985.

[27] H.T. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[28] N Laliberté. Design and implementation of a primary memory version of Aldat, including recursive relations. Master's thesis, McGill University, Montreal, 1986.

[29] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10), October 1991.

[30] M.E. Lesk. Lex: a Lexical Analyzer Generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[31] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[32] A.P. Malvino and D.P. Leach. *Digital Principles and Applications.* McGraw-Hill, New York, fourth edition, 1986.

[33] T.H. Merrett. MRDS: An Algebraic Relational Database System. In *Canadian Computer Conference*, May 1976.

[34] T.H. Merrett. Relations as Programming Language Elements. *Information Processing Letters*, 6(1):29–33, Feb 1977.

[35] T.H. Merrett. The Extended Relational Algebra, a Basis for Query Languages. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*. Academic Press, New York, 1978.

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

[37] T.H. Merrett. Computations: Constraint Programming with the Relational Algebra. In *International Symposium on Next Generation Database Systems and their applications*, Fukuoka, Japan, September 1993.

[38] T.H. Merrett and N. Laliberte. Including Scalars in a Programming Language Based on the Relational Algebra. *IEEE Transactions on Software Engineering*, 15(11):1437–43, November 1989.

[39] T.H. Merrett and E. Munshkin. Common Conceptual Foundations of Object-Oriented and Relational Databases. (Unpublished), March 1992.

[40] A. Pidcock. FSL: A language for constraint programming with booleans and reals. Master's thesis, McGill University, Montreal, 1993.

[41] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10), October 1991.

[42] M. Stonebraker and L.A. Rowe. The Design of POSTGRES. In *Proceedings of ACM SIGMOD'86 International Conference on Management of Data*. Association for Computing Machinery, May 1986.

[43] G.J. Sussman and G.L. Steele Jr. CONSTRAINTS – A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14, 1980.

[44] T.J Teorey and J.P. Fry *Design of Database Structures.* Prentice-Hall, Englewood Cliffs, NJ. 1982

[45] S.J.P Todd. The Peterlee Relation Test Vehicle– a system overview. *IBM System Journal*, 15(4), 1976.

[46] J.D Ullman *Principles of Database Systems.* Computer Science Press, Rockville, MD, second edition, 1982.

[47] Versant Object Technology. *Versant Reference Manual.* Versant Object Technology Inc., Menlo Park, CA, 1990.

[48] S.B. Yao. Optimization of Query Evaluation Algorithms. *ACM Transactions on Database Systems*, 4(2), June 1979.

[49] S.B Zdonik and M. Hornick. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), January 1987