Language Support for A Relation ADT for Clifford Algebra

Rong Li

School of Computer Science
McGill University, Montreal

December 2005

A Project Report Submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science in Computer Science

Abstract

This project report describes the implementation of an Abstract Data Type (ADT) *CliffordADT* for Clifford algebra which supports addition, subtraction and production operations of Clifford algebra. It also describes the new features added in JRelix system to support the creation of this ADT, including grouping by nested relation domain in equivalence reduction, ordering by nested relation domain in functional mapping, as well as grouping and ordering by nested relation domain in partial functional mapping. In addition, three Boolean functions *isnulldc*, *isnulldk* and *isnull* are implemented to test *dc* or *dk* values of an attribute. Background knowledge is presented to make this project report readable, and the user's manuals are provided to illustrate the usage of the *CliffordADT* and the new features added in the system. This project is part of the Aldat project at McGill University.

Acknowledgements

I would like to acknowledge all those who make this project possible through their support and help. First, I would like to express my deepest gratitude and respect to my supervisor **Professor T. H. Merrett** for his guidance, advice, and encouragement, as well as his insightful help and supervising which is crucial for this project. I highly appreciate his generous financial support, very patient explanation, careful proofreading of this report, and the time he spent with me on this project.

I am indebted to Ms. Zongyan Wang for her valuable advices and kindly help. My appreciation also goes to the School of Computer Science for its research environment.

Finally, I would like to thank my husband, ZhiFeng Huang, who gives me support for all these years.

Contents

Chapter 1

Introduction

In this project report, both the implementation of *CliffordADT* and the new features introduced in JRelix to support the creation of this *ADT* are described. Section 1.1 gives the background and motivation of this project, and in Section 1.2, project report outline will be provided.

1.1 Background and Motivation

JRelix system, redesigned from Relix since 1997, is a Database Management System (DBMS) based on Aldat language [Hao98,Yua98,Bak98]. This system is developed in Java environment and uses Object-oriented structure, which enables flexible implementation and multi-platform support.

The first goal of this project, which is one of the JRelix implementation projects, is to implement an Abstract Data Type (ADT) to provide support of Clifford algebra operations in JRelix. Clifford Algebra is a type of associative algebra in mathematics. It provides a complete coordinate-free representation of geometric notation of direction and magnitude. Clifford Algebra is widely used in different fields [AbF00] including geometry, theoretical physics, engineering, etc., and leads to large amount of useful applications.

In the current JRelix system, when grouping or ordering by nested relation domain in vertical operations such as equivalence reduction, functional mapping and partial functional mapping, the system will group or order tuples according to the surrogates instead of the real values of these nested domains. Therefore, another goal of the project is to extend the system so that the real values of the nested relation domain are used in grouping or ordering tuples.

Also, the current system is unable to test whether the value of an attribute is *dc* or *dk* value. Thus, Boolean functions need to be implemented to detect these values, serving as another task of this project.

1.2 Project Report Outline

This project report is organized as follows. Given the topic of this project presented in this chapter, the related background knowledge is provided in the next Chapter 2. In Chapter 3, user's manual for the new features is described. The implementations of these new features, as well as the implementation and usage of *CliffordADT* are illustrated in Chapter 4 and Chapter 5, respectively. Finally, a brief summary is given in Chapter 6.

Chapter 2

Background

The purpose of this chapter is to introduce the required background knowledge to readers for helping them understand the rest of the project report. In Section 2.1 the usage of JRelix systems will be described, and outline of the implementation of JRelix will be given in Section 2.2.

2.1 Introduction to JRelix

2.1.1 Getting Started

JRelix system is a database engine running on any platform that has Java Runtime Environment 1.1 or up. To start JRelix, typing the following in the command line if it is in the same directory as where the classes files of JRelix locates:

java JRelix

or if it is in any other directory, providing classpath as following:

java –classpath [classpath] JRelix

If JRelix starts successfully, the following screen will appear and > is shown to prompt user inputs:

```
Starting stand alone JRelix.
+-----------------------------------------+
|            Relix Java version 0.93      |
| Copyright (c) 1997 -- 2004 Aldat Lab    |
|         School of Computer Science      |
|             McGill University           |
+-----------------------------------------+
>
```

2.1.2 Declaration

Domain Declaration

Actual domains are declared in JRelix through the use of keyword "domain", that is [Zhe02],

"domain" IDList Type *";"*

where IDList specifies the list of the domains being declared, and the types of these domains are presented in *Type*. There are two domain types in JRelix: One is atomic and the other is complex. The types of atomic domain include string, Boolean, short, integer, etc., as shown in Figure 2.1, while the types of complex domain include nested relation and computation.

Type	Short Form
integer	intr
long	long
short	short
float	float
double	double
string	strg
boolean	bool
universal	univ
numeric	num
attribute	attr

Figure 2.1: Types of atomic domain [Yu04]

The following syntax is used to declare nested relation domain. The attributes of the declared nested domain are listed in IDList. Note that the attributes in the IDList must be declared before the declaration of the nested domain [Yua98]:

domain nested_domain_name (IDList) ";"

The syntax for declaring computation is shown as below. Parameters of the declared computation are listed in IDList. Similarly, the parameters in the IDList must be declared before the declaration of the computation [Bak98]

domain computation_name **comp** (IDList) ";"

Examples for declaring domains in JRelix are illustrated in Figure 2.2.

```
>domain coeff float;
>domain index intg;
>domain cliff (index);
>domain cliffordL (coeff, cliff);
>domain cliffordR (coeff, cliff);
>domain clifford (coeff, cliff);
>domain Add comp(cliffordL,cliffordR, clifford);
```

Figure 2.2: Example of domain declarations

Relation Declaration & Initialization

The Syntax for relation declaration and initialization is shown as:

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

where the first IDList specifies the declared relations, and the second IDList specifies the attributes of the declared relation. If the "Initialization" is absent, an empty relation will be created without any tuple inside; Otherwise, it will be declared with the actual tuples. In the relation initialization, the curly brackets "{" and "}" are used to indicate the start and end of the initialization, and the data in the same tuple are surrounded by round bracket "(" and ")". In the case that a nested relation is declared and initialized, the surrogates are stored and used to link the actual values of the nested relation attribute which are stored in a relation with name "."+nested relation attribute's name.

An example for relation declaration and initialization is shown in Figure 2.3.

```
domain salePerson strg;
domain saleAmount intg;
domain department intg;
domain company strg;
domain productName strg;
domain product(company, productName);
relation SaleInfo(product, salePerson, department, saleAmount) <-
{({("IBM", "Thinkpad T43P"),("DELL", "Inspiron 5150")},"Smith", 1,
10000),
({("Sony", "Hi 8 Camcorder"), ("Kodak", "Digital Camera")}, "Jones", 1,
7800),
({("IBM", "Thinkpad T43P"),("DELL", "Inspiron 5150")}, "Brown", 2,
6900),
({("IBM", "Thinkpad T43P"),("DELL", "Inspiron 5150")}, "Larry", 2, 3400)
};
```

Figure 2.3: Example of relation declaration & initialization

In this example, nested relation *SaleInfo* with a nested relation domain product as its attribute is declared and initialized. After initialization, values of *SaleInfo* are stored in the system as shown in Figure 2.4. Note that the values of product shown in *SaleInfo* are surrogates. The real values of product is stored in relation *".product"*, and the function of *".id"* in relation *".product"* is to link surrogates to the real values.

```
>pr SaleInfo;
+----------------------+----------------------+-------------+-------------+
| product              | salePerson           | department  | saleAmount  |
+----------------------+----------------------+-------------+-------------+
| 3                    | Brown                | 2           | 6900        |
| 4                    | Larry                | 2           | 3400        |
| 1                    | Smith                | 1           | 10000       |
| 2                    | Jones                | 1           | 7800        |
+----------------------+----------------------+-------------+-------------+

>pr .product;
+----------------------+----------------------+----------------------+
| .id                  | company              | productName          |
+----------------------+----------------------+----------------------+
| 1                    | DELL                 | Inspiron 5150        |
| 1                    | IBM                  | Thinkpad T43P        |
| 2                    | Kodak                | Digital Camera       |
| 2                    | Sony                 | Hi 8 Camcorder       |
| 3                    | DELL                 | Inspiron 5150        |
| 3                    | IBM                  | Thinkpad T43P        |
| 4                    | DELL                 | Inspiron 5150        |
| 4                    | IBM                  | Thinkpad T43P        |
+----------------------+----------------------+----------------------+
```

Figure 2.4: Contents of Relation *SaleInfo*

2.1.3 Assignment

There are two assignment operators in JRelix System. One is "<-", which copies contents and attributes of the operand on the right side of an operator to the operand on the left side of the operator. The other is "<+", which appends the content of the right side operand to the left side operand. An example of using "<-" is shown in Figure 2.5. For examples of "<+", please refer to [Zhe02] for details.

```
cliffordR' <- [coeffR, cliffR] in cliffordR;
cliffordR <- where coeff != 0.0 in ([coeff, cliff] in cliffordR');
```

Figure 2.5: Example of assignment operator "<-"

2.1.4 Relation Algebra

In this section, some of the unary and binary operations implemented in JRelix will be briefly described.

Unary operations

As its name indicates, unary operations take one operand. There are six unary operations implemented in JRelix system, including *projection*, *selection*, *T-selection*, *QT-selections*, etc. Here *projection* and *selection* will be described in details. Details for the other operations can be found in [Mer84].

Projection

Project extracts a specified subset of attributes from the operand. The syntax of this operation is [Hao98]:

"[" (IDList)? "]" in (Projection | Selection)

Here, the list of attributes that need to be projected from the operand is specified in "IDList". If "IDList" is empty, a relation with one tuple of Boolean value will be returned. The Boolean value could be "true" or "false" depending on whether the operand relation is empty or not. An example is shown in Figure 2.6, where relation "SaleInfo" defined in Figure 2.3 is used.

```
> sAmount <- [saleAmount] in SaleInfo;
>pr sAmount;

+--------------+
| saleAmount   |
+--------------+
| 3400         |
| 6900         |
| 7800         |
| 10000        |
+--------------+

>pr [] in SaleInfo;

+---------+
| .bool   |
+---------+
| true    |
+---------+
```

Figure 2.6: Example of Unary Operation: Projection in JRelix

Selection

Selection operation extracts from the operand relation certain tuples which satisfy the specified condition. The syntax of selection operation is shown as below [Hao98]:

where [condition clause] in projection

where "condition clause" could be any expression which returns Boolean value for each tuple. An example of selection operation is illustrated in Figure 2.7, with relation *SaleInfo* being used.

```
>superSale <- where saleAmount > 7000 in [saleAmount,
              salePerson] in SaleInfo;

>pr superSale;

+-------------+----------------------+
| saleAmount  | salePerson           |
+-------------+----------------------+
| 7800        | Jones                |
| 10000       | Smith                |
+-------------+----------------------+
```

Figure 2.7: Examples of Unary Operation: Selection in JRelix

Binary Operations

Binary operations in JRelix fall in the following two categories: μ-join and σ-join. All these binary operations are set operations and satisfy closure, which means that if two operands are relations, the result of the binary operation is also a relation. The syntax of these join operators is [Hao98]:

Expression JoinOperator Expression

Or

Expression "["ExprList": "JoinOperator":"ExprList"]" Expression.

12 σ-join operators are implemented in JRelix and used to generalize logical operations, with definitions given in Ref. [Mer84]. Also, there are seven operators (as shown in Figure 2.8) belonging to μ-join operation, including *ijoin*, *ujoin*, *ljoin*, *rjoin*,*djoin*, *drjoin*, *sjoin*. If we define **center**, **left** and **right** as below [Mer84]

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

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

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

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

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

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

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

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

we can obtain the definition of these μ-joins as those in Figure 2.8:

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

Figure 2.8: Definition of μ-joins operators [YiZheng 2004]

The usage of *ujoin* operator is illustrated in Figure 2.9. Two relations are used in this example. One is *ProductAvaliable,* which stores the information about the available amount of certain product. The other relation *ProductPrice* contains the prices of products. The names of products are stored in different attributes in these two relations, that is, in attribute *product* of relation *ProductAvaliable*, and in attribute *item* of *ProductPrice.* To combine all the information in these two relations, union of the two relations on their common attributes *product* and *item* is used to create a new relation

ProductInfo. Note that the value of *price* for product "*Inspiron 5150*", which is not in relation *ProductPrice*, is given *dc* value in the *ProductInfo*.

```
>domain product strg;
>domain amount intg;
>domain item strg;
>domain price intg;

>relation ProductAvaliable (product, amount) <- {("Thinkpad T43P", 20),
                                                 ("Hi 8 Camcorder",50),
                                                 ("Inspiron 5150", 10)};

>pr ProductAvaliable;

+----------------------+-------------+
| product              | amount      |
+----------------------+-------------+
| Hi 8 Camcorder       | 50          |
| Inspiron 5150        | 10          |
| Thinkpad T43P        | 20          |
+----------------------+-------------+

>relation ProductPrice (item, price) <- {("Thinkpad T43P", 2500),
                                         ("Hi 8 Camcorder", 980)};

>pr ProductPrice;

+----------------------+-------------+
| item                 | price       |
+----------------------+-------------+
| Hi 8 Camcorder       | 980         |
| Thinkpad T43P        | 2500        |
+----------------------+-------------+

>ProductInfo <- ProductAvaliable[product:ujoin:item]ProductPrice;

>pr ProductInfo;

+----------------------+-------------+----------------------+-------------+
| product              | amount      | item                 | price       |
+----------------------+-------------+----------------------+-------------+
| Hi 8 Camcorder       | 50          | Hi 8 Camcorder       | 980         |
| Inspiron 5150        | 10          | Inspiron 5150        | dc          |
| Thinkpad T43P        | 20          | Thinkpad T43P        | 2500        |
+----------------------+-------------+----------------------+-------------+
```

Figure 2.9: Example of μ-joins operator: *ujoin*

2.15 Domain Algebra

Domain algebra is an algebra on attributes. There are two main components in Domain algebra: Scalar operations and Aggregation operations. Scalar operations allow arithmetic, logic and string processes on attributes within each tuple. Therefore, they could also be referred as Horizontal operations. On the other hand, Aggregation operations work vertically on all tuples in a relation, and thus they are also called Vertical operations.

Scalar operations

Scalar operations could be used in defining constants, renaming attributes, performing arithmetical and logical operations on attributes, as illustrated in Figure 2.10.

Defining constants:

```
>let one be 1;
```

Rename attributes:

```
>let coeffR be coeff;
>let cliffR be cliff;
```

Performing arithmetical operation on attributes:

```
>let evodsjoin be evodsjoin' mod 2
>let d3 be seqR - seqLi;
```

Performing logical operation on attributes:
(condition statement if-then-else)

```
>let tempd2 be if seqdiff < 0 then 0 else seqdiff;
```

Figure 2.10: Examples of Scalar operations in Domain Algebra

Aggregate Operations

Aggregate Operations include the following four operations: Reduction, Equivalence reduction, Function mapping and Partial function mapping.

Reduction

The example below is used to illustrate the meaning of Reduction operation:

> *let saleTotal be red + of saleAmount;*

which will sum up all the values of *saleAmount*. Other built-in operations, including *, *min*, *max*, *and*, *or*, *nop*, *ijoin*, *ujoin*, *sjoin*, could also be used in red reduction. The last three operations are for relations, and operation *nop* is for both primary typed domain and relation domain. The rest six operations are for primary typed domain [Yua98].

Reduction operations could also be used as below:

> *let count be red + of 1;*

which will count the number of tuples of the relation which *count* is projected from.

> *let avgSale be (red + of saleAmount)/(red + of 1);*

which is the combination of two red reductions (Aggregate operations) with division (Scalar operation). The first *red* reduction calculates the sum of *saleAmount* and the second *red* reduction counts the total number of person; the division gives the average sale amount.

Equivalence Reduction

Equivalence reduction allows reduction to be performed to groups of tuples within a relation [Mer84]. Whether tuples are in the same group (i.e., they are equivalent) or not depends on whether they have the same value for a specified set of domains. An example

for using this operation is illustrated in Figure 2.11. Relation *SaleInfo* defined in Figure 2.4 is used in this example, and the values of *equivSum* are aggregated inside each group.

```
>let equivSum be equiv + of saleAmount by product;
>equivRel <- [product, saleAmount, equivSum] in SaleInfo;
>pr equivRel;

+----------------------+-------------+-------------+
| product              | saleAmount  | equivSum    |
+----------------------+-------------+-------------+
| 3                    | 3400        | 20300       |
| 3                    | 6900        | 20300       |
| 3                    | 10000       | 20300       |
| 2                    | 7800        | 7800        |
+----------------------+-------------+-------------+
```

Figure 2.11: Example of Equivalence reduction

Functional Mapping

Functional Mapping is used to introduce order into vertical operation and perform calculation that Reduction could not perform. For example, to rank the sale persons based on how much they sale could only be done in Function mapping, as shown in Figure 2.12.

```
>let rank be fun + of 1 order saleAmount;

>pr [salePerson, saleAmount, rank] in SaleInfo;

+----------------------+-------------+-------------+
| salePerson           | saleAmount  | rank        |
+----------------------+-------------+-------------+
| Brown                | 6900        | 2           |
| Jones                | 7800        | 3           |
| Larry                | 3400        | 1           |
| Smith                | 10000       | 4           |
+----------------------+-------------+-------------+
```

Figure 2.12: Example of Functional mapping

In Functional mapping, first the tuples are ordered according to domains listed in the order clause, which is *saleAmount* here, and then *rank* is aggregated.

Partial Functional Mapping

Partial Functional Mapping could be viewed as the combination of Functional mapping and Equivalence reduction. It adds a group facility to functional mapping, which means tuples are first grouped by domains in group clause, then are ordered according to domains in the order clause inside each group. An example shown in Figure 2.12 is used to illustrate the usage of Partial functional mapping. Relation *SaleInfo* is used in this example.

```
>let parSum be par + of 1 order product by department;
>parRel <- [department, product, parSum] in SaleInfo;
>pr parRel;

+-------------+----------------------+-------------+
| department  | product              | parSum      |
+-------------+----------------------+-------------+
| 1           | 1                    | 1           |
| 1           | 2                    | 2           |
| 2           | 4                    | 1           |
+-------------+----------------------+-------------+
```

Figure 2.12: Example of Partial functional mapping

In this example, tuples in SaleInfo are first grouped by department. The values of *parSum* aggregate in each group following the same rule as Functional mapping.

2.1.6 Computation

The concept of Computation is similar to procedure in some programming languages such as C, Fortune. It encapsulates a set of codes together to perform a certain functionality. It accepts a list of parameters which are usually relations or other computations. Keywords "*in*" or "*out*" are used to indicate the input and output parameters. Input parameters will be used inside the computation and the result will be written into output parameters. Computation may contain several block of codes separated by keyword "*alt*" which is shorten for "alternation". Depending on the input and output parameters, different block of codes will be triggered. Computation could be declared at two levels: top level and nested level [Bak98] depending on whether the declarations of computations are nested in any declarations of relations or computations. Top level computations could be invoked anywhere after their declarations. For nested level computations, It could be invoked either in the computation code block where the nested level computations are declared or exported from Abstract Data Type where the computations are defined in. There are advanced usages of computation such as stateful computations, packages, constrain verification, etc. Readers are encouraged to refer to [Bak98] for detail information. In the following examples, the definition and invoking of nested level computations will be illustrated. In Figure 2.13, two nested computations *Add()* and *Divide()* are defined inside Abstract Data Type *calculator*. Each of the computation has three parameters. The invocations of these two nested level computations are shown in Figure 2.14. Keywords "*in*" and "*out*" are not used in the invocations, instead the shortcut is used, i.e. present parameters are input parameters and absent parameter is output parameter.

```
domain left, right, sum, division float;
domain Add comp(left, right, sum);
domain Product comp(left, right, division);
comp calculator(Add, Product) is
{

   comp Add(left, right, sum) is
   {
       sum <- left + right;
   } alt
   {
      left <- sum - right;
   } alt
   {
      right <- sum - left;
   };

   comp Divide(left, right, division) is
   {
       division <- left / right;
   };

};
```

Figure 2.13: Definition of abstract data type *calculator*

```
>calculator (out Add, out Product);
>A1 <- Add[ 3, 5,];
>pr A1;
+---------------+
| sum           |
+---------------+
| 8.0           |
+---------------+
>A2 <- Add[ 3, , 8];
>pr A2;
+---------------+
| right         |
+---------------+
| 5.0           |
+---------------+
>A3 <- Add[ ,8,10];
>pr A3;
+---------------+
| left          |
+---------------+
| 2.0           |
+---------------+
>D1 <- Divide[10.0, 5.0, ];
>pr D1;
+---------------+
| division      |
+---------------+
| 2.0           |
+---------------+
```

Figure 2.14: Invocations of computation *Add()*, *Divide()*

2.2 Introduction to Implementation of JRelix

2.2.1 System Overview

The JRelix system consists of the following three main modules: the front-end processor, the database engine, and the system database maintainer [Yu04]. These three modules interact with each other to fulfill the functions of JRelix system, as shown in Figure 2.15.

When an end-user enters a JRelix command, the command first is processed by the front-end processor which is composed of parser, interpreter and top-level evaluator. The parser accepts user input and performs syntax analysis. If any error occurs during the syntax analysis, no further process will be performed and an error message will be returned to the user to indicate error. Otherwise, a tree structure translated from the input command will be generated by the parser and passed to the interpreter.

The interpreter accepts syntax tree passed from the parser and does some evaluations such as type checking etc. It then traverses the tree and issues a set of system calls which the database engine can understand.

The database engine is critical to the JRelix system. The actual computations are performed and the results are generated by it. It consists of three main function modules, including Relation Processor, Virtual Domain Actualizer and Computation Processor, which correspond to the three conceptual aspects in JRelix system: relation Algebra, domain Algebra and computation. [Yua97]

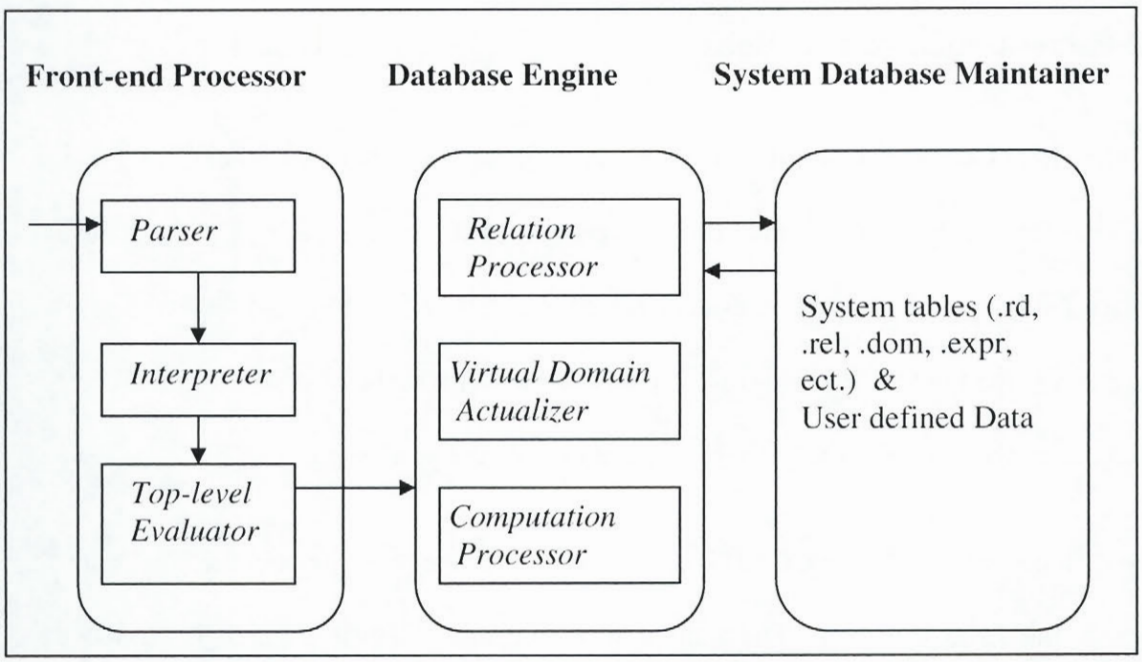

Figure 2.15: JRelix System Overview

The System database maintainer is responsible for maintaining system related information and user-defined data. Such information is maintained in a set of system tables and stored permanently as system files on the disk. The system tables are stored in files ".rel", ".dom", ".comp", ".rd", ".expr", and ".surrogate". When a user declares a relation or domain, the definitions of the relation or domain will be persistent to the system files ".rel" and ".dom", and the information of linking a relation and the domains that it is defined on is stored in file ".rd". When a user declares a computation, a syntax tree will be generated according to the definition of the computation and be stored in file ".comp". The Syntax trees for virtual domains and views declared in the system are stored in file ".expr" [Yu04]. File ".surrogate" is used to record next available surrogate for nested relation.

2.2.2 Parser Generation in JRelix

In JRelix, Java Compiler Compiler (JavaCC) is used for automatically generating parser. JavaCC is a popular parser generator for java applications. It reads the high level specification of grammar which is usually stored in a ".jjt" file, and transfers it to a set of Java classes including "Parser.java", "Token.java", "ParserTokenManager.java", etc. These classes work together to recognize the matches to the grammar.

As the preprocessor of JavaCC, JJTree inserts parse tree building actions at various places of the JavaCC source. The output of JJTree is a ".jj" file, which is passed to JavaCC to create the parser. The commands used for creating the parser are shown in Figure 2.16.

```
> jjtree Parser.jjt
> javacc Parser.jj
```

Figure 2.16: Parsing Commands

2.2.3 Virtual Domain Actualizer

Virtual Domain Actualizer is the key component of the JRelix system. It provides support for horizontal and vertical operations in domain algebra. When virtual domains are listed in the destination relation, the interpreter will call virtual domain actualizer to actualize them. The virtual domain actualizer then obtains the required source relation information from domain table and environment and instantiates the virtual domains in the destination relation. Finally, it returns the actualized destination relation back to the interpreter. The

relation between virtual domain actualizer and other compenents in JRelix is described in Figure 2.17.

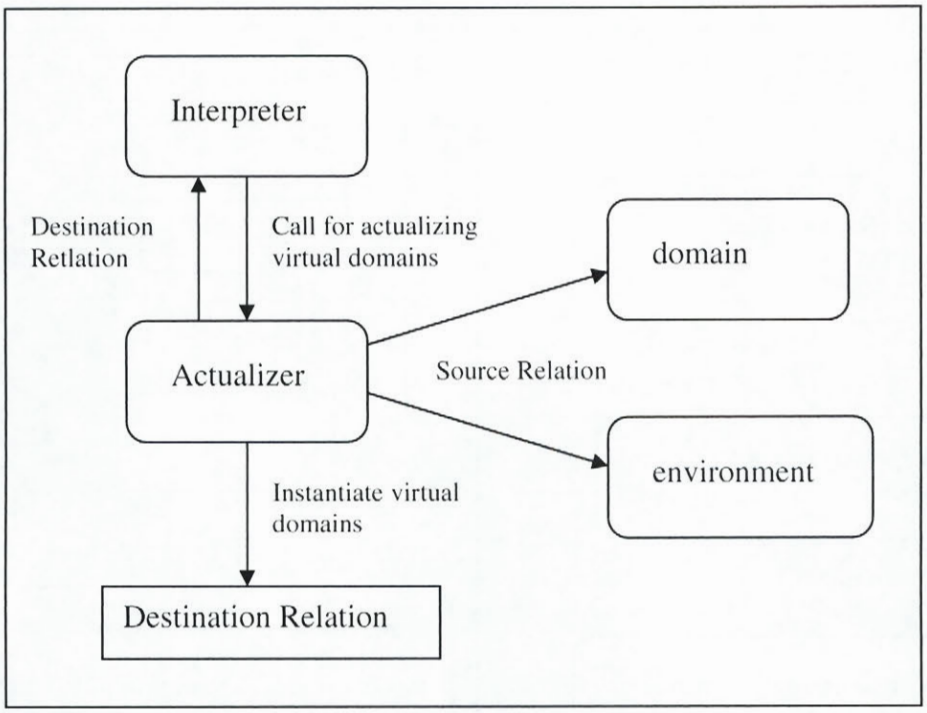

Figure 2.17: Relation Between Virtual Domain Actualizer and Other Components in JRelix

Equivalence Reduction

For the actualization of virtual domains which contain equivalence reduction operation, the following steps need to be performed before actualizing the virtual domain tuple-by-tuple [Kan01]: 1) Source relation should be loaded into memory first; 2) the syntax trees for these domains will be loaded into memory; 3) virtual domains are appended to the source relation to generate the destination relation; 4) the destination relation is sorted by the **by-domains**. Once the process of actualizing the virtual domain tuple-by-tuple starts, the start row and the end row need to be recorded to distinguish among different groups.

When the group change checking is found to be true, the virtual domains will be assigned the accumulated value in all the tuples of the group. The simplified process diagram is shown in Figure 2.18.

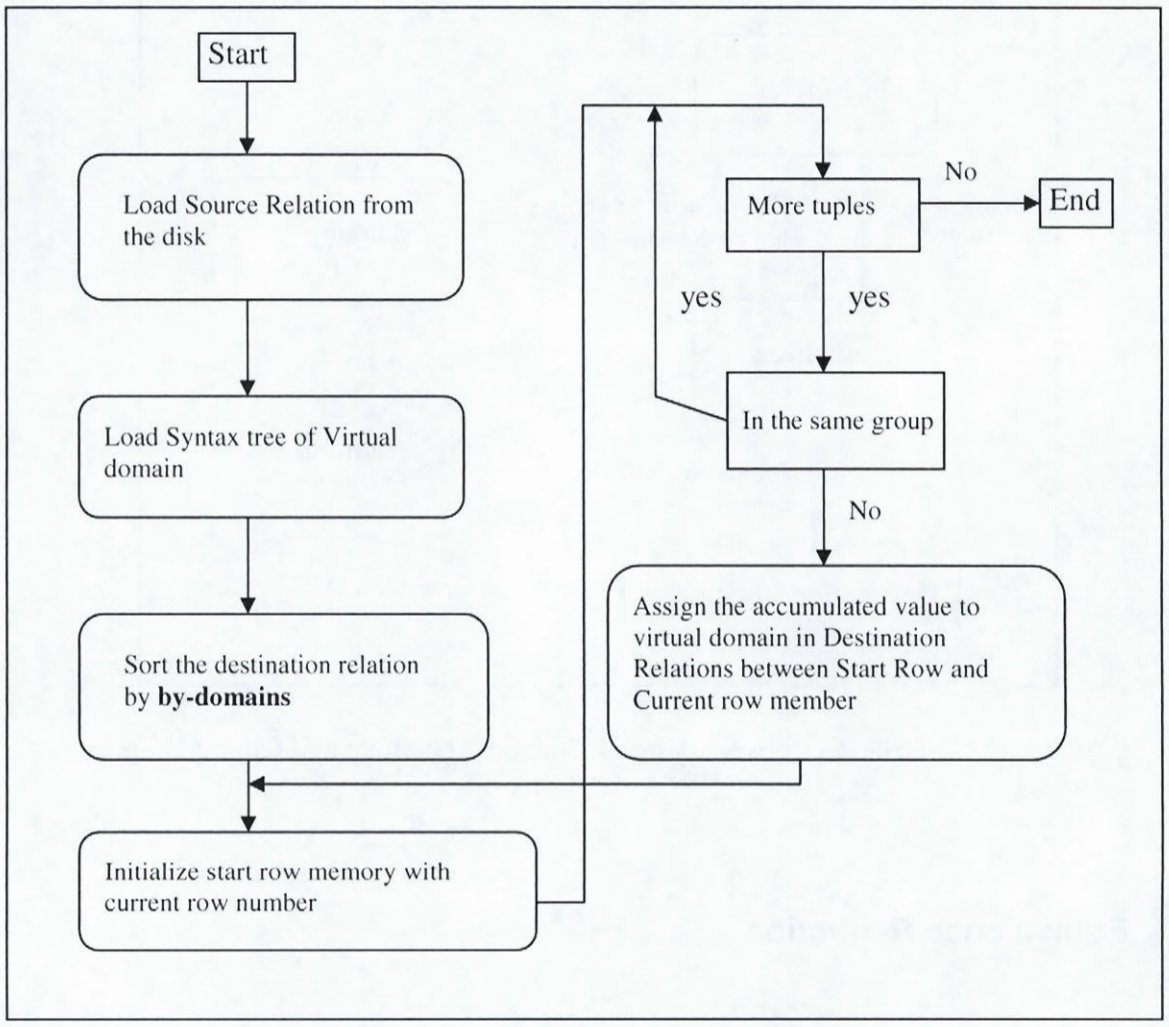

Figure 2.18: Equivalence Reduction Process Diagram [Kan01]

Functional Mapping

Similar to the actualization of a virtual domain containing equivalence reduction, when virtual domains containing functional mapping are actualized, the source relation and the

syntax trees for these domains will be loaded into memory and destination relation will be generated by adding virtual domains to the source relation. Before any further process is conducted, the destination relation needs to be sorted by the **order-domains**. Then the virtual domains are actualized tuple-by-tuple [Kan01]. The order memory needs to be initialized before two adjacent tuples are compared in order. When a change is found during the comparison, the domain value is accumulated and written to the destination relation, and the order memory is re-initialized with the changed value. In the case that there is no change in order domain memory, to avoid violating functional mapping, the accumulation process is bypassed and the current accumulated value is written into the destination relation. The simplified process is depicted in Figure 2.19.

Partial Functional Mapping

The process of actualizing a virtual domain containing partial functional mapping is similar to that of actualizing functional mapping, but with more complexity due to its definition described in chapter 2. While virtual domains are actualized with partial functional mapping operation, the destination relation will first be sorted based on its **by-domains**, and then be further sorted according to its **order-domains** in each group. Inside each group, value will be accumulated only when a change is detected during tuple-by-tuple comparison. When the group is found to have changed, the **order** and **by** memory are re-initialized. The above process is illustrated in Figure 2.20.

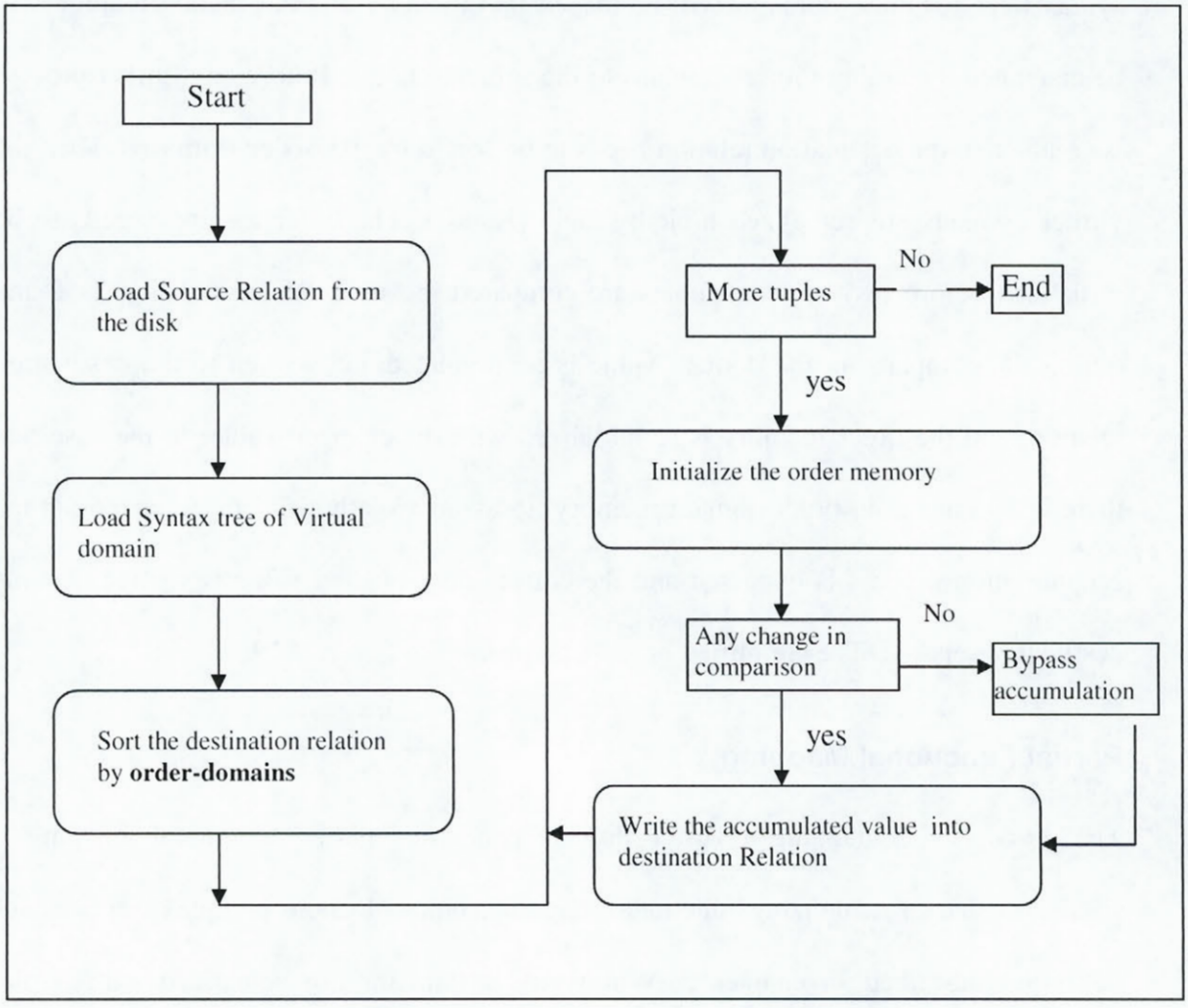

Figure 2.19: Functional Mapping Process Diagram [Kan01]

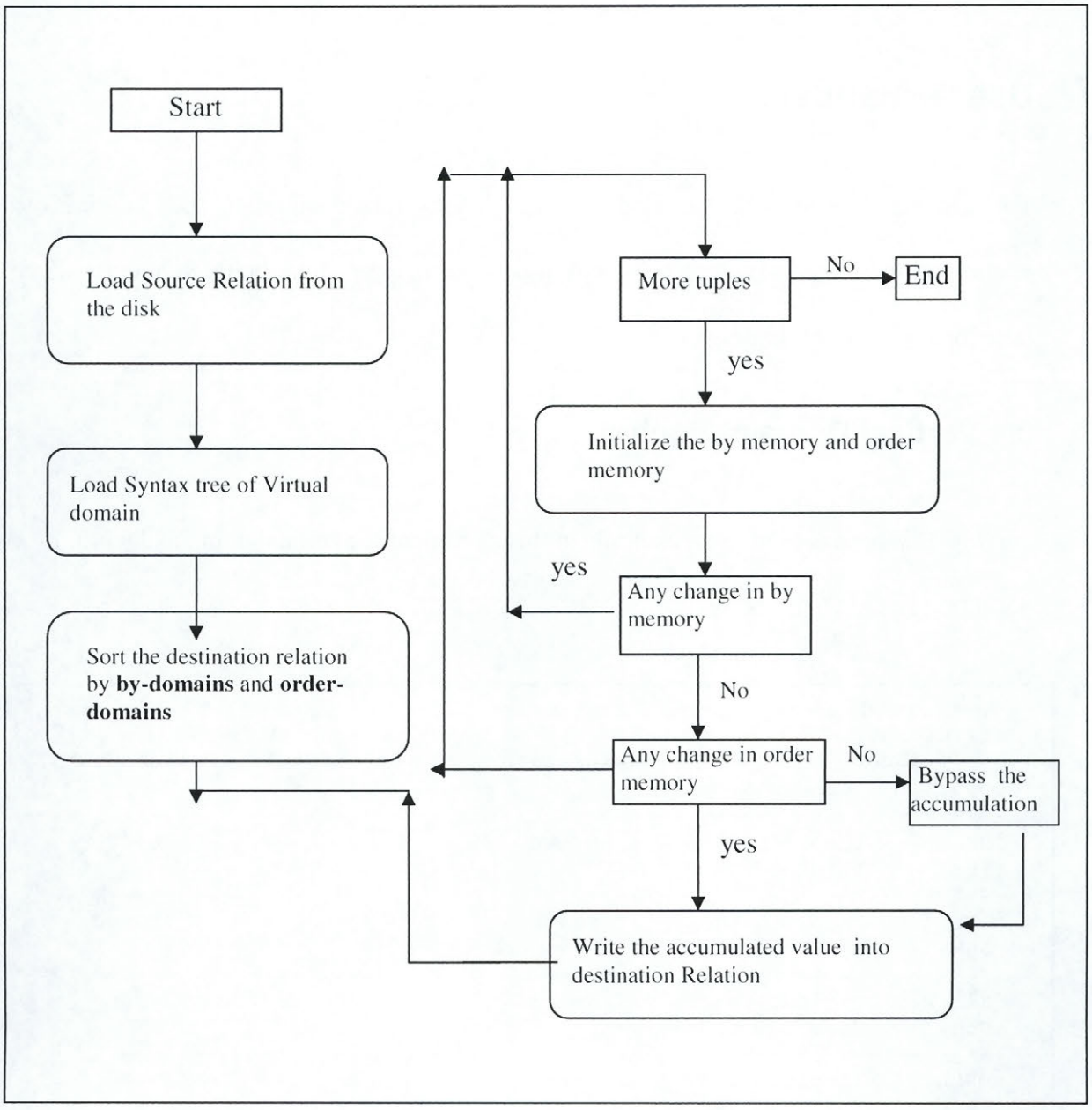

Figure 2.20: Partial Functional Mapping Process Diagram [Kan01]

Chapter 3

User's Manual

This chapter will describe the usage of grouping by relation domain and ordering by relation domain in equivalence reduction, functional mapping and the three new Boolean functions by concrete examples.

3.1 Vertical Domain Algebra

SaleInfo database is used as an example in this document for vertical domain algebra. Its definition is shown in Figure 3.1:

SaleInfo

product (company	productName)	**salePerson**	**department**	**saleAmount**
DELL	Inspiron 5150	Brown	2	6900
IBM	Thinkpad T43P			
DELL	Inspiron 5150	Smith	1	10000
IBM	Thinkpad T43P			
Kodak	Digital Camera	Jones	1	7800
Sony	Hi 8 Camcorder			
DELL	Inspiron 5150	Larry	2	3400
IBM	Thinkpad T43P			

Figure 3.1: SaleInfo Database

3.1.1 Equivalence Reduction to Support Group by Nested Relation Domain

The syntax of equivalence reduction is defined as:

Let virtual-domain **be equiv** operation **of** domain **by** domain-list

In the following example, *saleAmount* is grouped by the nested relation domain product. The declaration is shown in Figure 3.2.

```
>let equivSum be equiv + of saleAmount by product;
>EquivRel <- [product, saleAmount, equivSum] in SaleInfo;
```

Figure 3.2: Equivalence Reduction Declaration

Virtual domain *equivSum* is contained in the result relation *EquivRel*, as shown in Figure 3.3. In *EquivRel*, *saleAmount*s are accumulated according to domain product. Since the product with *saleAmount* 6900, the product with *saleAmount* 10000 and the product with *saleAmount* 3400 are the same, the value of *equivSum* for these three tuples is the sum of 6900 and 10000 and 3400. For the product with *saleAmount* 7800, which is different from the above three, the s*aleAmount* in the related tuple is not accumulated.

3.1.2 Functional Mapping to Support Order by Nested Relation Domain

The syntax of functional mapping is defined as:

Let virtual-domain **be fun** operation **of** domain **order** domain-list

```
-------------------------------------------------------------
 EquivRel
(     product                    saleAmount    equivSum)
 (company     productName)
-------------------------------------------------------------
   DELL       Inspiron 5150       6900          20300
   IBM        Thinkpad T43P
-------------------------------------------------------------
   DELL       Inspiron 5150       10000         20300
   IBM        Thinkpad T43P
-------------------------------------------------------------
   DELL       Inspiron 5150       3400          20300
   IBM        Thinkpad T43P
-------------------------------------------------------------
   Kodak      Digital Camera      7800          7800
   Sony       Hi 8 Camcorder
-------------------------------------------------------------
```

Figure 3.3: Result of Equivalence Reduction

In the example of Functional Mapping, *funSum* is ordered by domain product. The declaration is shown in Figure 3.4.

```
>let funSum be fun + of 1 order product;

>FunRel <- [product, saleAmount, funSum] in SaleInfo;
```

Figure 3.4: Functional Mapping Declarations

Virtual domain *funSum* is contained in the result relation *FunRel*, as illustrated in Figure 3.5. In relation *FunRel*, the number of product is accumulated according to domain product. Since the product with *saleAmount* 10000 is the same as the product with *saleAmount* 6900 as well as that with *saleAmount* 3400, the value of *funSum* for these three products does not increase, that is, all of them have value 1. For the product with

saleAmount 7800, which is different from the above three, the *funSum* in the related tuple is accumulated.

FunRel (product (company productName)		saleAmount	funSum)
DELL	Inspiron 5150	6900	1
IBM	Thinkpad T43P		
DELL	Inspiron 5150	10000	1
IBM	Thinkpad T43P		
DELL	Inspiron 5150	3400	1
IBM	Thinkpad T43P		
Kodak	Digital Camera	7800	2
Sony	Hi 8 Camcorder		

Figure 3.5: Result of Functional Mapping

3.1.3 Partial Functional Mapping to Support Group & Order by Nested Relation Domain

The syntax of partial functional mapping is defined as:

Let virtual-domain **be par** operation **of** domain **order** domain-list1 **by** domain-list2

In the example shown below, the number of product *parSum* is ordered by nested relation domain *product* and grouped by domain *department*. The declaration is depicted in Figure 3.6.

```
>let parSum be par + of 1 order product by department;

>ParRel <- [department, product, parSum] in SaleInfo;
```

Figure 3.6: Partial Functional Mapping Declaration

Also, virtual domain *parSum* is contained in the result relation *ParRel*, as shown in Figure 3.7.

```
-------------------------------------------------------------
 ParRel
( department            product                      parSum)
                (company    productName)
-------------------------------------------------------------
     1          DELL      Inspiron 5150              1
                IBM       Thinkpad T43P
-------------------------------------------------------------
     1          Kodak     Digital Camera             2
                Sony      Hi 8 Camcorder
-------------------------------------------------------------
     2          DELL      Inspiron 5150              1
                IBM       Thinkpad T43P
-------------------------------------------------------------
     2          DELL      Inspiron 5150              1
                IBM       Thinkpad T43P
-------------------------------------------------------------
```

Figure 3.7: Result of Partial Functional Mapping

In *ParRel*, number of product is first grouped by domain *department* and then ordered by domain *product*. Since the two products in the department with value 1 are different, the value of *parSum* increases. In the department with value 2, the value of *parSum* does not increase due to the same values of these two products.

3.2 Three New Boolean Functions

The three new Boolean functions including *isnull*, *isnulldc*, *isnulldk* are introduced in this document. The usage of these functions are similar to that of others such as *abs()*, *sin()*, *asin()*, etc., except that the output of them will be Boolean values. One example of their application is to use them in conditional statement such as "*if*".

CourseFeedBack database will be used in this document for demonstrating the usage of these functions. Its definition is shown in Figure 3.8.

CourseFeedBack

student	course	feedback
John	CS613	Good
Kartrina	CS613	dc
Larry	CS614	dc
Mary	CS613	Excellent
Patrick	CS555	Good
Rita	CS555	dk
Tommy	CS614	Great

Figure 3.8: CourseFeedBack Database

3.2.1 Function isnulldc

This function will accept one parameter, which is a domain, and test if the value of this domain is *dc* or not. It will return true if the value is *dc* and false if not. In the example

below, *isnulldc* function is used in if statements in the declaration of virtual domain *numDC*. The declarations are shown in Figure 3.9.

```
>let numDC be red + of if isnulldc(feedback) then 1 else 0;

>TestDC <- [numDC] in CourseFeedBack;
```

Figure 3.9: Function *isnulldc* declarations

In the result relation *TestDC*, the number of *dc* values contained in domain *feedback* is calculated, as given in Figure 3.10. Since there are 2 *dc* values in this domain, the value of *numDC* is 2.

```
----------------
 TestDC
( numDC )
----------------
       2
----------------
```

Figure 3.10: Result of Function *isnulldc*

3.2.2 Function isnulldk

Similar to function *isnulldc*, this function will accept one domain parameter and test if the value of this domain is *dk* or not. If it is, the function will return true; otherwise, it will return false.

In the following example, function *isnulldk* is used in the declaration of virtual domain *numDK* as shown in Figure 3.11.

```
>let numDK be red + of if isnulldk(feedback) then 1 else 0;
>TestDK <- [numDK] in CourseFeedBack;
```

Figure 3.11: Function *isnulldk* Declarations

In the result relation *TestDK*, virtual domain *numDK* is calculated according to the number of *dk* value in domain *feedback*. Since there is only 1 *dk* value in domain *feedback*, the value of *numDK* is 1, as we can see in Figure 3.12.

```
----------------
 TestDK
( numDK )
----------------
       1
----------------
```

Figure 3.12: Result of Function *isnulldk*

3.2.3 Function isnull

Similar to the above two functions *isnulldc* and *isnulldk*, this function will accept one domain parameter and test if the value of the domain is *dk* or *dc*. If it is *dk* or *dc*, the function will return the Boolean value true. Otherwise, it will return false.

In the example of Figure 3.13, function *isnull* is used in the declaration of virtual domain *unknown*.

```
>let unknown be red + of if isnull(feedback) then 1 else 0;
>TestNULL <- [unknown] in CourseFeedBack;
```

Figure 3.13: Function *isnull* Declarations

In the result relation *TestNull*, virtual domain *unknown* is calculated according to the number of *dc* and *dk* value in domain *feedback*. Since there are 3 such values in domain *feedback*, the value of unknown is 3, as shown in Figure 3.14.

TestNULL
(unknown)
3

Figure 3.14: Result of Function *isnull*

3.2.4 Further Examples

In the following document, three more complete examples will be given to illustrate the usage of the three Boolean functions.

Example for Function isnulldc

Relation *R* which contains 5 different type of attributes is used in this example, with definition shown in Figure 3.15.

```
>domain d1 strg;
>domain d2 double;
>domain d3 boolean;
>domain d4 short;
>domain d5 long;
>relation R (d1, d2, d3, d4, d5) <- { ("try", 11.2, true, 3, 12),
                                      (dc,    dc,   dc,   dc,dc),
                                      ("this", 2.5, false, 6, 33)};

>pr R;

+----------------------+--------------------------+--------+--------+-------+
| d1                   | d2                       | d3     | d4     | d5    |
+----------------------+--------------------------+--------+--------+-------+
| _dc                  | dc                       | dc     | dc     | dc    |
|                      |                          |        |        |       |
| this                 | 2.5                      | false  | 6      | 33    |
|                      |                          |        |        |       |
| try                  | 11.199999809265137       | true   | 3      | 12    |
+----------------------+--------------------------+--------+--------+-------+
```

Figure 3.15: Definition of Relation R

Function *isnulldc* is used in the definitions of virtual domains such as *t1*, *t2*, *t3*, *t4*, and *t5* to test the value of different type of attributes in relation *R*. The result relations containing the virtual domains are shown in Figure 3.16.

```
>let t1 be if isnulldc(d1) then "yes" else "no";
>R1 <- [d1, t1] in R;
>pr R1;
+----------------------+----------------------+
| d1                   | t1                   |
+----------------------+----------------------+
| _dc                  | yes                  |
| this                 | no                   |
| try                  | no                   |
+----------------------+----------------------+

>let t2 be if isnulldc(d2) then 0.0 else 1.0;
>R2 <- [d2, t2] in R;
>pr R2;
+--------------------------+---------------+
| d2                       | t2            |
+--------------------------+---------------+
| dc                       | 0.0           |
| 2.5                      | 1.0           |
| 11.199999809265137       | 1.0           |
+--------------------------+---------------+

>let t3 be if isnulldc(d3) then true else false;
>R3 <- [d3, t3] in R;
>pr R3;
+--------+--------+
| d3     | t3     |
+--------+--------+
| dc     | true   |
| false  | false  |
| true   | false  |
+--------+--------+

>let t4 be if isnulldc(d4) then 1 else 0;
>R4 <- [d4, t4] in R;
>pr R4;
+--------+-------------+
| d4     | t4          |
+--------+-------------+
| dc     | 1           |
| 3      | 0           |
| 6      | 0           |
+--------+-------------+

>let t5 be if isnulldc(d5) then 1 else 0;
>R5 <- [d5, t5] in R;
>pr R5;
+----------------------+-------------+
| d5                   | t5          |
+----------------------+-------------+
| dc                   | 1           |
| 12                   | 0           |
| 33                   | 0           |
+----------------------+-------------+
```

Figure 3.16: Result of Example for Function *isnulldc*

Example for Function isnulldk

Similar to the above example, relation *R* which contains five different types of attributes including string, double, Boolean, short and long is used to illustrate the usage of function *isnulldk*. The definition of relation *R* is shown in Figure 3.17.

```
>domain d1 strg;
>domain d2 double;
>domain d3 boolean;
>domain d4 short;
>domain d5 long;
>relation R (d1, d2, d3, d4, d5) <- { ("try", 11.2, true, 3, 12),
                                      (dk,    dk,   false,   dk,33),
                                      ("this", 2.5, dk, 6, dk)};

>pr R;

+----------------------+--------------------------+--------+--------+-------+
| d1                   | d2                       | d3     | d4     | d5    |
+----------------------+--------------------------+--------+--------+-------+
| _dk                  | dk                       | false  | dk     | 33    |
|                      |                          |        |        |       |
| this                 | 2.5                      | dk     | 6      | dk    |
|                      |                          |        |        |       |
| try                  | 11.199999809265137       | true   | 3      | 12    |
+----------------------+--------------------------+--------+--------+-------+
```

Figure 3.17: Definition of Relation *R*

Function *isnulldk* is used in the definitions of virtual domains such as *t1*, *t2*, *t3*, *t4*, and *t5*. The result relations *R1*, *R2*, *R3*, *R4* and *R5* containing the virtual domains *t1*, *t2*, *t3*, *t4*, *t5* are shown in Figure 3.18.

```
>let t1 be if isnulldk(d1) then "yes" else "no";
>R1 <- [d1, t1] in R;
>pr R1;
+----------------------+----------------------+
| d1                   | t1                   |
+----------------------+----------------------+
| _dk                  | yes                  |
| this                 | no                   |
| try                  | no                   |
+----------------------+----------------------+

>let t2 be if isnulldk(d2) then 0.0 else 1.0;
>R2 <- [d2, t2] in R;
>pr R2;
+--------------------------+----------------+
| d2                       | t2             |
+--------------------------+----------------+
| 2.5                      | 1.0            |
| 11.199999809265137       | 1.0            |
| dk                       | 0.0            |
+--------------------------+----------------+

>let t3 be if isnulldk(d3) then true else false;
>R3 <- [d3, t3] in R;
>pr R3;
+--------+--------+
| d3     | t3     |
+--------+--------+
| dk     | true   |
| false  | false  |
| true   | false  |
+--------+--------+

>let t4 be if isnulldk(d4) then 1 else 0;
>R4 <- [d4, t4] in R;
>pr R4;
+--------+-------------+
| d4     | t4          |
+--------+-------------+
| dk     | 1           |
| 3      | 0           |
| 6      | 0           |
+--------+-------------+

>let t5 be if isnulldk(d5) then 1 else 0;
>R5 <- [d5, t5] in R;
>pr R5;
+----------------------+-------------+
| d5                   | t5          |
+----------------------+-------------+
| dk                   | 1           |
| 12                   | 0           |
| 33                   | 0           |
+----------------------+-------------+
```

Figure 3.18: Result of Example for Function *isnulldk*

Example for Function isnull

In the example, relation *R* contains five different types of attributes. The values of these attributes are mixing of *dc*, *dk* and normal value. The definition of relation *R* is shown in Figure 3.19.

```
>domain d1 strg;
>domain d2 double;
>domain d3 boolean;
>domain d4 short;
>domain d5 long;
>relation R (d1, d2, d3, d4, d5) <- {("try", 11.2, dk,  3,  dk),
                                    (dc,    dc,   dc,    dc, dc),
                                    (dk,    dk,   false, dk,  33)};
>pr R;
+----------------------+--------------------------+--------+--------+----+
| d1                   | d2                       | d3     | d4     | d5 |
+----------------------+--------------------------+--------+--------+----+
| _dk                  | dk                       | false  | dk     | 33 |
|                                                                        |
| _dc                  | dc                       | dc     | dc     | dc |
|                                                                        |
| try                  | 11.199999809265137       | dk     | 3      | dk |
+----------------------+--------------------------+--------+--------+----+
```

Figure 3.19: Definition of Relation *R*

Function *isnull* is used in the definition of virtual domains *t1*, *t2*, *t3*, *t4* and *t5* to test the *dc* and *dk* value containing in attributes *d1*, *d2*, *d3*, *d4* and *d5* of Relation *R*. The result relations are shown in Figure 3.20.

```
>let t1 be if isnull(d1) then "yes" else "no";
>R1 <- [d1, t1] in R;
>pr R1;
+----------------------+----------------------+
| d1                   | t1                   |
+----------------------+----------------------+
| _dk                  | yes                  |
| _dc                  | yes                  |
| try                  | no                   |
+----------------------+----------------------+

>let t2 be if isnull(d2) then 0.0 else 1.0;
>R2 <- [d2, t2] in R;
>pr R2;
+-------------------------+---------------+
| d2                      | t2            |
+-------------------------+---------------+
| dc                      | 0.0           |
| 11.199999809265137      | 1.0           |
| dk                      | 0.0           |
+-------------------------+---------------+

>let t3 be if isnull(d3) then true else false;
>R3 <- [d3, t3] in R;
>pr R3;
+--------+--------+
| d3     | t3     |
+--------+--------+
| dk     | true   |
| dc     | true   |
| false  | false  |
+--------+--------+

>let t4 be if isnull(d4) then 1 else 0;
>R4 <- [d4, t4] in R;
>pr R4;
+--------+-------------+
| d4     | t4          |
+--------+-------------+
| dk     | 1           |
| dc     | 1           |
| 3      | 0           |
+--------+-------------+

>let t5 be if isnull(d5) then 1 else 0;
>R5 <- [d5, t5] in R;
>pr R5;
+----------------------+-------------+
| d5                   | t5          |
+----------------------+-------------+
| dk                   | 1           |
| dc                   | 1           |
| 33                   | 0           |
+----------------------+-------------+
```

Figure 3.20: Result of Example for Function *isnull*

Chapter 4

Implementation

In this chapter, the implementation for the functionalities of grouping by relation domain and ordering by relation domain in equivalence reduction and functional mapping, as well as the three new Boolean functions are described. The implementation is based on the previous implementation of JRelix. In Section 4.1, the develop environment of this project will be introduced briefly. In Section 4.2, the implementation for the new features of grouping by relation domain and ordering by relation domain in equivalence reduction and function mapping will be discussed in details. The implementation of the three new Boolean functions will be shown in Section 4.3.

4.1 Development Environment

This project is written in Java and has been developed under JDK 1.4.2 environment. Jbuilder 2005 Foundation is used for developing, testing and debugging purpose. The compiled JRelix runs on both Windows and Linux.

4.2 Vertical Domain Operations to Support Nested Relation Domain

In the previous JRelix version, although domain list could contain nested relation domains, the system can not distinguish between the same values of two nested relation domains since the system determines whether the values of these two domains are the

same according to the surrogates instead of the real values. Now the system is extended to be able to interpret nested relation domain used in **by-domains** or **order-domains** correctly by using the real values instead of surrogates to determine if two nested relation domains are the same. The following sections will describe the implementation of grouping by relation domain and ordering by relation domain in equivalence reduction and functional mapping in details.

4.2.1 Equivalence Reduction to Support Group by Nested Relation Domain

As mentioned in Chapter 2, virtual domain actualizing is performed in the class "Actualizer.java" and a set of methods such as *actIntCell()*, *actNumCell()*, etc. are implemented to actualizing a "cell" according to the domain type. Given a virtual domain which contains equivalence reduction operation, the actualizing process is done in method *actualizeEquiv()*.

The previous method *actualizeEquiv()* already has the ability to sort destination relation according to the **by-domains.** If there are nested relation domains in the **by-domains**, it sorts according to the real values instead of the surrogates of the nested relation domains by calling method *Relation.sort(Domain[])*. Such sorting process is very important since it ensures the precondition, which is required by the subsequent tuple-by-tuple comparison to correctly detect the boundary of each group, is satisfied. However, when comparing two values of a nested relation domain, the previous implementation uses surrogate values instead of the real values to determine if these two are the same. Therefore, those codes in the previous implementation should be modified. The current

implementation uses existing method *Relation.compareTwoRows()* to compare the real values of a nested relation domain. As its name indicates, method *compareTwoRows()* compares the values of the attributes. If the attributes are nested relation domains, it calls method *Relation.compareRelation()* to compare the real values of the relations domain as shown in Figure 4.1. Since method *compareRelation()* is a recursive method, even if the comparing relation is a nested relation (meaning that the relation contains nested relation domains), it can still generate the correct result.

```
if (domsl[j].type==IDLIST)
 {
      …
      Relation doml = myEnv.lookupRel("."+domsl[j].name, true);
      Relation rl = doml.getRelation(bm);
      Relation domr = myEnv.lookupRel("."+dn.name, true);
      Relation rr = domr.getRelation(bn);
      int result = rl.compareRelation(rr);
      if (result != EQ) return result;
      else break;
}
```

Figure 4.1: Method *compareTwoRows()*, which handles nested relation domain.

Before the method *compareTwoRows()* is called, the value of nested relation domain, which is in **by-domains**, needs to be recorded in **by-memory**. In the current implementation, the row number is chosen to be stored because it is a required argument when method *compareTowRows()* is called. The codes are shown in Figure 4.2.

```
For (i=0; i < byarray.length; i++)
{
     switch(byarray[i].type)
     {
          …
          case IDLIST:

            if (env.lookupRel("."+byarray[i].name, true) != null)
            {
                 Integer rownum = new Integer(currow);
                 vals.addElement(rownum);
            }
            break;
             …
       }
}
```

Figure 4.2: Codes added in *actualizeEquiv()* to initialize **by-memory**

The codes added to compare the real value of nested relation domains are shown in Figure 4.3. First, whether the domain is a nested relation domain is checked by looking up if a relation named .domain name exists. Then arguments for method *Relation.compareTwoRows()* are constructed. Among them, array *flags* contains Boolean values for each domain stored in array `domains`. Such Boolean values are used to signal method *compareTwoRows()* about whether the values for a nested relation domain should be compared by surrogates or not. Assigning *false* indicates such nested relation domain should be compared by the real value. If the result value that *compareTwoRows()* returns is not equal to 0, i.e., *result != 0*, which means the two values are not the same and the boundary of two different group is reached, the **by-memory** is reset to the current row and the *breakflag* is set to 1 to indicate that the current group reaches its boundary and the current accumulated value should be written to the destination relation. If *result = 0*, which means the two values are the same and they are still in the same group, the surrogate value stored in the by-memory is assigned

to the current nested relation domain to ensure that in the destination relation, the same values of the nested relation domain are represented by the same surrogate.

```
for (i=0;i < byarray.length; i++)
{
     …
     switch(byarray[i].type)
    {
          …
          case IDLIST:
           // if this domain is a nest relation
           if (env.lookupRel("."+byarray[i].name, true) != null)
           {
                Relation tmpr = new Relation(env);
                Object[] nestdata = new Object[1];
                nestdata[0] = destrel.data[bypos[i]];
                int mrow = ( (Integer) (vals.elementAt(i))).intValue();
                Domain[] domains = new Domain[1];
                domains[0] = byarray[i];
                boolean[] flags = new boolean[1];
                flags[0] = false;
                int[] atypes = tmpr.toTypes(domains);
                int result = tmpr.compareTwoRows(nestdata, mrow,
                             nestdata, currow, 1, atypes, flags, domains,
                             domains);
                if (result != 0)
                {
                     vals.setElementAt(new Integer(currow), i);
                     breakflag = 1;
                }
                else
                {
                     //if two row are equivalent, update surrogate
                     ((long[])destrel.data[bypos[i]])[currow] =
                             ((long[])destrel.data[bypos[i]])[mrow];
                 }
           }
           break;
           …
    }
}
```

Figure 4.3: Codes added in *actualizeEquiv()* to compare the real value of nested relation domain.

4.2.2 Functional Mapping to Support Order by Nested Relation Domain

Similar to equivalence reduction, given a virtual domain that contains function mapping operation, the actualizing process is done in method *Actualizer. actualizeFun()*.

The implementation of method *actualizeFun()* follows the same steps as *actualizeEquiv()*, except that destination relation is sorted by by-domains in *actualizeEquiv()*, while it is sorted based on **order-domains** in *actulizeFun()*. Also, the time when the accumulated value should be written to the destination relation is different. Same as *actualizeEquiv()*, the destination relation is sorted by calling method *Relation.sort()*. In *actualizeFun()*, first the destination relation is sorted based on the real values of nested relation domains contained in the **order-domains**, and then the accumulated value is calculated through tuple-by-tuple comparison. Similar to the situation in method *actualizeEquiv()*, the previous implementation of *actulizeFun()* compares the nested relation domains according to surrogates instead of the real values. In the current implementation, the following codes are added to ensure that tuple-by-tuple comparison is based on the real values. In the codes shown in Figure 4.4, the order-memory is filled with current row number if the order domain is a nested relation domain.

```
case IDLIST:

   if (env.lookupRel("."+orderarray[i].name, true) != null)
   {
          Integer rownum = new Integer(currow);
          ordermemory.addElement(rownum);
   }

   break;
```

Figure 4.4: Codes added in *actualizeFun()* to initialize **order-memory**

The codes illustrated in Figure 4.5 present the way of constructing the comparison based on the real values of the nested relation domains. Similar to that described in the above section, method *Relation.CompareTwoRows()*is used to compare the real values of nested relation domains. If the two values are different (indicated by *result != 0* in the codes), the *ordermemory* needs to be reset and *breakflag1* is set to be true. If the two values are the same, the surrogate value in the order-memory is assigned to the current nested relation domain to ensure that in the destination relation the same values of the relation domain have the same surrogate.

```
For (i=0; i<orderarray.length; i++)
{ …
  switch(orderarray[i].type)
  {  …
     case IDLIST:
       //if this domain is a nest relation
       if (env.lookupRel("."+orderarray[i].name, true) != null)
       {
           Relation tmpr = new Relation(env);
           Object[] nestdata = new Object[1];
           nestdata[0] = destrel.data[orderpos[i]];
           int mrow = ( (Integer) (ordermemory.elementAt(i))).intValue();
           Domain[] domains = new Domain[1];
           domains[0] = orderarray[i];
           boolean[] flags = new boolean[1];
           flags[0] = false;
           int[] atypes = tmpr.toTypes(domains);
           int result = tmpr.compareTwoRows(nestdata, mrow, nestdata,
                        currow,1, atypes, flags, domains, domains);
           if (result != 0)
           {
                ordermemory.setElementAt(new Integer(currow), i);
                breakflag1 = true;
           }
           else
           {
                 //if two rows are equivalent, update surrogate
                  ((long[])destrel.data[orderpos[i]])[currow] =
                  ((long[])destrel.data[orderpos[i]])[mrow];
           }
       }
      …
   }
  …
}
```

Figure 4.5: Codes added in *actualizeFun()* to compare the real value of nested relation domain.

4.2.3 Partial Functional Mapping to Support Group & Order by Nested Relation Domain

In JRelix, given a virtual domain that contains partial functional mapping operation, the actualizing process for this virtual domain is carried out in method *Actualizer. actualizeParFun()*.

Since partial functional mapping has more complicated definition than equivalence reduction and functional mapping, the implementation of *actualizeParFun()* presents more complexity although some parts of the implementation are similar to those of methods *actualizeEquiv()* and *actualizeFun()*. In the definition of partial functional mapping, the value of a virtual domain containing partial functional mapping operation is accumulated according to its **by-domains** and **order-domains**. In method *actualizeParFun()*, destination relation is first sorted by **byorder-domains** which is the combination of **by-domains** and **order-domains**; then *bymemory* and *ordermemory* are initialized sequentially before the tuple-by-tuple comparison is performed. To be able to support group & order by nested relation domains, in the process of initializing *bymemory* and *ordermemory* the following codes shown in Figures 4.6 and 4.7 are added.

```
case IDLIST:
   if (env.lookupRel("."+byarray[i].name, true) != null) {
         Integer rownum = new Integer(currow);
         bymemory.addElement(rownum);
   }
   break;
```

Figure 4.6: Codes added to initialize *bymemory*

```
case IDLIST:
 if(env.lookupRel("."+orderarray[i].name, true) != null) {
        Integer rownum = new Integer(currow);
        ordermemory.addElement(rownum);
 }
 break;
```

Figure 4.7: Codes added to initialize *ordermemory*

In the process of tuple-by-tuple comparison, we need to compare nested relation domains based on the real values instead of the surrogates, and the related codes are presented in Figure 4.8 and Figure 4.9. Codes in Figure 4.8 are used to detect the group boundary, while codes in Figure 4.9 are used to detect the order boundary. When the current values are different from the values in *bymemory* or *ordermemory*, i.e. the boundaries are reached, the breakflag will be set to be true and the values in *bymemory* or *ordermemory* will be set to the current values. In the case that the current values are the same as the values in *bymemory* and *ordermemory*, the surrogates of the current nested relation domains will be set as the same as those in *bymemory* and *ordermemory* so that in the destination relation, equivalent values of a nested relation domain will be represented by the same surrogate. Similar to the implementation of *actualizeEquiv()* and *actualizeFun()*, the existing method *Relation.compareTwoRows()* is used to compare the nested relation domain by its real values.

```
for(currow=0; currow < destrel.numtuples;currow++)
{
    …
    for(i=0;i<byarray.length;i++)
    {
        switch(byarray[i].type)
        {
            …
            case IDLIST:
                // if this domain is a nest relation
                if(env.lookupRel("." + byarray[i].name, true) != null)
                {
                    Relation tmpr = new Relation(env);
                    Object[] nestdata = new Object[1];
                    nestdata[0] = destrel.data[bypos[i]];
                    int mrow = ( (Integer)
                                 (bymemory.elementAt(i))).intValue();
                    Domain[] domains = new Domain[1];
                    domains[0] = byarray[i];
                    boolean[] flags = new boolean[1];
                    flags[0] = false;
                    int[] atypes = tmpr.toTypes(domains);
                    int result = tmpr.compareTwoRows(nestdata, mrow,
                                 nestdata, currow,
                                 1, atypes, flags, domains, domains);
                    if (result != 0)
                    {
                        bymemory.setElementAt(new Integer(currow), i);
                        breakflag = true;
                    }
                    else
                    {
                     // if two rows are equivalent, update the surrogate
                      ((long[])destrel.data[bypos[i]])[currow] =
                       ((long[])destrel.data[bypos[i]])[mrow];
                    }
                }
                break;
                …
        }
    }
    …
}
```

Figure 4.8: Codes added to detect the group boundary

```
for(i=0;i<orderarray.length;i++)
{
    switch(orderarray[i].type)
    {
       …
       case IDLIST:
            // if this domain is a nest relation
            if (env.lookupRel("."+orderarray[i].name, true) != null)
            {
               Relation tmpr = new Relation(env);
               Object[] nestdata = new Object[1];
               nestdata[0] = destrel.data[orderpos[i]];
               int mrow = ( (Integer)
                           (ordermemory.elementAt(i))).intValue();
               Domain[] domains = new Domain[1];
               domains[0] = orderarray[i];
               boolean[] flags = new boolean[1];
               flags[0] = false;
               int[] atypes = tmpr.toTypes(domains);
               int result = tmpr.compareTwoRows(nestdata, mrow,
                           nestdata, currow, 1, atypes, flags,
                           domains, domains);
               if (result != 0)
               {
                   ordermemory.setElementAt(new Integer(currow), i);
                   breakflag1 = true;
               }
               else
               {
                  ((long[])destrel.data[orderpos[i]])[currow] =
                  ((long[])destrel.data[orderpos[i]])[mrow];
               }
            }
            break;
         …
    }
}
```

Figure 4.9: Codes added to detect the order boundary

4.3 Three New Boolean Function Implementation

To implement the three new Boolean functions: *isnulldc*, *isnulldk* and *isnull* introduced in chapter 2, we first need to modify the grammar file and generate new parser so that these new functions could be recognized when the codes entered by users contain these functions. The lines shown in Figure 4.10 are added into the grammar file Parser.jjt to generate new parser.

```
TOKEN :                          /* FUNCTIONS */
{
    ...
    ...
   < ISNULLDC: "isnulldc" > |
   < ISNULLDK: "isnulldk" > |
   < ISNULL: "isnull" >
    ...
}
```

Figure 4.10: Modifications in Parser.jjt

In JRelix, as mentioned before, the command users enter will be analyzed first by the parser, and then a syntax tree derived from it will be generated and passed to the interpreter for further process. The syntax tree contains all the information of a command. Such information is stored in linked Simple Node objects, which are components of the syntax tree. Simple Node class contains a set of attributes and methods. Among them, attribute *opcode* is used to store the functions or operations information. Since the type of *opcode* is integer, every function or operation defined in JRelix needs a constant integer value to represent itself in Simple Node object. The integer assigned to each function or operation is defined in file "Constants.java". Integers also need to be assigned to the three

new Boolean functions so that these functions can be represented in Simple Node object. The codes added into "Constants.java" are shown in Figure 4.11.

```
static final int OP_ISNULLDC =  520
static final int OP_ISNULLDK =  521;
static final int OP_ISNULL   =  522;
```

Figure 4.11: Codes added in Constants.java

When a virtual domain is declared, several checkings should be finished before its syntax tree is cut off from the syntax tree of declaration command and put into *domtable*. One of them is to check if there is any mistype matching in the definition of the virtual domain. Such task is performed in method *interpreter.traveType()* which is also used to determine the type of the virtual domain. In this method, different cases are first categorized by *node.type*, and then further distinguished by *node.opcode*. Each case is handled separately. To handle the three new Boolean functions, the following codes are added in the method *interpreter.traveType()* as illustrated in Figure 4.12.

During the actualization of virtual domains, virtual tree building is a very important procedure, which serves as a preprocess to filter out those virtual domains that could not be actualized and only allows those that could be actualized to pass to the actualizing engine. In JRelix, virtual tree building is performed in method *actualizer.buildTree()*. In addition to the building virtual tree, other functionalities such as validality check, virtual Tree expansion, recursive loop detection etc. are also implemented in this method. The

following codes shown in Figure 4.13 are added in *buildTree()* method to guarantee that the correct type will be returned for the three new Boolean functions.

```
private int traverseType(SimpleNode node, Environment env)
        throws InterpretError
{
   …
   switch(node.type)
   {
        …
        case OP_FUNCTION:
              …
              switch (node.opcode)
              {
                    …
                    case OP_ISNULL:
                    case OP_ISNULLDC:
                    case OP_ISNULLDK:
                          return BOOLEAN;
                    …
               }
          …
    }
```

Figure 4.12: Codes Added in Method traverseType

```
private int buildTree(SimpleNode node)
    throws InterpretError
{
      …
       switch(node.type)
       {
                …
                case OP_FUNCTION:
                …
                switch (node.opcode)
               {
                     …
                     case OP_ISNULL:
                     case OP_ISNULLDC:
                     case OP_ISNULLDK:
                             return BOOLEAN;
                     …
                }
         }
}
```

Figure 4.13: Codes Added in Method buildTree()

In JRelix, actualizing a virtual domain could be viewed as filling calculated values into the corresponding position in a table. Such position is specified as "cell" in JRelix implementation. A bunch of methods, including *actIntCell()*, *actStrCell()*, *actRelCell()*, etc., are implemented in "Actualizer.java" to handle the filling of different types of "cell". Among them, *actBoolCell()* is specified to calculate Boolean value. Since the three new functions are Boolean functions, the results of these functions should be calculated in the method *actBoolCell()*. The following codes shown in Figure 4.14 are added in this method.

```
private byte actBoolCell_old(SimpleNode node) throws InterpretError
 {
    switch (node.type)
    {
       …
       case OP_FUNCTION:
            if (node.opcode == OP_ISNULL)
            {
               SimpleNode arg = (SimpleNode) node.jjtGetChild(0);
               if (isnulldc(arg) == (byte)BOOL_TRUE ||
                   isnulldk(arg) == (byte)BOOL_TRUE)
                            return BOOL_TRUE;
                else
                            return BOOL_FALSE;
             }
             else if (node.opcode == OP_ISNULLDC)
             {
                 SimpleNode arg = (SimpleNode) node.jjtGetChild(0);
                 return isnulldc(arg);
              }
              else if (node.opcode == OP_ISNULLDK)
              {
                   SimpleNode arg = (SimpleNode) node.jjtGetChild(0);
                   return isnulldk(arg);
               }
              else
                  throw new InterpretError("actBoolCell: function is not
                            boolean function");
                  …
   }
```

Figure 4.14: Code Added in Method ActBoolCell()

Furthermore, three auxiliary methods are introduced in Actualizer.java, including *isnulldc()*, *isnulldk()* and *getType()*. Method *isnulldc()* is used to determine if the cell value is *dc* or not, and returns *true* or *false* correspondingly. Since *dc* value is represented differently for different types of domain in JRelix system as seen in Figure 4.15, and also as mentioned before, different types of cell are calculated by different methods such as *actInt()*, *atclong()*, etc., in Method *isnulldc()* the node type will be checked first and depending on its type, different actualizing methods will be called to calculate the cell value. Next, the calculated cell value will be compared with different *dc* values. The method is shown in Figure 4.16.

Integer, short	INT_DC	INT_DK
Long	LONG_DC	LONG_DK
Double, float	DOUBLE_DC	DOUBLE_DK
Number	NUMERIC_DC	NUMERIC_DK
String, Attribute	STRING_DC	STRING_DK
Boolean	BOOL_DC	BOOL_DK

Figure 4.15: DC, DK Value Represented in JRelix

The implementation of method *isnulldk()* is similar to that of method *isnulldc()*. In this method the node type is checked first, and handled differently according to its type. The calculated cell value is then compared with different *dk* values. The representation of *dk* value is shown in Figure 4.15 and the code for Method *isnulldk()* is illustrated in Figure 4.17.

Method *getType()* is used in Method *isnulldc()* and *isnulldk()* to return the type name of a node. Such information is used when a InterpretError is thrown. Figure 4.18 lists the code for this method.

```
private byte isnulldc(SimpleNode arg) throws InterpretError
{
     byte bm = BOOL_DC;
     int im = INT_DC;
     long lm = LONG_DC;
     double dm = DOUBLE_DC;
     String sm = STRING_DC;
     number nm = NUMERIC_DC;

     switch(destrel.domains[arg.bits].type) {
          case BOOLEAN:
               bm = actBoolCell(arg);
               if (bm == BOOL_DC)
                         return BOOL_TRUE;
               else
                         return BOOL_FALSE;
          case SHORT:
          case INTEGER:
               im = actIntCell(arg);
               if (im == INT_DC)
                         return BOOL_TRUE;
               else
                         return BOOL_FALSE;
          case LONG:
               lm = actLongCell(arg);
               if (lm == LONG_DC)
                         return BOOL_TRUE;
               else
                         return BOOL_FALSE;
          case FLOAT:
          case DOUBLE:
                dm = actDoubleCell(arg);
                if (dm == DOUBLE_DC)
                         return BOOL_TRUE;
                else
                         return BOOL_FALSE;
          case ATTRIBUTE:
          case STRING:
                sm = actStrCell(arg);
                if (sm.compareTo(STRING_DC) == 0)
                          return BOOL_TRUE;
                else
                          return BOOL_FALSE;
          case NUMERIC:
                nm = actNumCell(arg);
                if(nm.compareTo(NUMERIC_DC) == 0)
                          return BOOL_TRUE;
                else
                          return BOOL_FALSE;
          case IDLIST:
          default:
                Utility.dump(arg, "actBoolCell ERROR=>");
                throw new InterpretError("nulldc(" + getType(arg) + "): to
                                          be implemented");
     }
}
```

Figure 4.16: Method *isnulldc()*

```
private byte isnulldk(SimpleNode arg) throws InterpretError
{
    byte bm = BOOL_DK;
    int im = INT_DK;
    long lm = LONG_DK;
    double dm = DOUBLE_DK;
    String sm = STRING_DK;
    number nm = NUMERIC_DK;

    switch(destrel.domains[arg.bits].type) {
            case BOOLEAN:
                    bm = actBoolCell(arg);
                    if (bm == BOOL_DK)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case SHORT:
             case INTEGER:
                    im = actIntCell(arg);
                    if (im == INT_DK)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case LONG:
                    lm = actLongCell(arg);
                    if (lm == LONG_DK)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case FLOAT:
             case DOUBLE:
                    dm = actDoubleCell(arg);
                    if (dm == DOUBLE_DK)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case ATTRIBUTE:
             case STRING:
                    sm = actStrCell(arg);
                    if (sm.compareTo(STRING_DK) == 0)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case NUMERIC:
                    nm = actNumCell(arg);
                    if(nm.compareTo(NUMERIC_DK) == 0)
                            return BOOL_TRUE;
                    else
                            return BOOL_FALSE;
             case IDLIST:
             default:
                    Utility.dump(arg, "actBoolCell ERROR=>");
                    throw new InterpretError("nulldk(" + getType(arg) + "):
                                            to be implemented");
            }

        }
```

Figure 4.17: Method *isnulldk()*

```
private String getType(SimpleNode node)
{
            String typeName = "";
            switch (node.type)
            {
                case BOOLEAN:
                  typeName = "boolean"; break;
                case SHORT:
                   typeName = "short"; break;
                case INTEGER:
                   typeName = "int"; break;
                case LONG:
                    typeName = "long"; break;
                case FLOAT:
                    typeName = "float"; break;
                case DOUBLE:
                    typeName = "double"; break;
                case NUMERIC:
                    typeName = "numeric"; break;
                case STRING:
                    typeName = "string"; break;
                case TEXT:
                    typeName = "text"; break;
                case STMT:
                    typeName = "stmt"; break;
                case EXPR:
                    typeName = "expr"; break;
                case COMP:
                     typeName = "comp"; break;
                case IDLIST:
                     typeName = "IDList"; break;
                case DC:
                     typeName = "DC"; break;
                case DK:
                     typeName = "DK"; break;
                case RELATION:
                     typeName = "relation"; break;
                case VIEW:
                     typeName = "view"; break;
                case COMPUTATION:
                     typeName = "computation"; break;
                default:
                     typeName = "type number: " + node.type;
                     break;
            }
            return typeName;
}
```

Figure 4.18: Method *getType()*

Chapter 5

Clifford Algebra & Clifford ADT

5.1 Introduction to Clifford Algebra

Clifford Algebra is a type of associative algebra in mathematics, named after English geometer William Clifford. It provides a complete mathematical representation of geometric notions of direction and magnitude, thus making itself a powerful tool to describe the physical world [Lou01], and leading to large amount of useful applications in many different fields including geometry, theoretical physics, engineering, computer vision, robotics, navigation, space flight, etc. [AbF00]. Accompanying with Graβmann-Cayley algebras, Clifford algebras broaden the views of a lot of fields. Automatic theorem proving is related to Clifford algebras; A specific Clifford algebra called "Deformed Clifford algebra" is used to solve problems in quantum field theory. And Clifford algebra will certainly play a major role in quantum computing and the design of quantum computers [AbF00].

One of the important features of Clifford algebra, which distinguishes itself from other ways to represent geometric concepts, is that this algebra provides a complete coordinate-free notation to describe space, and hence it can represent any dimensional space in a generic way and provide unified formalism [Mer05].

In Clifford algebra, a d-dimensional space is composed of elements including points, edges, faces, volumes, and so on, with each element itself a k-dimensional space with elements ranging from points(0), edges(1), faces(2), volumes(3) up to hypervolume(d), and is represented by the linear combination of these components. Therefore the total

number of elements in a d-dimensional space is 2^d [Mer05]. For example, a 2-dimensional space is composed of elements $\{1, e_1, e_2, e_{12}\}$, and the total number of elements are $2^d = 2^2 = 4$, while a 3-dimensional space has basic elements $\{1, e_1, e_2, e_3, e_{12}, e_{23}, e_{13}, e_{123}\}$ and the total number of elements is $2^d = 2^3 = 8$.

There are two operations in Clifford algebra: addition and multiplication. Clifford algebras are associative under addition and multiplication, and they are commutative under addition but anti-commutative under multiplication. This means that given Clifford elements x, y and z, we have x+(y+z) = (x+y)+z and x+y=y+x under addition and x*(y*z)=x*(y*z) under multiplication, but x*y≠y*x under multiplication. The reason that Clifford algebras are not commutative under multiplication operation is that Clifford algebra integrates direction information in its expression, and thus x*y and y*x represent elements with different directions and are not equivalent.

Although unlimited number of dimensional space could be represented in Clifford algebra, the 2-dimensional space will be used to introduce the multiplication operation in Clifford algebra and the corresponding geometrical interpretation of the operation.

Elements in Clifford algebra Cl_2

The Clifford algebra Cl_2 represents a 2-dimensional space R^2. As mentioned above, the following elements

1	point
e_1, e_2	edges
e_{12}	faces

form the basis elements for the Clifford algebra Cl_2, and an arbitrary element in Cl_2 is represented as: $u = u_0 + u_1e_1 + u_2e_2 + u_{12}e_{12}$ which is a linear combination of a point, edges and a face [Lou01]. The geometrical meanings of these elements are shown in Figure 5.1, where e_1 and e_2 are orthonormal components, $u_1e_1 + u_2e_2$ (with u_1=cos θ and u2=sin θ) represents any arbitrary edge with angle θ, and $u_{12}e_{12}$ represents the oriented plane area of the square with edges e_1 and e_2. e_{12} is short for the production of e_1 and e_2, i.e. $e_{12} = e_1\ e_2$. As mentioned before, multiplication is not commutative in Clifford algebra, which means $e_1\ e_2 \neq e_2\ e_1$, i.e. $e_{12} \neq e_{21}$. This is due to the fact that e_{12} represents a plane with direction towards outside as shown in Figure 5.1, while e_{21} represents a plane with direction towards inside. The relation between e_{12} and e_{21} is $e_{12} = -e_{21}$.

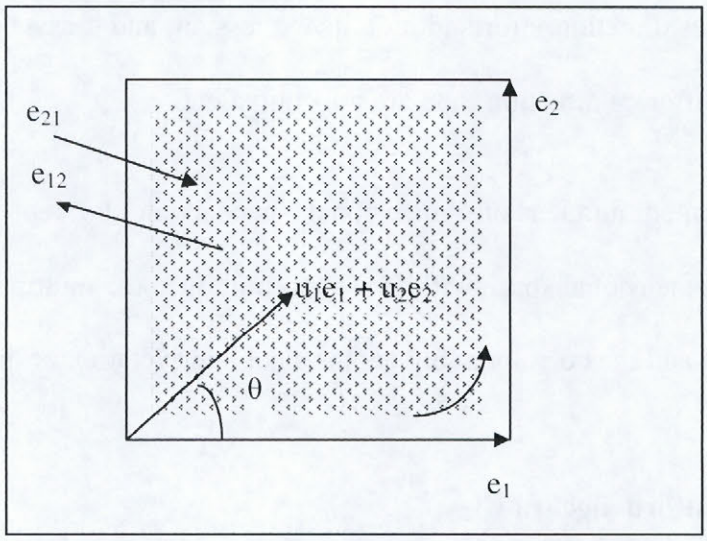

Figure 5.1: Clifford elements in 2-demential space

Multiplication in Clifford algebra

Similar to the coordinate form, it is defined in Clifford algebra that any edge v produces itself, i.e. the square of this edge is equal to the square of the length of this edge: $v^2 = |v|^2$.

Therefore, for normalized orthonormal edges e_1 and e_2, we have $e_1^2=|e_1|^2=1$ and $e_2^2=|e_2|^2=1$. For an arbitrary edge $u_1e_1 + u_2e_2$ in the Clifford algebra Cl_2, $(u_1e_1 + u_2e_2)^2 = u_1^2e_1^2 + u_1 u_2e_{12} + u_1 u_2e_{21} + u_2^2e_2^2 = u_1^2 + u_2^2$.

Multiplication is a very common operation in the Clifford algebra. A lot of geometrical operations such as rotation and reflection could be done by multiplying certain factors. For example, e_{12} is a right angle rotation factor, meaning that the result of postmultiplying e_{12} with any edge v is an edge generated by counterclockwise rotating v through right angle, and premultiplying e_{12} with any edge v is equivalent to clockwise rotating v through right angle. Furthermore, $u_1 + u_2e_2$ with $u_1=\cos\theta$ and $u_2=\sin\theta$ is a rotation factor for rotating any edge through angle θ. If θ is positive, any edge multiplied by this factor will be rotated counterclockwise through θ. On the other hand, if θ is negative, the edge multiplied by this factor will be rotated clockwise through θ. Moreover, given normalized edges u and v, uvu is the reflection of v in u. [Mer05]

In the following section, how to use *CliffordADT* to calculate the addition and multiplication will be discussed in details.

5.2 User Manual of Clifford ADT

5.2.1 Introduction to the Usage of Clifford ADT

CliffordADT is a predefined stateless Abstract Data Type (ADT) which provides methods for supporting basic operations such as plus, subtract and product in Clifford algebra. The definition of *CliffordADT* is shown in Figure 5.13. This definition is based on [Mer052]

with some modifications so that it can be supported by the current version of JRelix. *CliffordADT* is stateless, meaning that it does not have state variables and the previous invocation of methods in it will not affect the result of the current invocation of the methods. There are two public methods defined in *CliffordADT*. One is *Add()* and the other is *Product()*. Method *Add()* requires three parameters, which are *cliffordL*, *cliffordR* and *Clifford*. Depending on details of the two input parameters and the output parameter when the method is invoked, either plus or subtract operation of Clifford algebra will be performed respectively. Method *Product()* also requires the three parameters *cliffordL*, *CliffordR* and *Clifford* when it is invoked. As indicated by the name, it performs the product operation in Clifford algebra.

Before the usage of *CliffordADT* is described in detail, how Clifford algebra is represented in *CliffordADT* must be discussed. In *CliffordADT*, the Clifford element is represented by a two – level nested relation domain with *coeff* and *cliff* as attributes. *Coeff* is a real scalar for representing the coefficient of the Clifford element; *Cliff* is defined as *cliff(index),* which contains an attribute *index* of type integer used to store the indices of the Clifford element. It has been assumed that the indices are in ascending order in the Clifford element [Mer052]. Examples of representing a Clifford element in *CliffordADT* are shown in Figure 5.2.

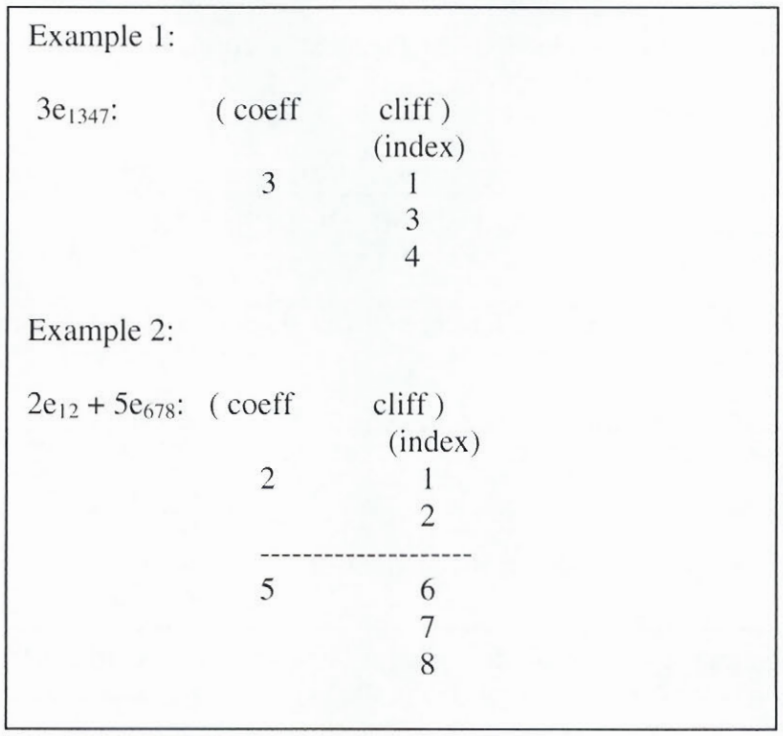

Figure 5.2: Examples of representing Clifford elements in CliffordADT

After the declarations of *Clifford ADT*, the Aldat code shown in Figure 5.3 must be performed before *Add*() and *Product*() could be invoked.

```
CliffordADT(out Add, out Product);
```

Figure 5.3: Code for making *Add()* and *Product()* available.

Method *Add()* and *Product()* have the same parameters *cliffordL*, *cliffordR* and *Clifford*. Among them, *CliffordL* and *cliffordR* are used to represent the left operand and right operand respectively, and *Clifford* is used to represent the result of the operation. There are three ways to invoke *Add()* as illustrated in Figure 5.4. Depending on the input and output relations, *clifford*, *cliffordL*, and *cliffordR* could be returned. For method

Product(), there is only one way to invoke this method, as seen in Figure 5.5, which returns the result of the production. In the next section, examples will be given to illustrate the usage of *CliffordADT*.

Invocation 1: return Clifford <- CliffordL + CliffordR

```
Add (in CliffordL, in CliffordR, out Clifford)
```

Invocation 2: return the difference CliffordL <- Clifford - CliffordR

```
Add (out CliffordL, in CliffordR, in Clifford)
```

Invocation 3: return the difference CliffordR <- Clifford - CliffordL

```
Add (in CliffordL, out CliffordR, in Clifford)
```

Figure 5.4: Invocations of Method *Add()*

Invocation: return the product Clifford <- CliffordL * CliffordR

```
Product (in CliffordL, in CliffordR, out Clifford)
```

Figure 5.5: Invocation of Method *Product()*

There is a requirement when using *CliffordADT*, that is indices in *cliff* must be in ascending order. E.g. e_{34} is allowed while e_{43} is not. Also, there is a limitation in the current implementation of *CliffordADT*, that is a Clifford element multiplies itself is not supported by *CliffordADT*. E.g. $e_{12} \times e_{12}$ is not allowed.

5.2.2 Examples

Example 1: using Add()

This first example shows how to use method *Add()* to calculate the several expressions in Clifford algebra. First for the expression $(e_{12}+3e_{23}) + e_{23}$, input parameter *cliffordL* $e_{12}+3e_{23}$ and *cliffordR* e_{23} are represented by relation *leftopd* and *rightopd* respectively.

The sum *clifford* is represent by relation *aSum*, and the calculation of it is shown in Figure 5.6. The result relation *aSum* represents Clifford element ($e_{12}+4e_{23}$). Note that when *Add()* is invoked, *leftopd* and *rightopd* are set as input parameters and *asum* as output parameter. Also, since *cliff* is a nest relation domain, the values shown in the result are surrogates instead of the real values. The real values that these surrogates represent are in *.cliff*.

```
>CliffordADT(out Add, out Product);
>relation leftopd (coeff, cliff) <- {(1.0, {(1), (2)}),
                                     (3.0,{(2),(3)})};
>relation rightopd (coeff, cliff) <- {(1.0, {(2), (3)})};
>Add(in leftopd, in rightopd, out asum);
>pr asum;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| 1.0           | 7                    |
| 4.0           | 8                    |
+---------------+----------------------+
relation asum has 2 tuples

> pr.cliff;

+----------------------+-------------+
| .id                  | index       |
+----------------------+-------------+
| 1                    | 1           |
| 1                    | 2           |
| 2                    | 2           |
| 2                    | 3           |
| 3                    | 2           |
| 3                    | 3           |
| 7                    | 1           |
| 7                    | 2           |
| 8                    | 2           |
| 8                    | 3           |
+----------------------+-------------+
```

Figure 5.6: Using method *Add()* to calculate the sum.

Given *cliffordL* ($e_{12}+3e_{23}$) and the sum *clifford* ($e_{12}+4e_{23}$), we can calculate *cliffordR*, as shown in Figure 5.7. The result relation *rightopd* is interpreted as e_{23}. Note that when

Add() is invoked, the input parameters are *leftopd* and *asum* and output parameter is *rightopd*. In the result the value for *cliff* is 16, which is the surrogate value. The real value represented by this surrogate is shown in Figure 5.12.

```
>Add(in leftopd, out rightopd, in asum);
>pr rightopd;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| 1.0           | 16                   |
+---------------+----------------------+
```

Figure 5.7: Calculating the right operand by using method *Add()*

On the other hand, given *cliffordR* e_{23} and *clifford* ($e_{12}+4e_{23}$), the way using method *Add()* to calculate *cliffordL* is shown in Figure 5.8. The result relation *leftopd* represents Clifford element ($e_{12}+3e_{23}$). Similar to the above example, the values for *cliff* are surrogates. The real values these surrogates represent are given in Figure 5.12.

```
>Add(out leftopd, in rightopd, in asum);
>pr leftopd;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| 1.0           | 22                   |
| 3.0           | 23                   |
+---------------+----------------------+
```

Figure 5.8: Calculating the left operand by using method *Add()*

Example 2: using Product()

This example will show how to use method *Product()* to calculate product expressions in Clifford algebra. Given an Clifford expression: $(e_1+3e_2)*(e_{12}+2e_{23}+e_{13})$, similar to the

above example, the left operand (e_1+3e_2) will be represented by a relation *lopd* shown in Figure 5.9 and the right operand ($e_{12}+2e_{23}+e_{13}$) will be represented by relation *ropd* shown in Figure 5.10. The result of the product will be represented by relation *aproduct*. To invoke method *Product()*, relation *lopd* and *ropd* will be the input parameters and *aproduct*, which is the result of the production, will be the output parameter. The codes of the invocation are listed in Figure 5.11, with the result relation *aproduct* representing a Clifford element ($-3e_1-e_{123}+e_2+7e_3$). Similar to the above example, the values of *cliff* in the relations are surrogates, and the corresponding real values are shown in Figure 5.12.

```
>relation lopd (coeff, cliff) <- {(1.0, {(1)}),
(3.0,{(2)})};
>pr lopd;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| 1.0           | 24                   |
| 3.0           | 25                   |
+---------------+----------------------+
```

Figure 5.9: Representing (e_1+3e_2) by relation *lopd*

```
>relation ropd (coeff, cliff) <- {(1.0, {(1), (2)}),
(2.0, {(2),(3)}),(1.0, {(1),(3)})};
>pr ropd;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| 1.0           | 26                   |
| 1.0           | 28                   |
| 2.0           | 27                   |
+---------------+----------------------+
```

Figure 5.10: Representing ($e_{12}+2e_{23}+e_{13}$) by relation *ropd*

```
>Product( in lopd, in ropd, out aproduct);
>pr aproduct;

+---------------+----------------------+
| coeff         | cliff                |
+---------------+----------------------+
| -3.0          | 40                   |
| -1.0          | 39                   |
| 1.0           | 37                   |
| 7.0           | 42                   |
+---------------+----------------------+
```

Figure 5.11: Codes invoking method *Product()*

```
>pr .cliff

+----------------------+-------------+
| .id                  | index       |
+----------------------+-------------+
| 1                    | 1           |
| 1                    | 2           |
| 2                    | 2           |
| 2                    | 3           |
| 3                    | 2           |
| 3                    | 3           |
| 7                    | 1           |
| 7                    | 2           |
| 8                    | 2           |
| 8                    | 3           |
| 13                   | 1           |
| 13                   | 2           |
| 14                   | 2           |
| 14                   | 3           |
| 15                   | 1           |
| 15                   | 2           |
| 16                   | 2           |
| 16                   | 3           |
| 20                   | 1           |
| 20                   | 2           |
| 21                   | 2           |
| 21                   | 3           |
| 22                   | 1           |
| 22                   | 2           |
| 23                   | 2           |
| 23                   | 3           |
| 24                   | 1           |
| 25                   | 2           |
| 26                   | 1           |
| 26                   | 2           |
| 27                   | 2           |
| 27                   | 3           |
| 28                   | 1           |
| 28                   | 3           |
| 37                   | 2           |
| 38                   | 3           |
| 39                   | 1           |
| 39                   | 2           |
| 39                   | 3           |
| 40                   | 1           |
| 41                   | 1           |
| 41                   | 2           |
| 41                   | 3           |
| 42                   | 3           |
+----------------------+-------------+
```

Figure 5.12: Relation *.cliff*

5.3 Implementation of Clifford ADT

The implementation of Clifford ADT (*CliffordADT*), as derived from T.H. Merrett's unpublished note "Aldat code for Clifford algebra" [Mer052], is shown in Figure 5.13. Details for the algorithm used by *CliffordADT* are given in Ref. [Mer052]. In this section, the focus will be put on the modification of the algorithm.

```
 1      domain coeff float;
 2      domain index intg;
 3      domain cliff (index);
 4      domain cliffordL (coeff, cliff);
 5      domain cliffordR (coeff, cliff);
 6      domain clifford (coeff, cliff);
 7      domain Add comp(cliffordL,cliffordR, clifford);
 8      domain Product comp(cliffordL, cliffordR, clifford);
 9      comp CliffordADT (Add, Product) is
10      {
11         comp Add(cliffordL,cliffordR, clifford) is
12        {
13         let coeffL be coeff;
14         let cliffL be cliff;
15         cliffordL' <- [coeffL, cliffL] in cliffordL;
16         let coeffR be coeff;
17         let cliffR be cliff;
18         cliffordR' <- [coeffR, cliffR] in cliffordR;
19         clifford' <- cliffordL'[ cliffL:ujoin:cliffR]cliffordR';
20         let coeffL' be if isnulldc(coeffL) then 0.0 else coeffL;
21         let coeffR' be if isnulldc(coeffR) then 0.0 else coeffR;
22         let coeff be coeffL'+coeffR';
23         let cliff be cliffL;
24         clifford <- [coeff, cliff] in clifford';
25      }alt
26      {
27         let coeffL be coeff;
28         let cliffL be cliff;
29         cliffordL' <- [coeffL, cliffL] in cliffordL;
30         let coeff' be coeff;
31         let cliff' be cliff;
32         clifford' <- [coeff', cliff'] in clifford;
33         cliffordR' <- cliffordL'[cliffL:ujoin:cliff']clifford';
34         let coeffL' be if isnulldc(coeffL) then 0.0 else coeffL;
35         let coeff'' be if isnulldc(coeff') then 0.0 else coeff';
36         let coeff be coeff''- coeffL';
37         let cliff be cliffL;
38         cliffordR <- where coeff != 0.0 in ([coeff, cliff] in
                        cliffordR');
39      }alt
```

(continue next page)

```
40      {
41        let coeffR be coeff;
42        let cliffR be cliff;
43        cliffordR' <- [coeffR, cliffR] in cliffordR;
44        let coeff' be coeff;
45        let cliff' be cliff;
46        clifford' <- [coeff', cliff'] in clifford;
47        cliffordL' <- cliffordR'[cliffR:ujoin:cliff']clifford';
48        let coeffR' be if isnulldc(coeffR) then 0.0 else coeffR;
49        let coeff'' be if isnulldc(coeff') then 0.0 else coeff';
50        let coeff be coeff'' - coeffR';
51        let cliff be cliffR;
52        cliffordL <- where coeff != 0.0 in ([coeff, cliff] in cliffordL');
53      };
54
55      comp Product(cliffordL, cliffordR, clifford) is
56      {
57        let coeffL be coeff;
58        let cliffL be cliff;
59        cliffordL' <- [coeffL, cliffL] in cliffordL;
60        let coeffR be coeff;
61        let cliffR be cliff;
62        cliffordR' <- [coeffR, cliffR] in cliffordR;
63        let seqR be fun + of 1 order index;
64        let seqLi be -(fun + of 1 order index);
65        let seqL be (red + of 1) + 1 - (fun + of 1 order index);
66        let sind be fun + of 1 order index;
67        let tempd1 be if isnulldc(seqR) then seqLi else seqR;
68        let s be fun + of 1 order tempd1;
69        let seqdiff be if (sind - s) < 0 then 0 else (sind - s);
70        let cliffLR be ([index,seqLi] in cliffL) sjoin
                         ([index,seqR] in cliffR);
71        let cliff be [index] in cliffLR;
72        let tempd2 be if seqdiff < 0 then 0 else seqdiff;
73        let evodsjoin' be [red + of tempd2] in cliffLR;
74        let evodsjoin be evodsjoin' mod 2;
75        let evodinvert' be [red + of 1] in cliffL;
76        let evodinvert be (evodinvert' / 2) mod 2;
77        let tempd3 be seqR - seqLi;
78        let cliffLRijoin be ([index,seqLi] in  cliffL) ijoin
                              ([index,seqR] in cliffR);
79        let tempd4 be [red + of tempd3] in cliffLRijoin;
80        let evodijoin' be if ([] in cliffLRijoin) then tempd4 else 0;
81        let evodijoin be evodijoin' mod 2;
82        let amark be if (evodsjoin+evodinvert+evodijoin)mod 2 = 0
                       then 1 else -1;
83        let coeffLR be coeffL*coeffR*amark;
84        let coeff be equiv + of coeffLR by cliff;
85        clifford' <- [coeffLR, cliff] in (cliffordL' ijoin cliffordR');
86        clifford <- [coeff, cliff] in clifford';
87      };
88
89    };
```

Figure 5.13: Definition of *CliffordADT*

Method Add()

There are 3 blocks of codes in this method, as shown in Figure 5.13. Codes in different blocks will be triggered according to different input and output parameters. The first block of codes will be performed if input parameters are *cliffordL* and *cliffordR* and output parameter is *clifford*, with the output result *clifford* equal to *cliffordL* + *cliffordR*. If the input parameters are *cliffordL* and *clifford*, and output parameter is *cliffordR*, the second block of codes will be triggered and the output result *cliffordR* is equal to *clifford* – *cliffordL*. The third block of codes will be performed for the input parameters given by *cliffordR* and *clifford* and the output parameter by *cliffordL*. The output result *cliffordL* is equal to *clifford* – *cliffordR*. The first block of codes is illustrated in details in Figure 5.14, and the other two blocks of codes are similar to the first one.

In the codes shown in Figure 5.14, the attributes of input parameters *cliffordL* and *cliffordR* are renamed, and then *cliffordL* and *cliffordR* themselves are renamed to *cliffordL'* and *cliffordR'* through assignments. After performing union join on *cliffordL'* and *cliffordR'*, the coefficients for *cliffL* and *cliffR*, which have the same real values, will be put in the same tuple as shown in Figure 5.15.

```
13          let coeffL be coeff;
14          let cliffL be cliff;
15          cliffordL' <- [coeffL, cliffL] in cliffordL;
16          let coeffR be coeff;
17          let cliffR be cliff;
18          cliffordR' <- [coeffR, cliffR] in cliffordR;
19          clifford' <- cliffordL'[ cliffL:ujoin:cliffR]cliffordR';
20          let coeffL' be if isnulldc(coeffL) then 0.0 else coeffL;
21          let coeffR' be if isnulldc(coeffR) then 0.0 else coeffR;
22          let coeff be coeffL'+coeffR';
23          let cliff be cliffL;
24          clifford <- [coeff, cliff] in clifford';
```

Figure 5.14: First block of codes in Method *Add()*

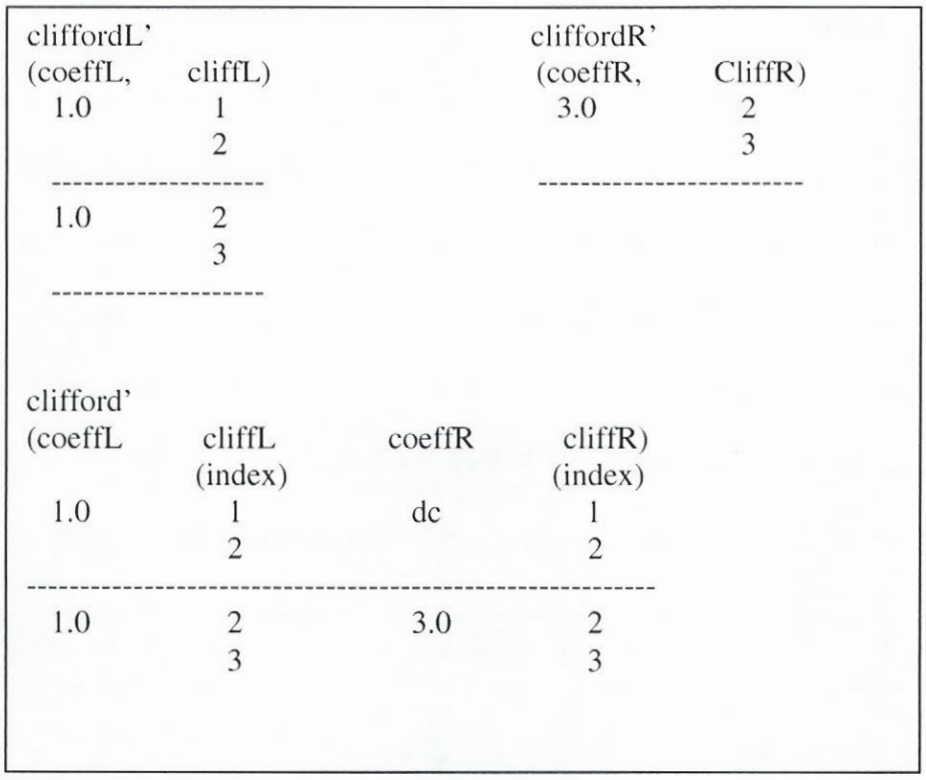

Figure 5.15: Example of method *Add()*

The virtual domains *coeffL* and *coeffR*, as defined in Figure 5.14, may have *dc* value, as for the example shown in Figure 5.15. It causes problems when *coeff*, which is equal to *coeffL+coeffR*, is calculated. To solve the problem, in Line 20 and 21 of Figure 5.14 two virtual domains *coeffL'* and *coeffR'* are defined, that is,

```
20        let coeffL' be if isnulldc(coeffL) then 0.0 else coeffL;
21        let coeffR' be if isnulldc(coeffR) then 0.0 else coeffR;
```

and Boolean function *isnulldc* is used in the definition to detect *dc* value. After the actualization, values of *coeffL'* and *coeffR'* will be the same as those of *coeffL* and *coeffR* except that the *dc* value is replaced by *0.0*, and then *coeffL'* and *coeffR'* are used to calculate *coeff*. At the end of the codes, attributes in *clifford'* are renamed back to *coeff* and *cliff* and assigned to the output parameter *clifford*.

Method Product()

The codes for method *Product()* are shown in Figure 5.13. Similar to the method *Add()*, first the input parameter *cliffordL* and its attributes *coeff* and *cliff* are renamed as *cliffordL'*, *coeffL* and *cliffL*. Another parameter *clifffordR* and its attributes *coeff* and *cliff* are renamed as *cliffordR'*, *coeffR*, and *cliffR*. Then the three factors *evodsjoin*, *evodinvert* and *evodijoin* which determine the final sign of the result are calculated. (The algorithm to calculate these factors is described in Ref. [Mer052].) In code line 67, virtual domain *tempd1* is introduced to remove *dc* value by using Boolean function *isnulldc*:

```
67        let tempd1 be if isnulldc(seqR) then seqLi else seqR;
```

Then in line 73, the concept of Red Scalar is applied:

```
73      let evodsjoin' be [red + of tempd2] in cliffLR;
74      let evodsjoin be evodsjoin' mod 2;
```

It means that evodsjoin' will not be a nest relation virtual domain; instead, it is a scalar value which could be used in mathematical operations such as +, -, *, /, mod, etc., as shown in code line 74. Similar to *evodsjoin'*, the definitions of *evodinvert'* and *tempd4* (see lines 75 and 79) also indicate that they are scalar values.

```
75          let evodinvert' be [red + of 1] in cliffL;
79          let tempd4 be [red + of tempd3] in cliffLRijoin;
```

The factor *evodijoin* is calculated in the codes below. In line 80, depending on whether the nested virtual domain *cliffLRijion* is empty or not, *evodijoin'* is given different values. This test condition must be added since *cliffLRijoin* may be empty and in that case, if *evodijoin'* is defined as *let evodijoin' be tempd4*, during the actualization of *evoidijoin'*

an error will be generated when the system tries to actualize *tempd4* on which *evodijoin'* is defined.

```
78   let cliffLRijoin be ([index,seqLi] in  cliffL) ijoin
                         ([index,seqR] in cliffR);
79   let tempd4 be [red + of tempd3] in cliffLRijoin;
80   let evodijoin' be if ([] in cliffLRijoin) then tempd4 else 0;
81   let evodijoin be evodijoin' mod 2;
```

In line 84, the equivalence reduction group by nested relation is used in the definition of virtual domain *coeff*.

```
84         let coeff be equiv + of coeffLR by cliff;
```

Here *coeffLR* is summed up according to the real value of the nested relation domain *cliff*.

Chapter 6

Summary

The purpose of this project is to build an Abstract Data Type (ADT) *CliffordADT* for Clifford algebra and provide language support for it. As a result of this project, the following new features have been added into JRelix system:

Vertical Domain Algebra operations have been extended in the following aspects:

- Equivalence reduction has been extended to support group by nested relation domain, so that the real values of nested relation domain that the surrogates represent, instead of the surrogates themselves, are compared. This ensures that equivalent values of nested relation domain are detected and grouped correctly.
- Functional mapping has been extended to support order by nested relation domain. The system has been extended to use the real values of nested relation domain to order tuples, instead of using the surrogates which represent the real values.
- Partial functional mapping has been extended to support group and order by nested relation domain. Real values of nested relation domain are used in group and order tuples, which ensures that partial functional mapping is correctly performed on nested relation domain.

Also, three new Boolean functions have been implemented:

- Function *isnulldc* has been implemented to test whether a value of an attribute is *dc* or not. This function returns Boolean value *true* or *false* depending on whether the value is *dc* or not;
- Function *isnulldk* has been implemented to test whether a value of an attribute is *dk* or not. Boolean value *true* or *false* is returned according to whether the value is *dk* or not;
- Function *isnull* has been implemented to test whether a value of an attribute is *dc* or *dk*. It returns *true* or *false* respectively.

These new features added in JRelix provide support to the creation of the Abstract Data Type *CliffordADT*. *CliffordADT* is a stateless ADT and provides two methods *Add()* and *Product()* to support the addition, subtraction and product operations in Clifford algebra. Depending on the input and output parameters, method *Add()* could perform addition and subtraction in Clifford algebra. Another method *Product()* is used for calculating the results of multiplication operations in Clifford algebra. Since the parameters of these two methods are the same, the output of one method could be used as input of another method, which ensures that complicated Clifford algebra operations could be performed by this ADT.

However, there is a limitation in method *Product()*: pairing products, e.g., $e_{12}*e_{12}$ are not supported in the current version of *CliffordADT*. Future works should be done to remove such limitation.

Bibliography

[AbF00] R. *Abfamowicz and B. Fauser (eds), Clifford Algebras and their Applications in Mathematical Physics,* Volume 1. Birkhäuser Boston, 2000.

[Bak98] P. Baker. *Design and Implementation of Database Computations in Java.* Master's thesis, McGill University, Montreal, 1998.

[Hao98] Biao Hao. *Implementation of the Nested Relational in Java.* Master's thesis, McGill University, Montreal, 1998.

[Kan01] Sung Soo Kang. *Implementation of Functional Mapping in Nested Relation Algebra.* Master's project report, McGill University, Montreal, 2001.

[Lou01] P. Lounesto. *Clifford Algebras and Spinors,* Second Edition. Cambridge University press, 2001.

[Mer84] T. H. Merrett. *Relations as programming language elements.* Reston Publishing Co. Reston, VA, 1984.

[Mer05] T. H. Merrett. *Clifford Algebra in Two and Three Dimensions.* Unpublished note for CS 612 "Information Systems", McGill University, Montreal, 2005.

[Mer052] T.H. Merrett. *Aldat code for Clifford algebra.* Unpublished note for CS 612 "Information Systems", McGill University, Montreal, 2005.

[Yu04] Zhan Yu. *Implementation of Recursively Nested Relation of JRelix.* Master's project report, McGill University, Montreal, 2004.

[Yua98] Zhongxia Yuan. *Java Implementation of the Nested Domain Algebra in a Database Programming Language.* Master's thesis, McGill University, Montreal, 1998.

[Zhe02] Yi Zheng. *Abstract Data Types and Extended Domain Operations on Nested Relation Algebra*. Master's thesis, McGill University, Montreal, 2002.