A Proof-of-Concept for Using PVS and Maxima to Support Relational Calculus

A Proof-of-Concept for Using PVS and Maxima to Support Relational Calculus

By
Huong Thi Thu Nguyen, B.Sc

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2006) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: A Proof-of-Concept for Using PVS and Maxima to Support Relational Calculus

AUTHOR: Huong Thi Thu Nguyen, B.Sc (Hanoi National University)

SUPERVISOR: Dr. Ridha Khedri

NUMBER OF PAGES: xiii, 91

Abstract

Mechanized mathematics systems, especially Theorem Provers (TP) and Computer Algebra Systems (CAS), can play a very helpful role in handling relational calculus. Computer Algebra Systems help to automate tedious symbolic computations. However, they lack the ability to make sophisticated derivations of logical formulas. Correspondingly, a Theorem Prover is powerful in deriving the truth-value of a logical formula. Nevertheless, it is not suitable for dealing with symbolic expressions.

The main goal for our research is to investigate the automation of relational calculus using existing mechanized mathematics technologies. Particularly, we elaborated a heuristic that enables the assignment of tasks to PVS and Maxima to help perform relational calculus. As well we built a proof-of-concept tool that supports this calculus.

To fulfill our objective, we adopted the following steps:

1. Investigated and evaluated the characteristics and capabilities of TPs and CASs. This step led us to select PVS and Maxima as the tools to be used by our system.

2. Explored a strategy that governs setting tasks to PVS and Maxima in order to perform relational calculus. Then, we propose a task assignment heuristic based on this strategy.

3. Designed and built a proof-of-concept tool that makes use of PVS and

Maxima to help perform relational calculus.

4. Assessed our tool by using it to handle some illustrative examples of operations on concrete relations.

In our work, relations are given by their characteristic predicates. We assume as well that predicates that are provided to our proof-of-concept tool are in a Disjunctive Normal Form. We adopt a linear notation for the representation of propositions, quantifications, and expressions. We fall short of providing a user interface, which makes the use of the tool that we built slightly difficult.

Acknowledgements

First and foremost, I would like to express my thankfulness and gratitude to my supervisor, Dr. Ridha Khedri, for his wise advice, valuable guidance and support, and continuous encouragement throughout the development of this thesis. Without his guidance and support, I would not have been able to complete this thesis.

In addition, I greatly appreciate the thoughtful comments from Dr. Kamran Sartipi and Dr. William M. Farmer serving as the committee members of my defense.

I would like to send my special thanks to Khair Eddin Sabri for his patient discussions and helpful comments on my research documents.

An extended thanks to Ms. Laurie LeBlanc, she has made me feel very welcome with her kind smile and willingness to help whenever I shown up at her office to ask for her help.

Last, but not least, special thanks to my parents for their love, encouragement, and support during my research study. This thesis dedicates to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of a relation plays an important role in many areas of computer science such as program semantics, graph theory, relational database, and logic programming [SS93].

Relational algebra is a mathematical structure that involves a set of relations to which we associate a set of operators. These operators include *union*, *intersection*, *complement*, *composition*, and *inverse*. For more details on relational algebra, we refer readers to Section 2.2.1.

There are many tools built to handle relational algebra and relation algebra (the first is a model of the second). Some of them are introduced in **RelMiCS** (**Rel**ation **M**ethods **i**n **C**omputer **S**cience) site [Rel06], such as RALF [KH98, Hat98], RELVIEW [BBMS98], and RALL [vOG97].

Relational algebra is used to express mathematical problems which can be solved by using mechanized mathematics systems. Mechanized mathematics is the study of how computer can be used to support, improve, and automate the mathematical reasoning process [FvM00]. There are two main classes of mechanized mathematics systems: *Theorem Provers* (TPs) and *Computer Algebra Systems* (CASs).

A TP provides mechanized support for proving conjectures using axiomatic theory. Mathematical expressions are manipulated using a fixed set of inference rules [Rus03]. An axiomatic theory is used to describe a collection of mathematical models which have similar structures [FvM00]. A TP is wide in scope and uses rigorous mathematics, but it provides little support for computation [FvM00]. We refer readers to Section 3.1 for more details on TPs.

A CAS is a software package that is designed to compute mathematical formulas. The principle purpose of a CAS is to automate tedious and difficult symbolic manipulation problems. It handles symbolic expressions and it is relatively easy to use, but it does not always provide reliable results. Besides, it does not offer conjecture proving since its mathematical knowledge is represented algorithmically [FvM00]. We elaborate with more details on CASs in Section 3.2

In our research, we aim at investigating means to demonstrate the feasibility of mechanizing relational calculus using available mechanized mathematics technologies.

1.1 Related tools

There are some tools that can be colligated to our research. The relationship is either from the perspective of using relation algebras or from the perspective of using a combination of CASs and TPs. We classify them into two classes: Tools that handle relation algebras and tools that use a combination of CASs and TPs. Subsections 1.1.1 and 1.1.2 discuss these two classes, respectively.

1.1.1 Tools that handle relation algebras

RALF (**R**elation **AL**gebraic **F**ormula Manipulation) [Hat98, KH98] is an interactive assistant proof system for relation-algebraic formulas. The main purpose

of RALF is to provide proofs for manipulating relation-algebraic reasonings in a calculational style, i.e., the proof can be represented as a sequence of transformations.

In RALF, the proof strategy is backward reduction, i.e., the system starts from the input theorem needed to be proved, applies the given rules, and derives the proof step by step to reach the valid theorems. The given rules can be meta, transformation, and inference rules [KH98] The system presents a formula as a graphical tree, in which the rewritten sub-formulas are chosen by mouse clicks.

The following is an illustrative example which is borrowed from [Hat98]. It presents the proof steps of the theorem $R \sqsubseteq \mathbb{I} \rightarrow R = R;R$ which states that when a relation is a subset of the identity, then it is equal to its relational composition with itself. This input theorem is put at the root of the graphical tree. In the derivation of the proof, the bold-faced expressions indicate the sub-expression which will be transformed in the next proof step. This corresponds to the selected sub-tree of the formula tree in the system. After applying transformation rules or meta rules step by step, it reduces to the trees corresponding to the universally valid theorems $R;R \sqsubseteq R \Rightarrow R;R \sqsubseteq R$ and $R \sqsubseteq \mathbb{I} \Rightarrow R \sqsubseteq R$.

$$\dfrac{\dfrac{\dfrac{R;R \sqsubseteq R \Rightarrow R;R \sqsubseteq R}{R;R \sqsubseteq \mathbf{R};\mathbb{I} \Rightarrow R;R \sqsubseteq R}}{\mathbf{R} \sqsubseteq \mathbb{I} \Rightarrow R;R \sqsubseteq R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{R \sqsubseteq \mathbb{I} \Rightarrow R \sqsubseteq R}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R} \sqcap \mathbf{R};\mathbb{I}^{\smile} \sqsubseteq R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{R \sqsubseteq \mathbb{I} \Rightarrow R \sqsubseteq R}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbb{I};\mathbf{R} \sqsubseteq R}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbb{I}^{\smile};R \sqsubseteq R}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R}^{\smile};R \sqsubseteq R}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbb{I} \sqcap \mathbf{R}^{\smile};\mathbf{R} \sqsubseteq R}}{R \sqsubseteq \mathbb{I} \Rightarrow (\mathbf{R} \sqcap \mathbf{R};\mathbb{I}^{\smile})(\mathbb{I} \sqcap \mathbf{R}^{\smile};\mathbf{R}) \sqsubseteq \mathbf{R};\mathbf{R}}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R};\mathbb{I} \sqcap \mathbf{R} \sqsubseteq R;R}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R} \sqcap R \sqsubseteq R;R}}{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R} \sqsubseteq R;R}}{\dfrac{R \sqsubseteq \mathbb{I} \Rightarrow \mathbf{R} = \mathbf{R};\mathbf{R}}{\Longrightarrow \mathbf{R} \sqsubseteq \mathbb{I} \rightarrow R = R;R}}$$

RELVIEW [BBMS98] is an interactive computer system for calculating with relations and relational programs. It represent relations as graphs. A homogeneous relation is displayed as a directed graph, while a heterogeneous relation is depicted as a Boolean matrix.

The main purpose of RELVIEW is to evaluate expressions which are inductively constructed from the basic relational operations (e.g., $^-$, $^\smile$, &, |, and $*$ for complement, transposition, intersection, union, and multiplication, respectively), residuals, quotients, relational functions, tests, and relational programs defined by users [BBMS98]. A relational function is denoted by $f(x_1, \cdots, x_n) = t$ where f is the function name, $x_1, \cdots, x_n$ are the relational parameters, and t is the relational term. For example, relational function f, which computes the expression $R \cap \overline{RR^+}$, is represented as $f(R) = R\& - (R * trans(R))$ [BBMS98]. A relational program, which is stored in a text file, is a while-program based on input binary relations. The structure of a relational program contains a head line, declaration part, and its body. The headline contains the program's name and a list of formal parameters. The declaration part consists of the declarations of local relational domains, local relational functions, and local variables. The body is a sequence of statements which are separated by semicolons and terminated by the return-clause.

The following is an example of a relational program given in [BBMS98]. It uses Prim's method to compute the relations of a spanning tree for a nonempty, undirected, and connected graph with relation **E**. The predefined operations *dom* and *ran* compute the domain and the corresponding range of a relation. Furthermore, the base operation *atom* yields for a non-empty relation a sub-relation which contains exactly one ordered pair.

```
Prim(E)
DECL T, v
BEG T = atom(E);
      v = dom(T)|ran(T);
      WHILE -empty(-v)
          DO
```

```
                T = T | atom(v * -v^ & E);
                v = dom(T)|ran(T)
               OD
        RETURN T| T^.
END.
```

RALL (**R**elation **A**lgebraic **L**anguage and **L**ogic) [vOG97] is a proof assistant system for relational calculus based on the TP Isabell/HOL. It provides the full language of heterogeneous relation algebra including higher-order operators, such as *join*, *meet*, *complement*, and *quantification* over relations. In addition, it enables the verification of the *type correctness* of all involved formulas for heterogeneous relations [vOG97].

The system supports capabilities for both interactive and automatic theorem proving. The interactive nature primarily represents forward and backward chaining in which each step is a predicate logic or an algebraic manipulation of terms, substitutions of equal to equal relations, and estimations of relational inclusions performed by using monotonicity of operators [vOG97]. The automatic nature represents a transformation from relational algebraic formulas into propositional logic, i.e, an inclusion is turned into an implication, a join into a disjunction, a meet into a conjunction. For example, $x \subseteq y \cup z$ becomes $\forall(x \mid x \in atom : x \subseteq y \cup z \rightarrow x \subseteq y \vee x \subseteq z)$ [vOG97].

1.1.2 Tools that use a combination of CASs and TPs

Maple-PVS [SG05], as its name suggests, is an interface between Maple and PVS. The main objective of the interface is to handle all the communications (e.g., relay messages) between them. It allows a Maple user, from its session, to access checkable proof environment of PVS. The activities of the interface are accessed

by three main components: the filter, the GUI front end, and the wrapper. The filter is responsible for classifying PVS outputs into different categories. Then, it relays all messages to the GUI front end. The GUI front end provides a graphical front end to Maple-PVS and distributes messages to Tcl/Tk window and Maple. The wrapper is used to initialize the interface, assemble PVS commands, get inputs and outputs, and shutdown the interface.

Besides the tool Maple-PVS, MathScheme [Mat06] is a project which aims at developing a TP and a CAS to develop a new approach to mechanized mathematics where it merges the formal deduction and the computation into a single activity.

Compared to other related approaches, the one adopted in the thesis is the only one that uses the batch modes of a TP and a CAS to handle relational calculus. The most important difference is that our approach uses a heuristic to assign tasks to be performed by these tools. For more details on heuristic, we refer readers to Chapter 5.

1.2 Motivation for our research

We think that mechanized mathematics systems, especially TPs and CASs, can have a critical impact on solving relational algebraic problems. The characteristics of a CAS help to automate the tedious or complicated symbolic computation. However, a CAS does not have the ability to make sophisticated derivations of logical formulas. Correspondingly, a TP is powerful in making logical formulas verification, but it meets difficulties in dealing with symbolic expressions.

For an illustrative example, we consider the following conjecture:

$$\forall(x \mid x \in \mathbb{R} : 3 * x^2 - 7 * x + 1 = 0 \Rightarrow x \geq 0) \tag{1.1}$$

From [Wes99, page 41, problem A8], all the chosen CASs in the test suite studied in the referred paper lack the capabilities to establish the truth value of this universal quantification. In general, a CAS does not have a mechanism for verifying the correctness of a deduction. We attempt to ask a TP, for example PVS, to determine its truth value. However, with the limitation of PVS in dealing with symbolic expressions, it is challenging for it to handle the equation $3*x^2-7*x+1=0$. Also, from [Wes99, page 44, problem G6], it shows that this equation can be successfully solved by almost all CASs. Thus, we use a CAS to deal with it first. Then, Maxima is selected to compute it by the command *solve*:

```
(%i1)solve(3 * x^2 - 7 * x + 1, x);
(%o1) [x = - (sqrt(37) - 7)/6, x = (sqrt(37) + 7)/6]
```

Then, substituting $x = -\frac{sqrt(37)-7}{6}$ or $x = \frac{sqrt(37)+7}{6}$ into (1.1), we get:

$$\forall(x \mid x \in \mathbb{R} : x = -\frac{sqrt(37)-7}{6} \vee x = \frac{sqrt(37)+7}{6} \Rightarrow x \geq 0) \qquad (1.2)$$

Representing (1.2) in a PVS theory as follows:

```
forall_quantifier: THEORY
BEGIN
x: VAR real
forall_expression: LEMMA(FORALL x: x = ((7-sqrt(37))/6)
                         OR x = ((7+sqrt(37))/6) IMPLIES x>=0)
END forall_quantifier
```

PVS makes the verification and returns the status "Q.E.D", which is short for Latin phrase "*quod erat demonstrandum*" in meaning that "which was to be proved". Thus, with one time use of a CAS to find the solutions of the expression and then using a TP to verify the universally quantified formula, we can get the final result.

For more details on the application of the combination of a TP and a CAS, we attempt another example: $\exists(x \mid x \in \mathbb{N} : \frac{dy(x)}{dx} = x \ \wedge \ y(1.5) = 0 \ \wedge \ y(x) = 0)$.

In order to compute the conjecture, we can process according to the following steps:

1. We use a CAS to solve the differential equation $\frac{dy(x)}{dx} = x$. We get $y(x) = \frac{x^2}{2} + c$. Using $y(1.5) = 0$, we get $y(x) = \frac{4x^2-9}{8}$. Representing this in Maxima, we need to use the following commands:

```
(%i1) equation:  diff(y,x)=x;
(%o1) dy/dx=x
(%i2) ode2(equation,y,x);
(%o2) y = x^2/2 + %c
(%i3) ic1(%,x=1.5,y=0);
(%o3) y = (4x^2-9)/8
```

2. We invoke a TP to prove the following conjecture: $\exists(x \mid x \in \mathbb{R} : \frac{4x^2-9}{8} = 0)$. Again, a TP might not be able to prove this universal quantification due to the polynomial in its body. We still need a CAS to solve the equation $\frac{4x^2-9}{8} = 0$. We get $x = \frac{-3}{2}$ or $x = \frac{3}{2}$. Using the *solve* command of Maxima, we get:

```
(%i4) solve(ev(%,y=0));
(%o4) [x = - 3/2, x = 3/2]
```

3. After that, the TP has the ability to give the results for the conjecture $\exists(x \mid x \in \mathbb{R} : (x = \frac{-3}{2}) \vee (x = \frac{3}{2}))$.

   Using the strategy *grind* of PVS to make the verification for the conjecture which is coded in the following PVS theory:

```
exists_quantifier:  THEORY
BEGIN
x: VAR real
exists_expression: LEMMA(EXISTS x: x = -3/2 OR x = 3/2)
END exists_quantifier
```

   Then, substituting the values of x into PVS conjecture and representing it in a PVS theory file, we finally get the result:

```
exists_expression :
 |-------
{1}(EXISTS x: x = -3/2 OR x = 3/2)
```

   We proceed it as follows:

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Therefore, using a CAS twice to deal with the expressions and then using a TP, we obtain the final result.

1.3 Problem statement

In the rest of the thesis, we handle relations through their characteristic predicates. The calculus that we use involves five fundamental operators: *union*, *intersection*, *inverse*, *composition*, and *complement*. Operations on relations are translated into the corresponding operations on their characteristic predicates. Therefore, they are operations on predicates.

For example, we attempt to use the operator *Union* to compute the union of two relations. We take the first following predicate:$\exists(z \mid z \in \mathbb{R} : x + z = y - z)$. This predicate can be considered as representing the following relation $R_1 = \{(x, y) \mid x, y \in \mathbb{R} : \exists(z \mid z \in \mathbb{R} : x + z = y - z)\}$.

We take a second predicate: $\exists(z \mid z \in \mathbb{N} : \frac{dy(x)}{dx} = -\frac{zx}{y} \ \wedge \ y(z) = 0)$. Let us assume that it represents the relation

$$R_2 = \{(x, y) \mid x, y \in \mathbb{R} : \exists(z \mid z \in \mathbb{N} : \frac{dy(x)}{dx} = -\frac{zx}{y} \ \wedge \ y(z) = 0)\}.$$

Then,

$$R_1 \cup R_2$$

$$=$$

$$\{(x, y) \mid x, y \in \mathbb{R} : \ \exists(z \mid z \in \mathbb{R} : x + z = y - z)$$
$$\vee \exists(z \mid z \in \mathbb{N} : \frac{dy(x)}{dx} = -\frac{zx}{y} \ \wedge \ y(z) = 0) \quad \}$$

For more details on operations on relations, we refer readers to Section 2.2.

As illustrated in Section 1.2, a TP or a CAS cannot solely deal with the obtained quantification. A TP meets the difficulties in simplifying and solving symbolic and numeric expressions while a CAS lacks the capabilities in establishing the truth value of the predicate that involves them.

1.4 Objectives and contributions of our research

The main objectives of the thesis consist of:

1. Investigating and evaluating the characteristics and capabilities of two separated classes of mechanized mathematics systems, TPs and CASs, and make decisions to choose two suitable systems to be used by our system.

2. Designing and building a proof-of-concept tool which makes use of PVS and Maxima to perform relational calculus.

3. Exploring a strategy in scheduling tasks to PVS and Maxima to perform relational calculus and provide a heuristic to meet this strategy.

4. Assessing our proof-of-concept tool by handling some illustrative examples.

Therefore, our main contribution consists of investigating the feasibility of the automation of relational calculus using existing mechanized mathematics technologies. Specially, we elaborated a heuristic that enables the assignment of tasks to PVS and Maxima in the aim to perform relational calculus. We as well built proof-of-concept tool that supports this calculus.

1.5 Structure of the thesis

The rest of the thesis is structured as follow:

Chapter 2 introduces the mathematical background of the thesis.

Chapter 3 introduces theorem proving systems and computer algebra systems.

Chapter 4 gives the design that we adopt for the proposed proof-of-concept tool.

Chapter 5 explores the heuristic on which the scheduling of the invocation of

PVS and Maxima is based and lays out representative examples that we use to assess our system.

Chapter 6 presents our conclusion and gives an idea about potential future work.

Chapter 2

Mathematical Background

This chapter introduces the mathematical concepts needed for the thesis. It includes sets, relations, predicates, relation algebra, polynomial, the natural deduction, and disjunctive normal forms.

2.1 Sets

A set is a collection of distinct elements [SS93, Chapter 1]. There are two main ways to represent a set: *enumeration* and *comprehension. Set enumeration* lists all the elements of a set. The set is delimited by "{" and "}" and its elements are separated by commas. For example, $A = \{2, 4, 6, 8\}$ denotes a set with four elements 2, 4, 6, and 8. *Set comprehension* (or set builder) states the properties the elements must satisfy to be a member of the set. It is represented in the form $\{x_1 : t_1, \cdots, x_n : t_n \mid P : E\}$ where x_i is a bound variable, t_i is a type of x_i for $i \in [1, \cdots, n]$, P is a predicate, and E is an expression. For example, a set of all positive integer numbers whose squares are less than or equal to 25, is denoted by $A = \{x : \mathbb{Z} \mid x \geq 0 : x^2 \leq 25\}$.

A set theory considers sets constructed from some collections of elements.

This collection of elements is called the *domain of discourse* or *universe of values*, denoted by **U**. The universe is considered as the type of every set variable in the theory. In the traditional form, set comprehension is denoted by $\{x : t \mid P\}$ or $\{x \mid P\}$.

When x is a member of a set A, we denote it by $x \in A$. The cardinality of a finite set A, denoted by $|A|$, is the number of elements of A. Set S is *finite*, with cardinality or size n for some natural number n, if there exists a bijective function $f : (0..n-1) \rightarrow S$; otherwise, S is *infinite* [GS93]. Empty set, denoted by $\emptyset$ or $\{\}$, is the set which has cardinality equal to 0. The power set of X, denoted by $\mathcal{P}(X)$ or 2^X, is the set of all subsets of X.

2.1.1 Operations on sets

Let A and B be sets, and let **U** denote the universe. The operations on sets are defined as follows (where $\triangleq$ denotes *defined as*):

Subset	$A \subseteq B \iff \forall(x \mid x \in A : x \in B)$
Proper subset	$A \subset B \iff A \subseteq B \ \wedge \ A \neq B$
Superset	$A \supseteq B \iff B \subseteq A$
Proper superset	$A \supset B \iff B \subset A$
Union	$A \cup B \triangleq \{x \mid x \in A \ \vee \ x \in B\}$
Intersection	$A \cap B \triangleq \{x \mid x \in A \ \wedge \ x \in B\}$
Difference	$A - B \triangleq \{x \mid x \in A \ \wedge \ x \notin B\}$
Complement	$\overline{A} \triangleq \mathbf{U} - A \triangleq \{x \mid x \in \mathbf{U} \wedge x \notin A\}$
Disjoint	$disj(A, B) \iff A \ \cap \ B = \emptyset.$

2.2 Relations and predicates

Let A and B be two sets. The *Cartesian product* of A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$, i.e. $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$. Every subset R of the Cartesian product of A and B is called a relation, i.e. $R \subseteq A \times B$. If $A = B$, R is called a *homogeneous* relation, i.e. $R \subseteq A \times A$. If $A \neq B$, R is called a *heterogeneous* relation. R has the type $A \leftrightarrow B$, which we denote by $R_{A \leftrightarrow B}$.

We list three special relations:

(1) Empty relation: $\bot_{A \leftrightarrow B} = \{(x, y) \mid false\}$

(2) Universal relation: $\top_{A \leftrightarrow B} = \{(x, y) \mid true\}$

(3) Identity relation: For every set A, $\mathbb{I}_{A \leftrightarrow A} = \{(x, x) \mid x \in A\}$

Operations on Relations

Let P and Q be homogeneous binary relations. We present the following operations on relations:

Union $P \cup Q \triangleq \{(x, y) \mid (x, y) \in P \vee (x, y) \in Q\}$

Intersection $P \cap Q \triangleq \{(x, y) \mid (x, y) \in P \wedge (x, y) \in Q\}$

Inverse $P^{\smile} \triangleq \{(y, x) \mid (x, y) \in P\}$

Complement $\overline{P} \triangleq \{(x, y) \mid (x, y) \notin P\}$

Domain $dom(P) \triangleq \{x \mid \exists(y \mid : (x, y) \in P)\}$

Range $ran(P) \triangleq \{y \mid \exists(x \mid : (x, y) \in P)\}$

Composition $P;Q \triangleq \{(x, z) \mid \exists(y \mid : (x, y) \in P \wedge (y, z) \in Q)\}$

defined iff P and Q are defined on a same set A (i.e., $P \subset A \times A$ and $Q \subset A \times A$ since P and Q are homogeneous binary relations.)

For simplicity, we write $\exists(y \mid : (x, y) \in P)$ instead of $\exists(y \mid true : (x, y) \in P)$.

2.2.1 Relation algebra and relational algebra

Definition. A *heterogeneous* abstract relation algebra is a structure $(\mathcal{R}, \cup, \cap, ;, ^{-}, ^{-1})$ on a nonempty set $\mathcal{R}$, whose elements are called relations. Each relation $R \in \mathcal{R}$ is associated with a type $A \leftrightarrow B$. We have the following conditions:

(i) The operations $^{-}$ (complement) and $^{\smile}$ (inverse) are total operations.

The operations $\cup$ (supremum), $\cap$ (infimum), and ; (composition) are partial.

a) The operation $Q_{A\leftrightarrow B} \cup R_{C\leftrightarrow D}$ is defined iff $A = C$ and $B = D$.

b) The operation $Q_{A\leftrightarrow B} \cap R_{C\leftrightarrow D}$ is defined iff $A = C$ and $B = D$.

c) The operation $Q_{A\leftrightarrow B}; R_{C\leftrightarrow D}$ is defined iff $B = C$.

We have:

$Q_{A\leftrightarrow B} \cup R_{C\leftrightarrow D} : A \leftrightarrow B$

$Q_{A\leftrightarrow B} \cap R_{C\leftrightarrow D} : A \leftrightarrow B$

$\overline{R_{A\leftrightarrow B}} : A \leftrightarrow B$

$(R_{A\leftrightarrow B})^{\smile} : B \leftrightarrow A$

$Q_{A\leftrightarrow B}; R_{B\leftrightarrow C} : A \leftrightarrow C$

(ii) Each structure $(\mathcal{B}_{A\leftrightarrow B}, \cup, \cap, \bot_{A\leftrightarrow B}, \top_{A\leftrightarrow B})$, where $\mathcal{B}_{A\leftrightarrow B}$ is the set of the relation having the type $A \leftrightarrow B$, is a complete atomic boolean algebra.

(iii) For each relation $R : A \leftrightarrow B$, there exists a left identity $\mathbb{I}_{B\leftrightarrow B} : B \leftrightarrow B$ and a right identity $\mathbb{I}_{A\leftrightarrow A} : A \leftrightarrow A$ such that: $\mathbb{I}_{A\leftrightarrow A}; R_{A\leftrightarrow B} = R_{A\leftrightarrow B}; \mathbb{I}_{B\leftrightarrow B} = R_{A\leftrightarrow B}$.

(iv) $Q_{A\leftrightarrow B}; (R_{B\leftrightarrow C}; S_{C\leftrightarrow D}) = (Q_{A\leftrightarrow B}; R_{B\leftrightarrow C}); S_{C\leftrightarrow D}$

(v) The Schröder rule holds:

$Q_{A\leftrightarrow B}; R_{B\leftrightarrow C} \subseteq S_{A\leftrightarrow C}$

$$\Longleftrightarrow$$

$$(Q_{A\leftrightarrow B})^{\smile}; \overline{S_{A\leftrightarrow C}} \subseteq \overline{R_{B\leftrightarrow C}}$$

$$\Longleftrightarrow$$

$$\overline{S_{A\leftrightarrow C}}; (R_{B\leftrightarrow C})^{\smile} \subseteq \overline{Q_{A\leftrightarrow B}}$$

(vi) The Tarski rule holds: $R_{B\leftrightarrow C} \neq \bot_{B\leftrightarrow C} \Longrightarrow \top_{A\leftrightarrow B}; R_{B\leftrightarrow C}; \top_{C\leftrightarrow D} = \top_{A\leftrightarrow D}$.

A relational algebra is a model of relation algebra where $\mathcal{R}$ is the set of binary relations.

2.2.2 Predicates

Predicates are applications of boolean functions whose arguments may be of type other than the set of Boolean $\mathbb{B}$, denoted by $f : \Sigma \rightarrow \mathbb{B}$ where Σ is a type in higher-order logic. The predicates over Σ form a function space that we write $\wp(\Sigma)$, i.e.

$\wp(\Sigma) = \{f \mid f : \Sigma \rightarrow \mathbb{B}\}$

A predicate $p : \Sigma \rightarrow \mathbb{B}$ determines a set $A_p \subseteq \Sigma$ such that $A_p = \{\sigma \in \Sigma \mid p(\sigma)\}$. Examples of predicates are *greater*, *even*, and *succ*. The arguments of predicates are expressions, i.e, arguments can be constants, variables and operations combining them. These arguments are also called *terms*. Terms with operations and constants are called *ground terms*. Examples of terms are: $\frac{\sqrt{x+y}}{2}$, and $b^2 - 4 * a * c$. Let r be a predicate, x a variable,

$y \in \{x \mid r\} \Longleftrightarrow r[x := y]$ for any expression y.

This result is borrowed from [GS93, Chapter 11]. It formalizes the connection between sets and predicates: a predicate is a representation for the set of argument-values for which it is true.

To each predicate r, there corresponds a set comprehension $\{x : t \mid r\}$, which contains the objects in t that satisfy r. Then, r is called a *characteristic predicate* of the set.

2.3 Polynomial

A polynomial is a function of the form

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

where a_0 is called the constant coefficient, a_n is called the leading coefficient, and n is called the degree of the polynomial. A *root* or a *zero* of a polynomial p is a number ζ such that $p(\zeta) = 0$.

2.4 Natural deduction

The Natural Deduction (ND) is used to provide the logical rules to simplify the logical formulas of the predicate. These rules are also discussed in the heuristic in Chapter 5.

The following definition of ND is borrowed from [GS93]. ND is a method to construct patterns of reasonings or proofs in natural languages. There are two main inference rules for each operator and each constant: an *introduction rule* and an *elimination rule*. Introduction rules produce complex statements from smaller statements by introducing connectives. Elimination rules produce simpler statements from complex statement by eliminating connectives.

Table 2.1 presents the inference rules for ND. In the table, for each operator or constant $\star$, the corresponding rules are named as $\star$-I and $\star$-E for introduction rule for $\star$ and elimination rule for $\star$, respectively. For example, the introduction

and elimination rules for $\wedge$ are $\wedge$-I and $\wedge$-E. Each inference rule is a schema, and substituting boolean expressions for the variables p, q, and r in it yields an inference rule as shown in Table 2.1.

Table 2.1: Inference rules for natural deduction

Introduction rules		Elimination rules	
Name	Rule	Name	Rule
$\wedge$-I	$\frac{p \quad q}{p \wedge q}$	$\wedge$-E	$\frac{p \wedge q}{p}$, $\frac{p \wedge q}{q}$
$\vee$-I	$\frac{p}{p \vee q}$, $\frac{q}{q \vee p}$	$\vee$-E	$\frac{p \vee q,\ p \Rightarrow r,\ q \Rightarrow r}{r}$
$\Rightarrow$-I	$\frac{p_1,\cdots,p_n \vdash q}{p_1 \wedge \cdots \wedge p_n \Rightarrow q}$	$\Rightarrow$-E	$\frac{p,\ p \Rightarrow q}{q}$
$\equiv$-I	$\frac{p \Rightarrow q, \quad q \Rightarrow p}{p \equiv q}$	$\equiv$-E	$\frac{p \equiv q}{p \Rightarrow q}$, $\frac{p \equiv q}{q \Rightarrow p}$
$\neg$-I	$\frac{p \vdash q \wedge \neg q}{\neg p}$	$\neg$-E	$\frac{\neg p \vdash q \wedge \neg q}{p}$
true-I	$\frac{p \equiv p}{\text{true}}$	true-E	$\frac{\text{true}}{p \equiv p}$
false-I	$\frac{\neg\text{true}}{\text{false}}$	false-E	$\frac{\neg\text{false}}{\text{true}}$

2.5 Disjunctive normal form and conjunctive normal form.

The following definitions are borrowed from [End72].

A propositional formula is in Disjunction Normal Form (DNF) if it is a disjunction of conjunctions of literals. A propositional formula is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions of literals. A literal is . For example, the expression $(x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z)$ is a DNF and $(x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z)$ is a CNF. Every propositional formula is logically equivalent to a DNF or a CNF [End72, page 49].

2.6 Linear notation

In our system, we use a linear notation to represent a quantification, a proposition (which can be handled by PVS), and an expression (which can be handled by Maxima). The format of a linear notation is structured as follows:

(1) ((2) | (3) : (4))

For a formula, (1) denotes quantifier symbol *Forall* or *Exists*, (2) denotes the list of bound variables which are separated by commas, (3) denotes the domain range of variables, and (4) denotes the body context of the quantification. The body context can be one of the following kinds: a Boolean value *True* or *False*, an expression, a proposition, another quantification, or the conjunction of them.

For example, $Forall(x, y, z \mid x\ in\ real,\ y\ in\ real,\ z\ in\ real : x + (y + z) = (x + y) + z)$ and $Forall(x \mid x\ in\ real : Exists(y, z \mid y\ in\ real, z\ in\ int : x = 2\ AND\ x + y + z = 3\ AND\ x * y * z = 1))$ are two quantifications represented in the linear notation.

For a proposition, (1) denotes the keyword "lemma", (2) denotes the list of bound variables which are separated by commas, (3) denotes the domain range of variables, and (4) denotes the proposition context.

For example, $lemma(x \mid x\ in\ nat : x > 1 \Rightarrow x > 2$ denotes the proposition $x > 1 \Rightarrow x > 2$

For an expression, (1) denotes a Maxima keyword, such as *solve* for solving an equation, *integrate* for simplifying an integration, *diff* for simplifying a differentiation, (2) denotes the list of bound variables which are separated by commas, (3) denotes the domain range of variables, and (4) denotes the expression context.

For example, $solve(x \mid x\ in\ real : x^3 - 5 * x + 4 = 0)$ denotes the solving of the polynomial equation $x^3 - 5 * x + 4 = 0$.

In the implementation, this predicate or expression is represented as a string

and stored in a file.

2.7 Conclusion

In this Chapter, we presented the basic notion and gave established results that form the ground on which the proof-of-concept tool we built stands. This chapter introduced as well some terms that are used in the sequel of the Thesis.

Chapter 3

Theorem Provers and Computer Algebra Systems

In Chapter 1, we illustrated the fact that a TP or a CAS alone cannot be of a great help in performing relational calculus. A TP faces challenges in the simplification of symbolic expressions and a CAS lacks the capabilities in making derivations. In order to overcome this difficulty, the idea of using a combination of a TP and a CAS to handle relational calculus is introduced and will be investigated in the sequel.

In this chapter, we aim at elaborating on the roles of TPs and CASs and on the TP and CAS that we choose to use. A TP provides mechanized support for proving conjectures using axiomatic theory. Examples of TPs include HOL [GM93], IMPS [FGT93], Isabelle [NPW02], PVS [ORS92], and TPS [ABB93]. A CAS is designed for performing symbolic computation. Symbolic computation can provide substitution of symbolic values for expressions, differentiation with respect to one or all variables, finding solutions of linear or polynomial equations, decomposition of polynomials, factorization of polynomial terms, integration of indefinite or definite integrals [Wik06a].

In [Wes99], Wester described a comprehensive test suite for symbolic mathematics softwares. He selected 542 mathematical problems and attempted to solve them with seven mathematics software systems: Axiom [JS92], Derive [KV00], Macsyma [Hel91], Maple [Hec93], Mathematica [Wol88], MuPAD [MuP06], and Reduce [Ray87]. The selected ones provide both the breadth and the depth of the problems in CASs [Wes99]. Among the seven CASs, Maple, Mathematica, and Macsyma are evaluated as strongest systems [Wes99]. Section 3.2 gives the literature reviews of CASs.

In our research, we use the batch mode of a TP and a CAS to perform relational calculus. The batch mode means that specifications and proofs of a TP or commands of a CAS being processed are not displayed. We can put all the conjectures that need to be proved by a TP or all the expressions that need to be simplified by a CAS in text files and then pass them to the corresponding system. After processing in batch mode, they generate files containing the results.

This chapter is organized as follows. The main characteristics of TPs and the reasons for choosing PVS as our TP are discussed in Section 3.1. The main characteristics of CASs and the reasons for choosing Maxima as our CAS are discussed in Section 3.2.

3.1 Theorem provers

3.1.1 PVS

PVS, short for **P**rototype **V**erification **S**ystem [ORS92], is a mechanized computation system for formal specification and verification. PVS is a system that consists of a specification language, a theorem prover, and various utilities and documentation [Owr04].

The specification language of PVS is based on typed higher order logic, extended with predicate subtype and dependent type. Typed higher order logic consists of base types, abstract data types, and uninterpreted types. Base types are built-in types such as boolean, reals, and integers. Abstract data types include set, tuple, record, list, and binary trees [ORS92]. Uninterpreted types are the types introduced by users. Predicate subtype is a type that contains the elements of a given type satisfying a given predicate, e.g, the type of non-zero numbers [ORS92]. Dependent types are the types that depend on values. For instance, we consider the function type rem : $[\text{nat}, d : \{n : \text{nat} \mid n/ = 0\} \rightarrow \{r : \text{nat} \mid r < d\}]$. The declaration for *rem* indicates the range of the remainder function, which depends on the second argument [OSRSC01].

A PVS specification is organized into parameterized theories which consist of assumptions, definitions, axioms, and theorems. PVS expressions include the usual arithmetics and logical operators, function applications, quantifiers, variables with freely overloaded names, i.e, multiple variables can be defined with the same name, and an extension prelude of built-in theories [ORS92].

The logic of PVS includes structural rules, propositional rules, quantifier rules, and equality rules. Structural rules permit a weaker statement that can be derived from a stronger one by adding either antecedent formulas or consequent formulas. For example, the relation $\Gamma_1 \subset \Gamma_2$ holds between two lists when all the formulas in Γ_1 is a subset of the list Γ_2, i.e, $\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2} w$ holds if $\Gamma_1 \subseteq \Gamma_2$ and $\Delta_1 \subseteq \Delta_2$ [OSRSC01]. The propositional rules consist of a propositional *Axiom* rule, the *Cut* rule for introducing case split, a rule for lifting *If* conditionals to the top level of a formula, and basic rules such as *conjunction*, *disjunction*, *implication*, and *negation*. The *quantifier* rules include a rule for substituting universally quantified variables with constants and a rule for replacing existentially quantified variables with terms. The *equality* rules include a rule for substituting one side of an equality

premise by another [ORS92].

PVS proof checker supports the development of readable proofs in all stages of the proof development life cycle. It handles a set of primitive inference rules, a mechanism to program these rules into strategies, a facility to rerun proofs, and a feature to verify that all secondary proof obligations, (e.g, type correctness conditions), have been discharged [ORS92]. A proof can be saved in a file and rerun automatically or rerun in single step mode. It is represented as a tree graph, allowing users easily to see all the branches of proofs [GH06]. For more details on PVS theorem proof, we refer readers to [ORS92].

The built-in commands support very powerful automaton that contains decision procedures for linear arithmetic, propositional simplification, and automatic rewriting [SORSC01]. In addition, a proof can be constructed by primitive proof rules which are programmed as strategies. Proof strategy is considered as a template which contains patterns of inference steps. PVS provides a large number of strategies. These strategies can be used to prove a variety of theorems. They can be used as a starting point for the proof of any conjecture.

3.1.2 Isabelle

Isabelle [NPW02] is a generic theorem prover system which supports a platform in which different logics can be represented by specifying their syntax and inference rules. The system represents rules as propositions and constructs proofs by combining rules [Pau94].

The specification language of Isabelle is originated from functional programming language, especially Meta Language (ML) [NPW02]. Based on the characteristics of ML, the syntax of Isabelle can be easily extended [Mar06]. Its specification allows one to import various theories, but it does not permit param-

eterization [Mar06]. It generates automatically an induction principle for each recursive data type.

There are various kinds of logics implemented in Isabelle. Its most important logic is meta logic. Other logics can be built based on it by asserting axioms and declaring types. The meta logic is a form of higher order logic. The rules of meta logic are represented in Natural Deduction form. Natural Deduction is a means to formalize reasoning patterns in natural languages [GS93]. Some inference rules that work in meta logic are: introduction, elimination, substitution, implication, negation, and quantifier rule. The inference rules are specified by using three meta-level connectives: implication $\Rightarrow$, quantification /$, and equality ==. For more details on its logic, we refer readers to [NPW02].

Isabelle provides *tactics* and *tacticals*. The first are proof commands. Tacticals which are called also *proof strategies* are functions which build new proof commands (i.e., tactics) using more basic ones [GH06]. A tactic is represented as an ML function where the input is a goal, the output is a list of subgoals along with a proof. A goal is a list of terms paired with a term, corresponding to the hypothesis and conclusion of a theorem. Tactics written directly in ML may fail in various ways and they usually cannot cause theorems to appear, thus, it is difficult to trace the failures [GM93]. In addition, it does not give elaborate proof support and it is not easy to see which branch belongs to a proof [Mar06].

Isabelle has often been seen as a tool for implementing various logics and examining proof systems.

3.1.3 HOL

The HOL system is a computer program for constructing formal specifications and proofs in higher order logic. It supports reasoning in many areas, including

program correctness and refinement, hardware design and verification, compiler verification, and proofs in real-time systems [GM93].

In HOL, the specification language is a higher order logic which allows functions and relations to be passed as arguments to other functions and relations [GM93].

The HOL logic contains the syntax and semantics of categories of types and terms. The types are expressions that denote sets. The terms are elements of the sets that are denoted by corresponding types. There are four kinds of types: type variables denoting arbitrary sets in the universe $\mathbb{U}$, atomic types denoting fixed sets in the universe, compound types denoting operations for constructing sets denoting the type of functions, and functions types. The type structure is a two-tuple (F, n) where F is a set of atomic types and function types, and n is its corresponding arities. The terms of the logic are expressions that represent elements of the sets denoted by types. Terms are classified into four groups: constants, variables, function applications denoting the combinations of the function symbols with the constants and variables, and λ-abstractions denoting a λ term $\lambda x.t$ where $\lambda x.t$ denotes a function $v \mapsto t[v/x]$ in which $t[v/x]$ is the result of substituting v for x in t [GM93]. For more details on its logic, we refer readers to [GM93].

The primary components that make theorem proving work in HOL are: the theory, the derived inference rules, tactic and tacticals. A theory is a record of already proved facts including type structure, a set of defined constants and a set of axioms (an axiom is represented by a list of sequents). A derived inference rule is considered as an ML procedure that implements a pattern of inferences and generates every primitive step of the proof. In each derived inference rule, all the hypotheses must appear as conclusions of some earlier inferences appearing in the proof. A tactic is a means of organizing the construction of proofs and

tacticals are used to create tactics [GM93]. For more details about its theorem proofs, we refer readers to [GM93].

It is often used as an open platform for theorem proving research. Users can add external tools to HOL and embed language just by programming in ML.

3.1.4 Motivation for the selection of PVS

Rationale for selecting PVS

In general, our system can be connected through an interface module to any theorem provers. However, to build the prototype needed for our work, we have selected PVS. Our selection is motivated by the following:

(1) In PVS, *proof strategies* built into the theorem prover handle different classes of theorem proofs automatically. We can write all the theorems needed to be verified and their proof strategies in text files and pass them to PVS to be processed together.

(2) PVS supports *batch mode* in which specifications and theorems are not displayed while they are automatically processed. In batch mode, there is no direct interaction with PVS; it processes whatever provided files and terminates after completing the last of them. Then, it generates the log file for the proof results of input theorems.

(3) PVS is continuously upgraded and its documentation is regularly updated online.

(4) PVS is evaluated as one of the most effective in decision procedure [Art95].

How is PVS used by our tool?

To verify conjectures in batch mode by PVS, one writes all conjectures intended to be proved and their proof strategies into text files and then sends both of them to PVS.

The strategy *grind* is sufficient for processing almost all the conjectures that our system handles. The strategy *grind* attempts to apply rewrite rules and lemmas to the handled conjecture [SORSC01].

To run PVS in batch mode, the user needs two files *.el* and *.pvs*. The file *.pvs* handles the specification of PVS theories. The *.el* handles the batch. The command *pvs -batch -load filename.el* can be used to run the batch file *filename.el* in the batch mode of PVS.

For instance, we use the batch mode to verify the conjecture

$$\forall(x, y, z \mid x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N} : x + (y + z) = (x + y) + z).$$

First, we represent the specification of this conjecture in the file named *pvsTheory.pvs* as below:

```
pvstheory: THEORY
BEGIN
     x, y, z: VAR nat
     associative_ax:AXIOM FORALL x, y, z: x + (y + z)
                                          = (x + y) + z
END
```

Second, we use the batch file *pvsFile.el* to run this theory. The content of *batchFile.el* is shown below. The *pvs-validate* macro is used to change context to the specified directory */PVS* and run the commands, collecting the output into the log file *pvsFile.log*. The *prove-formulas-theory* command runs the speci-

fication file *pvsTheory.pvs* using the proof strategy *grind*, and the last command prints out the message *Completed*.

```
(pvs-validate "pvsFile.log" "~/PVS"
(typecheck "pvsFile")
(prove-formulas-theory "pvsFile" "(grind)"))
(pvs-message ``Completed''))
```

Last, we use the batch command *pvs -batch -l pvsFile.el -v 3* to run *pvsFile.el*. After running the batch, it returns the *.log* file the result *Q.E.D*, i.e., the conjecture is proved. In our system design, the module *PVSInterface* is used to execute PVS in order to handle theories in batch mode.

3.2 Computer algebra systems

3.2.1 Maple

Maple [Hec93] is a commercial CAS developed in Waterloo University at the end of eighties. Maple's language is close to C and other declarative programming languages. It is written in its own language in which the source code is available to users, allowing users to inspect or modify according to their requirements.

The following commands, which are borrowed from [So06], evaluates the integration of a random polynomial. This example creates a random polynomial of two variables x and y and by using the created polynomial, we calculate the integration of level 1, 2, and 3. The purpose of the example is to provide brief ideas about the commands of Maple and show that a CAS can deal with various kinds of complicated symbolic expressions.

```
>f:= randpoly([x,y]);
f := -50 x + 23 x{^4} + 75 xy{^3} - 92 x^3 y^3 + 74 x y^4
>g1:= int(f,x);
g1:= -25x{^2} + 23/5 x^5 + 75/2 x^2 y^3 - 23x^4 y^2
                    + 2x^3 y^3 + 37x^2 y^4
>int(%, x);
-25/3 x^3 + 23/30 x^6 + 25/2 x^3 y^3 - 23/5 x^5 y^2
                    + 1/2 x^4 y^3 + 37/3 x^3 y^4
>int(%, x);
-25/12 x^4 + 23/210 x^7 + 25/8 x^4 y^3 - 23/30 x^6 y^2
                    + 1/10 x^5 y^3 + 37/ 12 x^4 y^4
>int(%, y);
-25/12 x^4 y + 23/210 x^7 y + 25/32 x^4 y^4 - 23/90 x^6 y^3
                    + 1/40 x^5 y^4 + 37/60 x^4 y^5
>int(%, y);
-25/24 x^4 y^2 + 23/420 x^7 y^2 + 5/32 x^4 y^5 - 23/360 x^6 y^4
                    + 1/200 x^5 y^5 + 37/360 x^4 y^6
```

Maple is evaluated as the fastest CAS. Moreover, it did extremely well in the test suite [Wes99, pages 41-60], especially in dealing with algebra and solving equations.

3.2.2 Mathematica

Mathematica [Wol88] is a commercial CAS which has been developed and distributed by Wolfram Research Incorporation since 1988.

It is written in C. It offers a functional programming language which features higher-order functions, pattern matching, and lazy evaluation [Wes99]. A higher-

order function means that a function can be passed as arguments of another function. Lazy evaluation allows arguments of a function only evaluated when they are needed.

The following example, which is borrowed from [Bla92], attempts to use the commands *Integrate*, *D*, and *simplify* to calculate numerically the integration, differentiation, and simplification of expressions, respectively.

```
In[1] := Integrate[1/(2 + 3x^2)^3, x]
Out[1] = x/(8(2 + 3x^2)^2) + 3x/(32(2 + 3x^2))
          + (3ArcTan[3x/Sqrt[6]])(32Sqrt[6])
In[2]:= D[%, x]
Out[2] = 3/(64(1 + 3x^2/2)) - 3x^2/(2(2 + 3x^2)^3)
           + 1/(8(2 + 3x^2)^2) - 9x^2/(16(2 + 3x^2)^2)
           + 3/(32(2 + 3x^2))
In[3]:= simplify[%]
Out[3] = (2 + 3x^2)^-3
```

In [Wes99, pages 41-60], it illustrates its strength in exact calculation of infinite integral and numerical calculation optimization. However, its language is the most distinctive of the other CASs. It is the least resembles a procedure language, such as C, thus, it is the hardest to pick up for those with programming experience. In addition, it includes an interface that is difficult to use from the command line.

3.2.3 Macsyma

Macsyma is a commercial CAS originally developed at Massachusets Institute Technology (MIT) from 1968 to 1982. It is one of the most mature CASs.

It is written in Common Lisp. Its input commands must end with either a

semi colon or a dollar sign. If the command is terminated by a semi colon, the result is printed out, otherwise, it is calculated but not printed out. Its language is case-insensitive.

In Macsyma, when a user first runs the application, it displays the prompt $(c1)$, i.e., the first command line has the label $(c1)$. The subsequent displayed output has the label $(d1)$. Each of the successive input-output pairs is labelled (cn), (dn) for $n = 1, 2, 3, 4, ...$

The example below, which is borrowed from [?], illustrates how to find all roots of a polynomial equation:

```
(c1)(2 * x + 1)^3 = 13.5 *  (x^5 + 1);
(d1)(2x + 1)^3 = 13.5(x^5 + 1)
(c2)allroots(%);
(d2)[x = 0.829675, x = -1.0157557, x = 0.9659626
     %i - 0.4069597, x = -0.9659626 %i - 0.4069597,
     x = 1.0000001]
```

Compared to other CASs, Macsyma contains a very user-friendly interface [Wes99, page 315]. In addition, from [Wes99, page 41-60], it shows that Macsyma did the most successful in the test suite, especially in the area of algebra and special functions.

3.2.4 Maxima

Maxima is an open source CAS which is distributed under the GNU General Public License. It is descended from the original Macsyma developed at MIT in 1982 and continuously maintained and upgraded until recently [Web05].

It is also written in Common Lisp. Different from Macsyma, its language is case sensitive. In Maxima, when a user first runs the application, it displays the

prompt ($\%i1$). The subsequent displayed output has the label ($\%o1$). Each of the successive input-output pairs is labeled ($\%in$), ($\%on$) for $n = 1, 2, 3, 4, \cdots$

In the following, we present an example which is borrowed from [Web05]. It sequences to get the differentiation, take the integration, and solve the equation of the expression $5 * x\hat{}3 - 4 * x - 1$.

```
(%i1) expr: (5 * x^3 - 4 * x - 1);
(%o1) 5x^3  - 4x - 1
(%i2) diff (expr, x);
(%o2) 15x^2 - 4
(%i3) integrate(expr,x,1/3,5/3);
(%o3) 80/27
(%i4) solve (expr, x);
(%o4) [x = -(sqrt(5) + 5)/10, x = (sqrt(5) - 5)/10, x = 1]
```

Due to the inheritance, Maxima has the advantages of Macsyma. Further more, because of its characteristics as an open-source system, Maxima is continuously maintained and upgraded by a strong community in the computer algebra area.

3.2.5 Motivation for the selection of Maxima

Rationale for selecting Maxima

All the above CASs (and many more) can deal with symbolic computation. However, we choose Maxima for the following reasons:

(1) Maxima has comparable capabilities in symbolic and numeric computation to Mathematica, Maple, and Macsyma. However, we can legally get a free copy of Maxima while the others are commercial. This is useful especially

for the ones who attempt to use our tool, they do not need to access to a commercial product.

(2) The interface is user-friendly and the language is easy to use [Web05]. This helps us quickly in getting familiar with the system and convenient in writing the commands in files used in the batch mode.

(3) Its source and documentation have been maintained and developed simultaneously. Although other systems, such as Maple and Mathematica are also upgraded, Maxima has a strong community that supports it.

How is Maxima used by our tool?

In our tool, we use the batch mode of Maxima to simplify the expressions of a Diophantine representation of a relation. To run it in batch mode, we use the option *-batch* or *-b*. The commands to simplify the mathematical expressions are written in batch files. The commands are terminated with ; or $. These files have the extension *.mc, .mac*, or *.dem*. For instance, we want to calculate the integration of the polynomial: $x^3 - x^2 - 2 * x$. We present the commands of Maxima to deal the polynomial in the file *batchfile.mac* as follows:

```
e:integrate(x^3 - x^2 - 2*x,x); stringout(result,e);
```

The command "stringout(result,e)" writes the expression to the file "result" in the same format of the input expression.

Running in batch mode of Maxima by the command *maxima -b batchfile.mac*, we obtain the following:

```
(%i1) e:integrate(x^3 -x^2 - 2*x,x); stringout(result,e);
(%o1) x^4/4 - x^3/3 - x^2
(%i2)
```

```
(%o2) /home/nguyehtt/Maxima/result
```

The output file gets the following result

```
x^4/4 - x^3/3 - x^2;
```

In our system design, the module MaximaInterface is used to execute Maxima to handle expressions in batch mode.

3.3 Conclusion

For our approach, we concentrate on two classes of systems, TPs and CASs, to help our tool in handling relational calculus. In order to choose the most appropriate system from each class, we reviewed the literature of the widely known TPs and CASs. Their characteristics are compared and evaluated. PVS and Maxima are selected to interface with our prototype tool. At the end, the mechanisms to implement the batch mode of PVS and Maxima together with illustrative examples are described.

Chapter 4

System Design

As we described in Section 1.2, a combination of usage of a TP and a CAS can help to perform relational calculus. A TP meets challenges in dealing with simplifying symbolic expressions and a CAS does not have the ability to make sophisticated derivations of logical formulas. However, a predicate might contain symbolic expressions. Therefore, a combination of a TP and a CAS is a solution to the difficulties faced by each of these technologies. A tool to automatically handle the combined usage of a TP and a CAS to perform relational calculus is the main contribution of our thesis. In this chapter, we discuss the design of a proof-of-concept tool.

The purpose of the design phase is to derive a system that meets the requirements specification. It includes the following activities [Bud93]:

- Postulate a solution

- Create a model for the solution

- Evaluate the model following the original requirement

- Elaborate the model to produce a detailed specification of the solution

In this chapter, we concentrate on building a proof-of-concept automated concrete relational calculus tool named CRCS (short for **C**oncrete **R**elational **C**alculus **S**implification). This proof-of-concept tool is implemented using functional programing language Haskell, and it interfaces with PVS and Maxima. It works on a Linux platform. It is intended to perform relational operators (*union*, *intersection*, *composition*, *inverse*, and *complement*). Each argument of a relational operation is stored in a text file. Then, the simplified result of an operation is written into a file. A heuristic is used to schedule a list of calls of PVS or Maxima in order to help simplify the relational calculus.

The rest of the chapter is structured as follows: Section 4.1 gives the overview of the system. In Section 4.2, we discuss the architectural design of CRCS.

4.1 Overview of the system

As described in the introduction of this chapter, CRCS system takes relations stored in files as inputs and communicate with two systems PVS and Maxima. The system overview is given in the Figure 4.1.

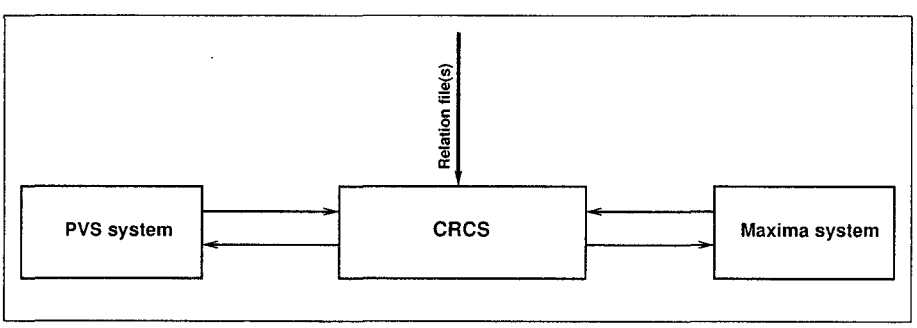

Figure 4.1: Overview of CRCS system

4.2 Architectural design

After giving the overview of the system, we introduce its architecture. The quality of the architectural design can be enhanced by adopting the following principles [GJM03]:

(1) *Modularization* principle is used to make the design easy to implement and manage. Each module denotes an individual work assignment. The notion of the work assignment is a portion small enough for a single programmer to implement in a reasonable time. A module has functional relation with other modules, which are called its clients, by USES relation. We say that a module A uses a module B if some programs in module A rely on the behavior of some programs in module B to accomplish their tasks [GJM03].

(2) *Design for change*, which is a motto adopted by Parnas [GJM03], is a way to make software easily modified as requirements change. This technique uses the Information Hiding principle. The concept of information hiding means that each module contains a secret which it hides from other modules. In general, the hidden information in each module can be divided into three main classes [HS99]:

- Behavior hiding where secrets are input formats, screen formats, and the text messages.

- Software decision hiding where secrets are internal data structures and algorithms.

- Machine hiding where secrets are hardware machine or virtual machine.

As illustrated in Figure 4.2, the architectural design of our system involves mainly three components: *Relational_Operations_Preparation*, *Simplification*,

and *Heuristic*. A software component is a system element providing a predefined global service. A component can be further decomposed into modules.

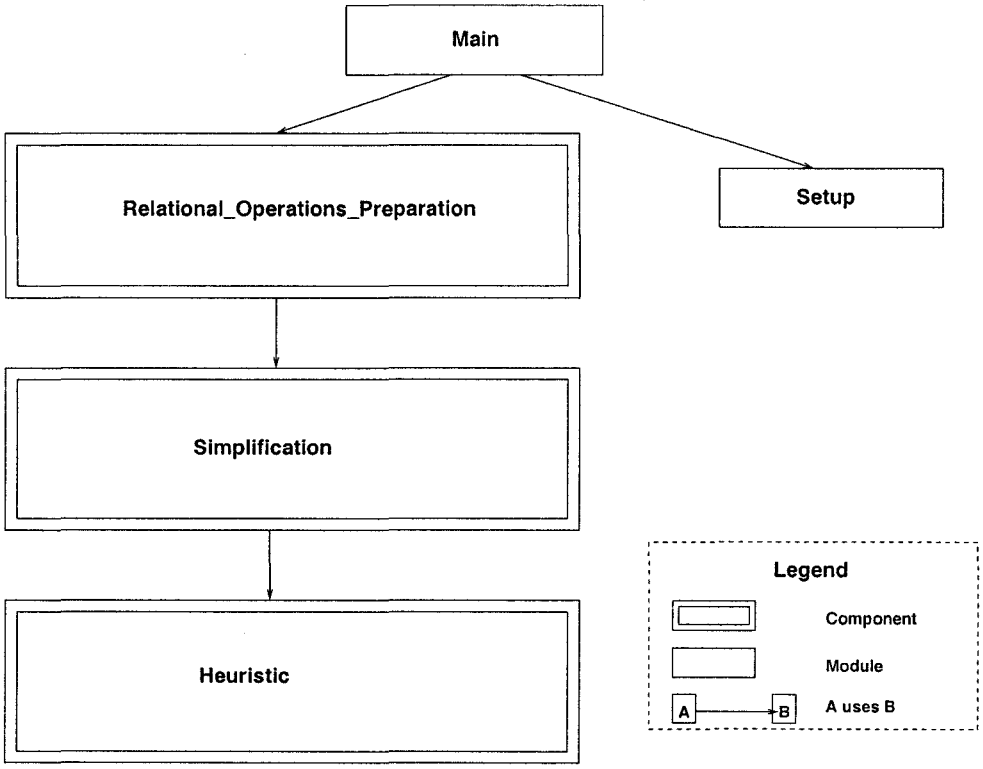

Figure 4.2: The main components of the proof-of-concept tool

4.2.1 Module decomposition

Using the principles of modularization [Par72], the CRCS is decomposed into a set of modules. Figure 4.3 and Figure 4.4 give the modules of the components *Relational_Operations_Preparation* and *Simplification*, respectively. The *Heuristic* component is discussed in the next chapter.

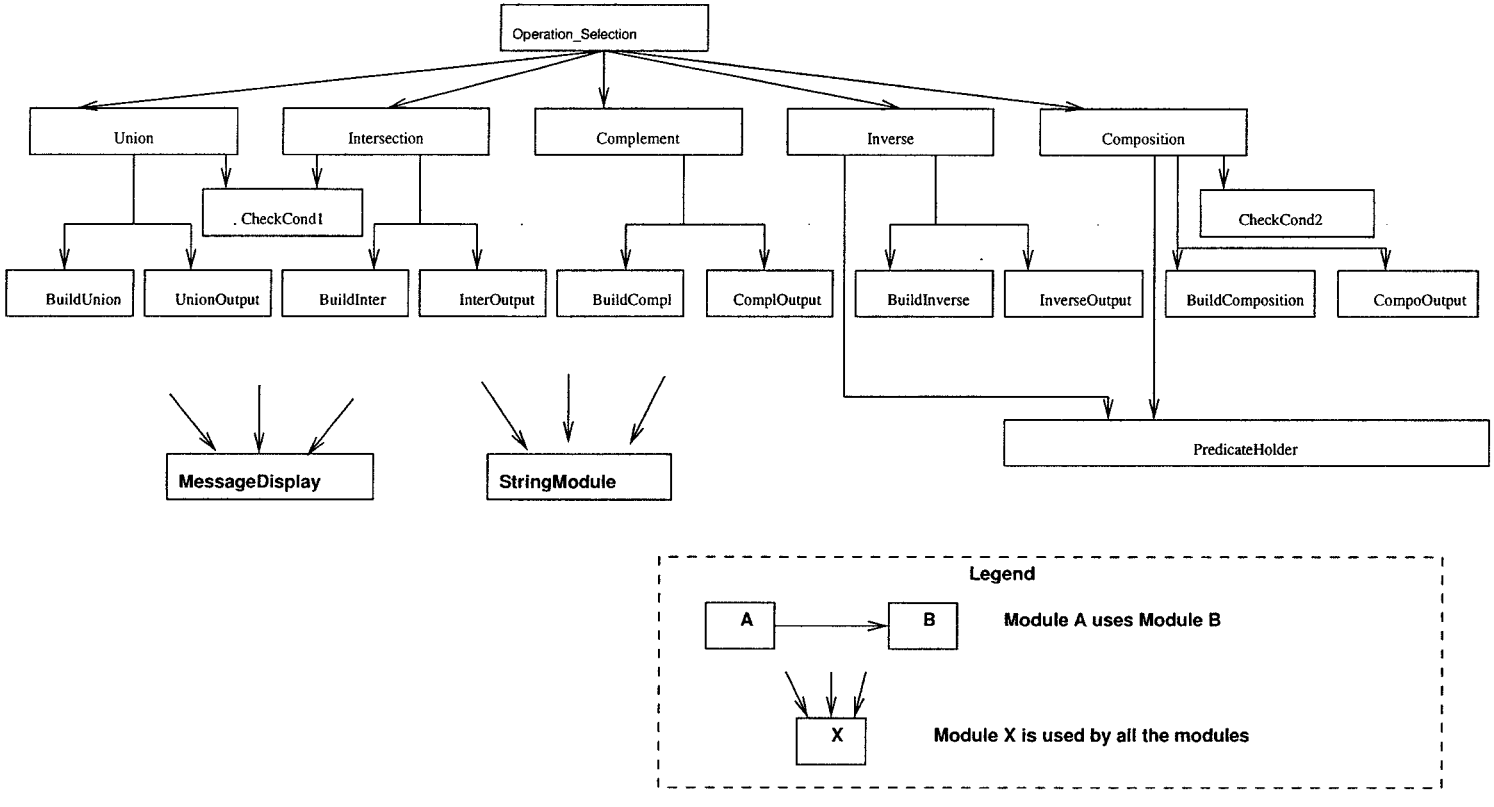

Figure 4.3: Module *Uses* diagram of the component *Relational_Operations_Preparation*

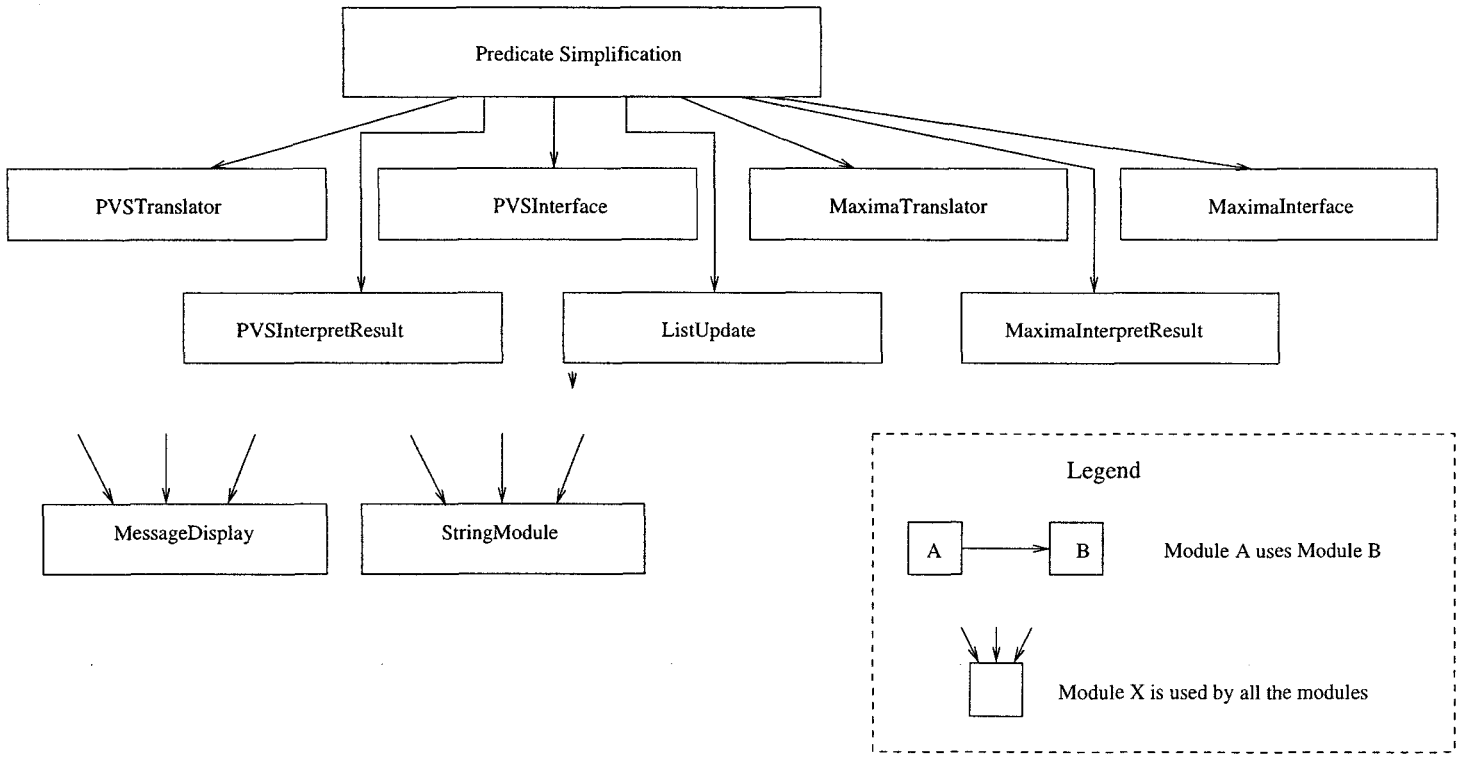

Figure 4.4: Module *Uses* diagram of the component *Simplification*

The module guide that presents the descriptions of the secret and service of each module is given in Appendix A. In Appendix B, we provide the detailed design of a selection of important modules.

4.2.2 Data flow diagram

Data Flow Diagram (DFD) is a notation for specifying the processes and flows of data between these processes [GJM03]. Data can be stored in data repositories, flowed in data flows, and transferred to or from the external environment. The basic elements of a DFD are:

- *Bubbles* are used to represent transformations. It identifies a transformation used to process input data and acquire output data.
- *Arrows* are used to represent data flows. It indicates direction of flow of named data.
- *Parallel lines* are used to represent data stores. It denotes the places where data structures are stored.
- *Rectangles* are used to represent external entities. It specifies the source or destination of a transaction.

For more information about the DFD, we refer readers to [GJM03, PP99]. Figure 4.5 gives the main DFD of the system. We note that certainly the DFD of Figure 4.5 can be refined more details on the flows of data.

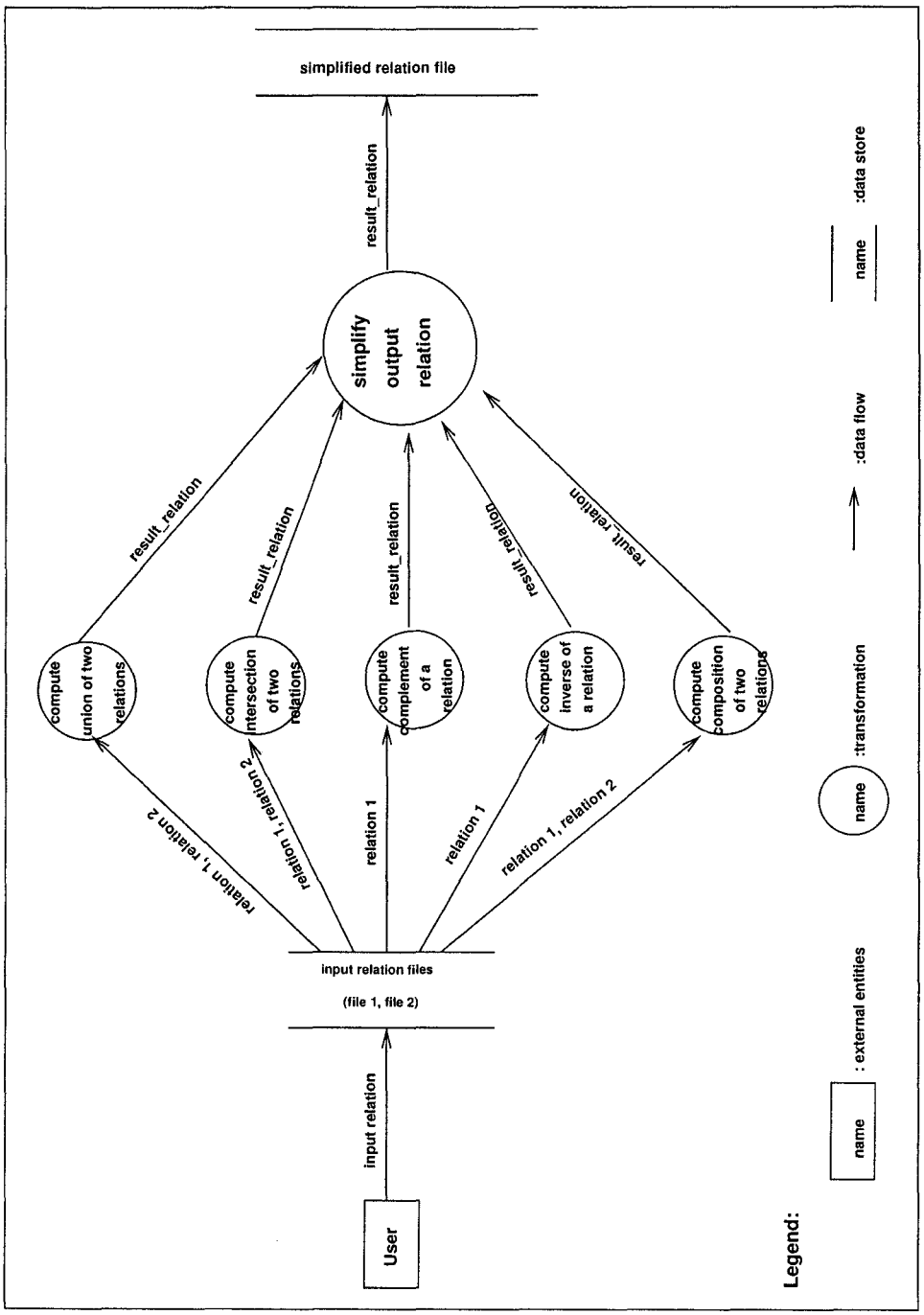

Figure 4.5: Data flow diagram of the system

4.3 Conclusion

We introduced the features of our proof-of-concept tool CRCS and the techniques to implement it. The principle of information hiding is used to decompose the system into modules. We encapsulate those that are likely to change in the module secret, allowing to keep the module interface stable even if there is some change in the implementation. Such a design makes the system easy to develop and maintain.

Chapter 5

Heuristic of Scheduling Modules

We find in [Wik06b] the following definition for a heuristic:

> A heuristic is a set of rules for solving a problem. A heuristic provides a quick means of solving a problem, utilizing a known rule of simplification to arrive at a solution in a shorter time than other conventional methods.

This means that a heuristic does not guarantee to be successful in dealing with a problem, however, it can solve the problem in a shorter time than a formal way.

In our research, we propose a heuristic to simplify relational calculus using available mechanized mathematics technologies. It provides some rules to allow us to assign tasks to PVS and Maxima while dealing with relational calculus. Achieving our objective in a formal way requires from us a lot of mathematical knowledge about mathematical reasoning. This requires us to spend much more research work relative to the scope of a Master thesis.

Before exploring the heuristic, first, we give our strategy in deriving the heuristic. The strategy introduces the high level options to construct the heuristic. The options of our strategy are presented in Section 5.1 and the heuristic rules are described in Section 5.2.

The notions of Diophantine representation of relations, disjunction normal form, and natural deduction are key mathematical background in our approach.

5.1 Strategy

In order to obtain guidelines or base options for our heuristic, we come up with a strategy. A strategy is defined as "a method including options and priorities towards the achievements of a defined goal objective" [Col96].

5.1.1 The elements of our strategy

Our strategy on which we aim to base our heuristic consists of the following elements:

1. In assigning tasks to a TP and a CAS, we favor the use of a TP.

   As it is stated in the [FvM00], a CAS is usually restricted to just a few areas of mathematics and often unreliable while a TP tends to be wide in scope and mathematically rigorous. Therefore, a TP is considered to be more stable and more rigorous while dealing with a predicate. Often, one can establish the truth value of a predicate without handling its symbolic expressions that might involve integrals and differential equations.

   In addition, in our research, we work frequently with quantifications and propositions. From [Wes99, page 41], a list of quantifications have been tested by CASs, however, all the chosen CASs lack the ability to establish their truth values while handling quantifications and propositions is one of major concerns of a TP.

   For instance, let us consider the statement: $\mathsf{false} \Rightarrow (x^5 - 3*x^4 + 5*x^3 - 2 = 0 \ \wedge \ \int_0^\infty \frac{x}{\sqrt{(x^2-25)}} dx = 0) \ \vee \ \forall(x \mid x \in \mathbb{R} \ \wedge \ x \geq 0 : x^2 \geq 4 \Rightarrow x \geq 2)$.

For this quantification, a TP itself is enough to verify the predicate and returns true based on the rule false $\Rightarrow P \equiv$ true with any value of P.

2. We use some rules of natural deduction to minimize the usage of both a TP and a CAS.

   Since CASs are unreliable, they cannot always succeed. Even for TPs, they are more rigorous, however, they are complicated to use. Minimizing the usage of not only CASs but also TPs makes our work more efficiently. In the following, a method to deal with this problem is presented.

   We attempt to apply the *Introduction rule* and *Elimination rule* of natural deduction rules, which are introduced in Section 2.4. We use only introduction and elimination rules for conjunction ($\wedge$) and disjunction ($\vee$) which are described in the Table 2.1 as $\wedge - I$, $\vee - I$, $\wedge - E$, and $\vee - E$ since all the other operators, such as $\Rightarrow$, $\neg$, $\equiv$, can be transformed into the combination of these two operators (For more details about the equivalent transformation from other operators to $\wedge$ and $\vee$, we refer readers to Section 2.5). In the following, we give some cases using these rules to simplify the conjunction and the disjunction of predicates. Without loss of generality, we apply these rules to two formulas named p and q.

   1. In the conjunction of p and q: $p \wedge q$

      1.1 For example if p is true, then we can simplify $p \wedge q$ to q by eliminating p in the conjunction.

      1.2 If any of p or q returns false, then it returns false.

   2. In the disjunction of p and q: $p \vee q$

      2.1 If any of p or q returns true, then it returns true.

2.2 For example, if p is false, then we can simplify $p \vee q$ to q by eliminating p.

For instance, we have a disjunction:

$\mathsf{true} \vee (x^5 - 3 * x^4 + 5 * x^3 - 2 = 0 \wedge \int_0^\infty \frac{x}{\sqrt{(x^2-25)}} dx = 0) \vee \forall(x \mid x \in \mathbb{R} \wedge x \geq 0 : x^2 \geq 4 \Rightarrow x \geq 2)$.

Using the $\vee - E$, we can easily get the result true without using a CAS to solve two equations $x^5 - 3 * x^4 + 5 * x^3 - 2 = 0$ and $\int_0^\infty \frac{x}{\sqrt{(x^2-25)}} dx = 0$ or a TP to handle the quantification $\forall(x \mid x \in \mathbb{R} \wedge x \geq 0 : x^2 \geq 4 \Rightarrow x \geq 2)$.

3. We give different priorities to the tasks that can be handled by a CAS.

While testing many kinds of symbolic expressions in [Wes99, pages 41-60] by CASs, the test suite returns results in various levels of successes (For more details about the level of successes, we refer readers to [Wes99, page 37]). Based on the results given in [Wes99], we also classify our CAS tasks into different levels of priorities. We set higher priority to the tasks which our CAS is more successful in computing. For more details, we consider the following example. Suppose that we have two kinds of symbolic expressions. One is the solving of a polynomial equation, e.g. $x^3 - 3 * x + 1 = 0$, and the other is the computing an integration, e.g. $\int_0^\infty \frac{x}{\sqrt{(x^2-25)}} dx$. In our work, we give higher priority to the first one since the ability a CAS to be successful in handling it is higher.

5.2 Heuristic

5.2.1 Design of the heuristic subsystem

The design of the sub-system *Heuristic* and its module guide are introduced as follows:

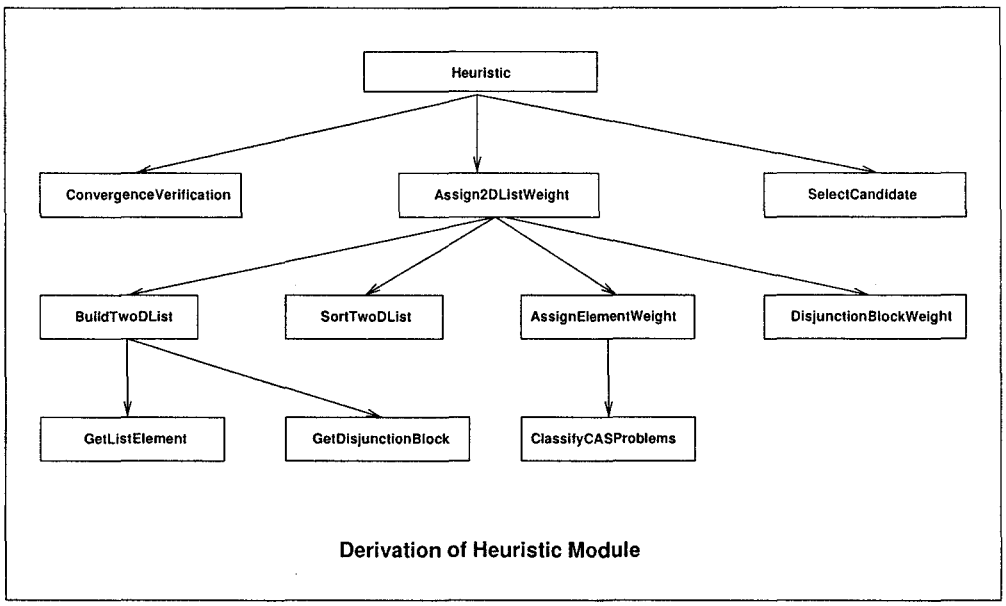

Figure 5.1: Derivation of Heuristic system

1. Module: Heuristic

   Service: Provides the heuristic rules to simplify relational calculus.

   Secret: The sequence of functions to provide the rules. Secret type: algorithm.

2. Module: BuildTwoDList

   Service: Builds a two-dimension list from a DNF relation.

   Secret: The algorithm is to build a two-dimension list from a DNF relation. Secret type: algorithm.

3. Module: GetListElement

   Service: Gets a predicate element of a row.

   Secret: The algorithm is to get a predicate element of a row. Secret type: algorithm.

4. Module: GetDisjunctionBlock

   Service: Gets a disjunction block from a two-dimension list.

   Secret: The algorithm to get a disjunction block. Secret type: algorithm.

5. Module: Assign2DListWeight

   Service: Assigns the weight for all elements of a two-dimension list.

   Secret: The algorithm to assign weights for these elements. Secret type: algorithm.

6. Module: AssignElementWeight

   Service: Assigns weight for each element of a two-dimension list.

   Secret: The algorithm to assign weight for each element. Secret type: algorithm.

7. Module: DisjunctionBlockWeight

   Service: Calculates the weight of a disjunction block.

   Secret: The algorithm to calculate a disjunction block. Secret type: algorithm.

8. Module: SortTwoDList

   Service: Sorts the weights of the row and the column of a two-dimension list.

Secret: The algorithm to sort the two-dimension list. Secret type: algorithm.

9. Module: SelectCandidate

   Service: Selects the predicate to simplify.

   Secret: The function to choose that predicate. Secret type: algorithm.

10. Module: InterpretPVSResult

    Service: Interprets the log file generated by PVS and provides PVS result file following the format we need.

    Secret: The algorithm to transfer PVS log file into the result file we need. Secret type: algorithm.

11. Module: InterpretMaximaResult

    Service: Interprets the log file generated by Maxima and provides Maxima result file following the format we need.

    Secret: The algorithm to transfer Maxima log file into the result file we need. Secret type: algorithm.

12. Module: ConvergenceVerification

    Service: Verifies the convergence of the simplification result.

    Secret: The algorithm to verify the convergence of the simplification result. Secret type: algorithm.

5.2.2 Prioritizing tasks

The options we provide in the strategy require us to set priority to the tasks handled by PVS and Maxima and the tasks handled among Maxima itself. A weight system to standardize the priority of these tasks in numbers is given.

5.2.2.1 Weight system

Let α be a DNF predicate such that $\alpha = \gamma_1 \vee \cdots \vee \gamma_k$ where each γ_i is a conjunction $\gamma_i = \beta_{i_1} \wedge \cdots \wedge \beta_{i_m}$.

We start by first assigning weights to each of the β_{ij} or to the expressions in their bodies. Then we assign weight to each of γ_i.

As introduced in Section 2.5, the relation that we consider is represented as a DNF predicate α. From α, we build a two-dimension table. A row i of the table is obtained from the corresponding disjunction block of the DNF (α_i). A disjunction block (γ_i) is constituted by sentence symbols or the negation of sentence symbols (β_{ij}). Each element of the table is a two-tuple in which the first member is obtained from the predicate of a disjunction block or from the expression that it contains and the second member is its initial weight. In addition, we take the first element of the row i to store the weight of γ_i. Initially, we assign the value 0 to the weight. Then, we assign the weight of each element based on the rules we introduce below. After that, we sort the rows of the table such that the highest weight of the row and its sorted elements become the first row, and so on. After sorting, we start to simplify predicates or expressions of the table. The predicate to simplify is chosen in the order of its presence in the table. Based on the weight, we choose PVS or Maxima to deal with the corresponding predicate or expression.

The rules to assign weight for β_{ij} are introduced in the following:

1. We set β_{ij} to 1, if the predicate is true or false.
2. We set β_{ij} to $\frac{1}{2}$, if the predicate can be directly handled by PVS.
3. We set β_{ij} to $\frac{1}{3}$, if the expression can be handled by Maxima in almost all the cases.
4. We set β_{ij} to $\frac{1}{4}$, if it is not guaranteed that the expression can be handled

by Maxima.

5. We set β_{ij} to $\frac{1}{99}$, if the predicate or the expression has been processed. Then, β_{ij} is sorted to become the last element of its row.

For example, we consider two conjunctions as follows:

$\gamma_1 = (x^3 - 3 * x^2 + 2)' = 0 \ \wedge \ \int_0^\infty (x^2 + 1)dx = 0$

$\gamma_2 = \exists(x \mid x \in \mathbb{N} : x \geq 0 \Rightarrow x \geq 3) \ \wedge \ 2 * x^2 - 5 * x + 2 = 0$

γ_1 consists of two expressions $(x^3 - 3 * x^2 + 2)' = 0$ and $\int_0^\infty (x^2 + 1)dx = 0$. We assign the value $\frac{1}{4}$ to both of them since they are not guaranteed to be solved successfully by Maxima.

γ_2 consists of one predicate (quantification) and one expression, $\exists(x \mid x \in \mathbb{N} : x \geq 0 \Rightarrow x \geq 3)$ and $2 * x^2 - 5 * x + 2 = 0$, respectively. We assign the value $\frac{1}{2}$ to the first one since it can be directly handled by PVS. We assign the value $\frac{1}{3}$ to the second one since solving a polynomial equation by Maxima is successful in almost all the cases.

Once each β_{ij} or the expressions forming it are weighed, we proceed to weigh the γ_i. For every γ_i, we assign to it the weight

$$\sum_{j=1}^{m} w_{\beta_{ij}} \tag{5.1}$$

where j is the index of an element of a row and $w_{\beta_{ij}}$ is the weight assigned to β_{ij}.

For illustration on how to evaluate weight of each γ_i, we take the example given above again.

The weight of γ_1 according to the equation 5.1 is equal to $\sum_{j=1}^{m} w_{\beta_{ij}} = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ and the weight of γ_2 is equal to $\sum_{j=1}^{m} w_{\beta_{ij}} = \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$.

Hence, we will proceed by handling first γ_2 then γ_1. We note that more in depth assessment of our weight system could reveal ideas for a better weight system. However, our heuristic is based on the weight given by the equation 5.1.

We note as well that the weight system we use does not always give the expected results. For instance, we have:

$\gamma_1 \equiv x^3 - 5 * x^2 + 7 * x - 3 = 0 \wedge (\int_0^\infty x^3 - 3 * x + 1\, dx = 0) \wedge (\sqrt{3 * x^2 + 7 * x + 13} = 3)$

$\gamma_1 \equiv (2 * x > 2 \Rightarrow x > 2) \wedge (3 * x^3 - 4 * x^2 - 5 * x + 2 = 0)$.

Following the rules of the weights given above, we assign the weights of elements of γ_1 corresponding to $\frac{1}{3}$, $\frac{1}{4}$, and $\frac{1}{3}$ and the weight of elements of γ_2 corresponding to $\frac{1}{2}$ and $\frac{1}{3}$. The weight of γ_1 is equal to $w_1 = \frac{1}{3} + \frac{1}{4} + \frac{1}{3} = \frac{11}{12}$. The weight of γ_2 is equal to $w_2 = \frac{1}{2} + \frac{1}{3} = \frac{10}{12}$. We see that $w_1 > w_2$, however, γ_2 is easier to solve than γ_1.

Each time we finish simplifying a predicate or an expression, we change its weight to $\frac{1}{99}$. Then, we sort the row. The recently processed element will become the last element and the next element will become the first one. We continue to deal with the elements of the row until all elements obtain the weight $\frac{1}{99}$. After processing a row completely, we sort the table. The recently processed row will become the last row of the table and the next row will become the first one. We continue to deal with the rest of the rows until all the elements are processed.

The general ideas introduced above are described in the following steps:

Step 1. Use module *BuildTwoDList* to build a two-dimension table from a DNF predicate.

Step 2. Use module *AssignTwoDListWeight* to assign corresponding weights for elements (β_{ij}) of each row and the weight of each row (γ_i) of the table.

Step 3. Use module *SortTwoDList* to sort the weights of the table. First, this module sorts the weights of the rows, then sorts the weights of each row.

Step 4. Use module *UpdateList* to update the result and the weight of each predicate or expression after it is processed by PVS or Maxima.

Step 5. Use module *Simplification* to simplify the DNF predicate. This mod-

ule use all the modules we introduce in the above steps for the simplification process.

Step 6. Iterate the simplification process by calling module *Simplification* many times to get a simplified result.

Step 7. Use module *ResultDisplay* to display the result of the simplified DNF predicate.

5.2.3 Illustrative examples

To illustrate how the system uses the heuristic rules that we propose, we give an example. We consider the following Diophantine representation of a relation:

$\forall(x \mid x \in \mathbb{R} : x^2 - 3 * x + 2 = 0 \ \wedge \ \int_0^\infty 2 * x - 1 \ dx = 0 \ \wedge \ 0 \leq x \leq 3)$

$\vee \ x > 2 \ \wedge \ (x^2 - 2 * x + 3)' = 0$

$\vee \ x > 2 \ \wedge \ True.$

The above quantification inside the example is equivalent to the following conjunction: $\forall(x \mid x \in \mathbb{R} : x^2 - 3 * x + 2 = 0) \ \wedge \ \forall(x \mid x \in \mathbb{R} : \int_0^\infty 2 * x - 1 \ dx = 0) \ \wedge \ \forall(x \mid x \in \mathbb{R} : 0 \leq x \leq 3)$.

The executions of the above steps 1, 2, 3 are illustrated respectively by Tables 5.1, 5.2, 5.3, 5.4, and 5.5 which represent the content of the two-dimension table built by *BuildTwoDList* module. The execution of step 4 is illustrated by Tables 5.5, 5.6, and 5.7. The execution of step 5 is illustrated by Table 5.8 and 5.9.

A cell in each of the following tables represents a predicate or an expression together with its weight.

Table 5.1: Build a table from an input DNF predicate.

(♯, 0)	$(x^2 - 3 * x + 2 = 0, 0)$	$(\int_0^\infty 2 * x - 1 = 0, 0)$	$(0 \leq x \leq 3, 0)$
(♯, 0)	$(x > 2, 0)$	$((x^2 - 2 * x + 3)' = 0, 0)$	
(♯, 0)	$(x > 2, 0)$	(True, 0)	

Table 5.2: Assign weight for predicates.

(♯, 0)	$(x^2 - 3 * x + 2 = 0, \frac{1}{3})$	$(\int_0^\infty 2 * x - 1 = 0, \frac{1}{4})$	$(0 \leq x \leq 3, \frac{1}{2})$
(♯, 0)	$(x > 2, \frac{1}{2})$	$((x^2 - 2 * x + 3)' = 0, \frac{1}{4})$	
(♯, 0)	$(x > 2, \frac{1}{2})$	(True, 1)	

Table 5.3: Calculate weight for a row.

(♯, 1.08)	$(x^2 - 3 * x + 2 = 0, \frac{1}{3})$	$(\int_0^\infty 2 * x - 1 = 0, \frac{1}{4})$	$(0 \leq x \leq 3, \frac{1}{2})$
(♯, 0.75)	$(x > 2, \frac{1}{2})$	$((x^2 - 2 * x + 3)' = 0, \frac{1}{4})$	
(♯, 1.50)	$(x > 2, \frac{1}{2})$	(True, 1)	

Table 5.4: Sort the table.

(♯, 1.50)	(True, 1)	$(x > 2, \frac{1}{2})$	
(♯, 1.08)	$(0 \leq x \leq 3, \frac{1}{2})$	$(x^2 - 3 * x + 2 = 0, \frac{1}{3})$	$(\int_0^\infty 2 * x - 1 = 0, \frac{1}{4})$
(♯, 0.75)	$(x > 2, \frac{1}{2})$	$((x^2 - 2 * x + 3)' = 0, \frac{1}{4})$	

Table 5.5: Change the weight of a predicate or an expression to $\frac{1}{99}$ each time it is processed and sort the row.

(♯, 1.50)	$(x > 2\ , \frac{1}{2})$	(True, $\frac{1}{99}$)	
(♯, 1.08)	$(0 \leq x \leq 3, \frac{1}{2})$	$(x^2 - 3 * x + 2 = 0, \frac{1}{3})$	$(\int_0^\infty 2 * x - 1 = 0, \frac{1}{4})$
(♯, 0.75)	$(x > 2, \frac{1}{2})$	$((x^2 - 2 * x + 3)' = 0, \frac{1}{4})$	

Table 5.6: Each time a row is processed completely, we sort the table

(♯, 1.08)	$(0 \leq x \leq 3, \frac{1}{2})$	$(x^2 - 3 * x + 2 = 0, \frac{1}{3})$	$(\int_0^\infty 2 * x - 1 = 0, \frac{1}{4})$
(♯, 0.75)	$(x > 2, \frac{1}{2})$	$((x^2 - 2 * x + 3)' = 0, \frac{1}{4})$	
(♯, 0.02)	$(x > 2, \frac{1}{99})$	(True, $\frac{1}{99}$)	

Table 5.7: Returns the original quantifier symbols back

$(x = 1, \frac{1}{99})$	$(x > 2, \frac{1}{99})$	
$(x > 2, \frac{1}{99})$	(True, $\frac{1}{99}$)	
$(\forall(x \mid x \in \mathbb{R} : x = 0 \vee x = 1), \frac{1}{99})$	$(\forall(x \mid x \in \mathbb{R} : 0 \leq x \leq 3), \frac{1}{99})$	$(\forall(x \mid x \in \mathbb{R} : x = 1 \vee x = 2), \frac{1}{99})$

Table 5.8: Use PVS to simplify predicates.

$x = 1$	$x > 2$	
$x > 2$	True	
false	$\forall(x \mid x \in \mathbb{R} : 0 \leq x \leq 3)$	$\forall(x \mid x \in \mathbb{R} : x = 1 \vee x = 2)$

Table 5.9: Use natural deduction to simplify the DNF predicate.

$x = 1$	$x > 2$	
$x > 2$		
false		

We obtain at the end the following result: $(x = 1 \wedge x > 2) \vee (x > 2)$ which can be simplified further to obtain $x > 2$.

5.3 Limitation

The following factors can impact the result and the usage of our system:

1. CRCS is built on two specific TP and CAS: PVS and Maxima. Therefore, it is limited by the limitation of these systems. For example, Maxima (and a CAS in general) fails to deal with many symbolic problems and PVS is not always successful in handling a theory.

2. The heuristic does not guarantee the success in all the cases.

3. For consistency and simplicity, we use linear notation to represent quantifications and symbolic expressions. However, this makes our system rigid since it requires the user to clearly give, in the notation, the nature of the operation to be performed.

4. Two modules *PVSTranslator* and *MaximaTranslator* of the implementation are able to translate only sub-languages of those handled by both tools PVS and Maxima.

5. In some cases, the system needs to iterate on the result many times to get the simplest result.

5.4 Tool Assessment

Each module that we developed was carefully tested. The whole system as it is at the current time has been assessed using examples similar to following.

1. $\forall(x \mid x \in \mathbb{R} : \sqrt{x^2+1} = x - 2 \ \wedge \ \int_0^\infty x^2 - 2 * x \ dx = 0)$

   $\vee \ \exists(x \mid x \in \mathbb{R} : x = 2 \Rightarrow x \geq 3)$

   $\vee \ x^2 = 4 \Rightarrow x = 2 \ \wedge \ (x^3 - 3 * x^2 + 5 * x)' = 0$

2. $(x^3 - 5 * x^2 + 7 * x - 3 = 0) \ \wedge \ (\int_0^\infty x^3 - 3 * x + 1 \ dx = 0) \ \wedge$
   $(\sqrt{3 * x^2 + 7 * x + 13} = 3)$

   $\vee \ ((x^3 - 4 * x)' = 0) \ \wedge \ (x \geq -3 \Rightarrow x \geq -4)$

   $\vee \ (2 * x > 2 \Rightarrow x > 2) \ \wedge \ (3 * x^3 - 4 * x^2 - 5 * x + 2 = 0)$

Applying module "BuildTwoDList.hs" to build the two-dimension list of the above second DNF predicate.

```
BuildTwoDList> buildList "solve(x | x in real: x^3 - 5 * x^2
+ 7 * x-3) AND solve(x | x in real: integrate(x | x in real:
x^3 - 3 *  x+1)=0) AND solve(x | x in real: sqrt(3 *  x^2 + 7
 *  x+13) =3) OR solve(x | x in real: diff(x | x in real:
 x^3 - 4 *  x)=0) AND lemma(x | x in int: x>=-3 IMPLIES x >= -4)
 OR lemma(x | x in nat: 2 *  x > 2 IMPLIES x > 2)
AND solve(x |  x in real: 3 *  x^3 - 4 *  x^2 - 5 *  x + 2 = 0)"
```

We then obtain the following list:

```
[[("#",0.0),("solve(x |  x in real:x^3-5* x^2+7* x-3)",0.0),
("solve(x |  x in real: integrate(x |  x in real: x^3 - 3* x+1)
=0)",0.0), ("solve(x |  x in real: sqrt(3* x^2 + 7* x+13) =3)",
0.0)],[("#",0.0),("solve(x |  x in real: diff(x |  x in real:
x^3-4* x)=0)",0.0), ("lemma(x |  x in int: x>=- 3
IMPLIES x >=-4)",0.0)],[("#",0.0), ("lemma(x |  x in nat:
2* x > 2 IMPLIES x > 2)",0.0), ("solve(x |  x in real:
3* x^3-4* x^2-5* x+2 = 0)",0.0)]]
```

Applying module *AssignTwDListWeight.hs* to assign the weights to the predicates or expressions of a DNF predicate.

```
AssignTwoDListWeight> assignTwoDListWeight [[("solve(x | x in
real:x^3-5*x^2+7*x-3)",0.0),("solve(x | x in real:
integrate(x | x in real: x^3-3*x+1)=0)",0.0),("solve(x | x in real:
sqrt(3*x^2+7*x+13) =3)",0.0)],[("solve(x | x in real:
diff(x | x in real: x^3-4*x)=0)",0.0),("lemma(x | x in int: x >=-3
IMPLIES x >= -4)",0.0)],\newline [("lemma(x | x in nat: 2*x > 2
IMPLIES x > 2)",0.0), ("solve (x  | x in real:
3*x^3 - 4*x^2 - 5*x + 2 = 0)",0.0)]]
```

The list is updated by adding the weight assigned to each cell. The result is the following.

```
[[("#",0.91),("solve(x| x in real:x^3-5*x^2+7*x-3)",0.33),
("solve(x| x in real: integrate(x| x in real: x^3-3*x+1)=0)",0.25),
("solve(x| x in real: sqrt(3*x^2+7*x+13) =3)",0.33)],
[("#",0.75),("solve(x| x in real: diff(x| x in real: x^3-4*x)=0)",
0.25),("lemma(x| x in int: x >=-3 IMPLIES x >= -4)",0.50)],
[("#",0.83),("lemma(x| x in nat: 2*x > 2 IMPLIES x > 2)",0.50),
("solve(x| x in real: 3*x^3 - 4*x^2 - 5*x + 2 = 0)",0.33)]]
```

Applying *SortTwoDList.hs* to sort the two-dimension list.

```
SortTwoDList> sortTwoDList [[("#",0.91),("solve(x | x in real:
x^3-5*x^2+7*x-3)",0.33),("solve(x | x in real:
integrate(x | x in real: x^3-3*x+1)=0)",0.25),("solve(x | x in real:
sqrt(3*x^2+7*x+13) =3)",0.33)],[("#",0.75),("solve(x | x in real:
diff(x | x in real: x^3-4*x)=0)",0.25),("lemma(x | x in int: x >=-3
IMPLIES x >= -4)",0.50)],\newline [("#",0.83),
("lemma(x | x in nat: 2*x > 2 IMPLIES x > 2)",0.50),
("solve(x | x in real: 3*x^3 - 4*x^2 - 5*x + 2 = 0)",0.33)]]
```

We obtain the following sorted list based on the decreasing weights of the first cell of each line in the list. The result is the following.

```
[[("#",0.91),("solve(x | x in real: sqrt(3*x^2+7*x+13) =3)",0.33),
("solve(x | x in real:x^3-5*x^2+7*x-3)",0.33),("solve(x | x in real:
integrate(x | x in real: x^3-3*x+1)=0)",0.25)],[("#",0.83),
("lemma(x | x in nat: 2*x > 2 IMPLIES x > 2)",0.50),
("solve(x | x in real: 3*x^3 -4*x^2 -5*x +2 = 0)",0.33)],
```

```
[("#",0.75),("lemma(x | x in int: x >=-3 IMPLIES x >= -4)",0.50),
("solve(x | x in real: diff(x | x in real: x^3-4*x)=0)",0.25)]]
```

Applying module *UpdateList.hs* to update the two-dimension list. The following is the illustration of how to update the elements of the first line of the list.

```
   UpdateList> updateElement [[("solve(x | x in real:
sqrt(3*x^2+7*x+13) =3)",0.33),("solve(x | x in real:
x^3-5*x^2+7*x-3)",0.33),("solve(x | x in real:
integrate(x | x in real : x^3-3*x+1)=0)",0.25)]
```

We obtain a new line in which the result of processed element is updated and sorted and it becomes the last element of the new line.

```
[[("solve(x | x in real:x^3-5*x^2+7*x-3)",0.33),
("solve(x | x in real: integrate(x | x in real: x^3-3*x+1)=0)",0.25),
("x = -4/3 OR x = -1",0.01)]
```

Illustration of how to update a whole row of the list.

```
   UpdateList> updateRowN [[("solve(x | x in real:
sqrt(3*x^2+7*x+13) =3)",0.33),("solve(x | x in real:
x^3-5*x^2+7*x-3)",0.33),("solve(x | x in real:
integrate(x | x in real : x^3-3*x+1)=0)",0.25)]
```

We get the following:

```
   [("x = 3 OR x = 1",0.01),("x = -4/3 OR x = -1",0.01),
("x = -sqrt(3)-1 OR x = sqrt(3)-1 OR x = 2 OR x = 0",0.01)]
```

Illustration of how to update the whole list.

```
UpdateList> updateTwoDListN [[("solve(x | x in real:
sqrt(3*x^2+7*x+13) =3)",0.33), ("solve(x | x in real:
x^3-5*x^2+7*x-3)",0.33), ("solve(x | x in real:
integrate(x | x in real: x^3-3*x+1)=0)",0.25)],
[("lemma(x | x in nat: 2*x > 2 IMPLIES x > 2)",0.50),
("solve(x | x in real: 3*x^3 - 4*x^2 - 5*x + 2 = 0)",0.33)],
[("lemma(x | x in int: x >=-3 IMPLIES x >= -4)",0.50),
("solve(x | x in real: diff(x | x in real: x^3-4*x)=0)",0.25)]]
```

We obtain the following relatively simplified list:

```
[[("x = 3 OR x = 1",0.01),("x = -4/3 OR x = -1",0.01),
("x = -sqrt(3)-1 OR x = sqrt(3)-1 OR x = 2 OR x = 0",0.01)],
[("False",0.01),("x = 1/3 OR x = -1 OR x = 2",0.01)],[("True",0.01),
("x = -2/sqrt(3) OR x = 2/sqrt(3)",0.01)]]
```

Further simplification.

```
Simplification>removeWeights [[("x = 3 OR x=1",0.01),("x = -4/3
OR x = -1",0.01),("x = -sqrt(3)-1 OR x = sqrt(3)-1
OR x = 2 OR x = 0",0.01)], [("False",0.01),
("x = 1/3 OR  x = -1 OR x = 2",0.01)],[("True",0.01),
("x = -2/sqrt(3) OR x = 2/sqrt(3)",0.01)]]
```

We get the following list:

```
[["x = 3 OR x = 1","x = -4/3 OR x = -1","x = -sqrt(3)-1
OR x = sqrt(3)-1 OR x = 2 OR x = 0"],["False","x = 1/3
OR x = -1 OR x = 2"],["True","x = -2/sqrt(3) OR x = 2/sqrt(3)"]]
```

Applying natural deduction rules on the DNF.

```
Simplification> simplifyTwoDList [["x = 3 OR x = 1","x = -4/3
OR x = -1","x = -sqrt(3)-1 OR x = sqrt(3)-1 OR x = 2 OR x = 0"],
["False","x = 1/3 OR x = -1 OR x = 2"],["True","x = -2/sqrt(3)
OR x = 2/sqrt(3)"]]
```

We obtain the following list:

```
[["x = 3 OR x = 1","x = -4/3 OR x = -1","x = -sqrt(3)-1
OR x = sqrt(3)-1 OR x = 2 OR x = 0"],["False"],["x = -2/sqrt(3)
OR x = 2/sqrt(3)"]]
```

Displaying the simplified DNF predicate.

```
Simplification> displaySimplification [["x = 3 OR x = 1","x = -4/3
OR x = -1", "x = -sqrt(3)-1 OR x = sqrt(3)-1 OR x = 2 OR x = 0"],
["False"],["x = -2/sqrt(3) OR x = 2/sqrt(3)"]]
```

We obtain the following result:

```
"(x = 3 OR x = 1 AND x = -4/3 OR x = -1 AND x = -sqrt(3)-1
OR x = sqrt(3)-1 OR x = 2 OR x = 0) OR (x = -2/sqrt(3)
OR x = 2/sqrt(3))"
```

5.5 Conclusion

In this approach, we present a relation as a predicate given in a DNF. We call each item of a DNF predicate an element. An element can be an expression (taken from a quantification), a proposition, or a Boolean value.

In this chapter, we discussed how to use a heuristic for scheduling PVS and Maxima to simplify DNF predicates. First, a strategy is created to provide general options for the heuristics. Then, a set of specific heuristic rules developed based

on the strategy. These rules introduce a weight system to evaluate the priority of elements of the DNF predicate and the combination of their corresponding conjunctions. Two corresponding kinds of weights are given: weight for each element of a DNF predicate and weight for conjunctions of elements. The weight of a conjunction is calculated based on the sum of inverse of the weights of its elements. After assigning weights, the simplification process is carried out. The process uses the rules of the weight system we proposed for assigning PVS and Maxima to work until finishing the last element of the predicate. The simplification process can be iterated many times. Our design includes a module called *ConvergenceVerification* that makes sure that the process stops.

Chapter 6

Conclusion and Future Work

Mechanized mathematics tools play a very important role in verifying properties of engineering artifacts in general and software in particular. We presented a proof-of-concept tool that uses current mechanized mathematics technologies to support relational calculus. We illustrated how a CAS meets difficulties in giving the truth value of predicates while a TP is not capable to handle symbolic expressions. To deal with this problem, we proposed the usage of a combination of a TP and a CAS. We first reviewed the literature on TPs and CASs. Then, we shortly described the tools that deal with relation algebras. We gave the rationale behind the selection of PVS and Maxima. Then, we proposed a heuristic that enables the assignment of tasks to either PVS or Maxima. Finally, we designed and built a proof-of-concept tool based on the proposed heuristic that uses PVS and Maxima to support relational calculus.

6.1 Contribution

The main contributions of our thesis are the following.

- Explore a heuristic to schedule the tasks handled by PVS and Maxima.

- Design and implement a proof-of-concept tool that uses existing technologies to help perform relational calculus. We note that some modules of the system are not completed (for example, the interface module). However, the critical modules of the system have been developed and tested.

6.2 Future work

As a follow up on the work presented in this thesis, we propose the following as future work:

- We assume that the predicate is represented in a DNF. However, predicates are not always represented in this format. Thus, adding a module which can transform any predicate into its DNF is necessary.

- We fall short of providing a user interface which make the use of the tool slightly difficult to the user.

- We adopted a rigid linear notation for the representation of quantifications and expressions. Exploring other notations to represent them is needed for a more flexible tool.

- We used some default strategies to prove theorems (e.g., strategy "grind"). The user-defined PVS strategies should be saved in a file called "pvs-strategies". Currently, we do not allow users to access "pvs-strategies" file directly. One should consider allowing users to input their own strategies.

- Defining Haskell data structures that enable us to capture PVS syntax and Maxima syntax would be of a great need for a thorough translation to PVS and Maxima languages. Currently, both translation modules handle only subsets of languages of both tools.

The main point that rises from the research exercise carried through this thesis work is that the currently available mechanized mathematics technologies can be used to ease relational calculus. However, we conjecture that the development of an integrated environment where both computer algebra and theorem prover are merged together would make a tool such the one we propose in this thesis unnecessary.

Bibliography

[ABB93] P. B. Andrews, M. Bishop, and C. E. Brown. TPS: An Interactive and Automatic Tool for Proving Theorems of Type Theory, 1993. http://gtps.math.cmu.edu/hug93.ps. [Last accessed: 25/01/2006].

[Art95] R. D. Arthan. Notes on PVS from a HOL perspective, August 1995. http://www.cl.cam.ac.uk/users/mjcg/PVS.html. [Last accessed: 13/01/2006].

[BBMS98] R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RELVIEW - A System for Calculating With Relations and Relational Programming. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering*, volume 1382, pages 318–321. Springer, 1998.

[Bla92] N. Blachman. *Mathematica: A Practical Approach.* Prentice Hall, 1992.

[Bud93] D. Budgen. *Software Design.* Addison-Wesley, 1993.

[Col96] Collins. *COBUILD Learner's Dictionary.* HarperCollins, 1996.

[End72] H. B. Enderton. *A Mathematical Introduction to Logics.* Academic Press, 1972.

[FGT93] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Automated Reasoning*, 1993.

[FvM00] W. M. Farmer and M. v. Mohrenschildt. Transformers for symbolic computation and formal deduction. In *S. Colton, U. Martin, and V. Sorge, editors, Proceedings of the Workshop on the Role of Automated Deduction in Mathematics,*, pages 36–45, June 2000.

[GH06] D. Griffioen and M. Huisman. A Comparison of PVS and Isabelle/HOL, 2006. http://www-sop.inria.fr/lemme/Marieke.Huisman/Comparison.html. [Last accessed: 12/01/2006].

[GJM03] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 2003.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GS93] D. Gries and F. B. Schneider. *A Logical Approach To Discrete Math.* Springer, 1993.

[Hat98] C. Hattensperger. RALF manual, 1998. http://www2-data.informatik.unibw-muenchen.de/Research/Tools/RALF. [Last accessed: 13/01/2006].

[Hec93] A. Heck. *Introduction to MAPLE.* Spring-Verlag, 1993.

[Hel91] B. Heller. *Macsyma for statiscians.* John Wiley - Sons, Inc, 1991.

[HS99] D. Hoffman and P. Strooper. *Software Design, Automated Testing and Maintenance: A Practical Approach.* International Thompson Computer Press, September 1999.

[JS92] R. D. Jenks and R. S. Sutor. *Axiom: The scientific computation system.* Springer-Verlag, NewYork, 1992.

[KH98] W. Kalh and C. Hattensperger. Second-Order Syntax in HOPS and in RALF. In *Program Systems for Computer-Aided System Development and Verification*, volume 1 of *BISS Monographs*, pages 140–164. Shaker Verlag, Aachen, 1998.

[KV00] B. Kutzler and V. K. Voljc. *Introduction to Derive 5.* Texas Instruments, 2000.

[Mar06] A. Marchenkova. A Comparison of PVS and Isabelle/HOL. http://www.ags.uni-sb.de/ chris/lectures/fol-hol-tp/ Anna-Marchenkova.pdf, 2006. [Last accessed: 12/01/2006].

[Mat06] Mathscheme: An Integrated Framework For Computer Algebra And Computer Theorem Proving, 2006. http://imps.mcmaster.ca/mathscheme. [Last accessed: 10/04/2006].

[MuP06] Mupad, 2006. http://www.mupad.de/. [Last accessed: 08/06/2006].

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logic.* Springer, 2002.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Lecture Notes in Artificial Intelligence*, volume 607, pages 748–752. Spring-Verlag, 1992.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Language Reference. Technical report, SRI International, 2001.

[Owr04] S. Owre. PVS Introduction. http://pvs.csl.sri.com/introduction.shtml, April 2004. [Last accessed: 13/01/2006].

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules.. In *Communications of the ACM*, pages 15(12):1053–1058, December. 1972.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[PP99] J. F. Peters and W. Pedrycz. *Software Engineering - An Engineering Approach.* John Wiley & Sons, Inc, 1999.

[Ray87] G. Rayna. *REDUCE: software for algebraic computation.* Springer-Verlag, 1987.

[Rel06] Relmics-relational methods in computer science, 2006. http://www2-data.informatik.unibw-muenchen.de/relmics/html. [Last accessed: 31/03/2006].

[Rus03] S. Rusovan. Inspecting the source code that implements the PPP. Master's thesis, McMaster University, 2003.

[SG05] C. M. So and H. Gottliebsen. Maple-PVS Superuser Guide. Technical report, Department of Computer Science Queen Mary, University of London, 2005.

[So06] C. M. So. Interfacing Maple and PVS, February 2006. http://www.cas.mcmaster.ca/ socm/documents/

MacTalkFeb2006/McMasterTalkFeb06-so.pdf. [Last accessed: 16/02/2006].

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide. Technical report, SRI International, November 2001.

[SS93] G. Schmidt and T. Strohlein. *Relations and Graphs: Discrete Mathematics for Computer Scientists.* Spring-Verlag, 1993.

[vOG97] D. v. Oheim and T. F. Gritzner. RALL: Machine-supported Proofs for Relation Algebra. In W. McCune, editor, *Conference on Automated Deduction – CADE-1*, number 380-394 in BISS Monographs. Springer, 1997.

[Web05] Maxima manual, October 2005. http://maxima.sourceforge.net/docs/manual/en/maxima.html. [Last accessed: 12/02/2006].

[Wes99] M. J. Wester. *Computer Algebra Systems: A practical guide.* John Wiley - Sons, Ltd, 1999.

[Wik06a] Computer Algebra System. http://en.wikipedia.org/wiki/Computer_algebra_system, 2006. [Last accessed: 12/01/2006].

[Wik06b] Wikipedia. Heuristic, June 2006. http://en.wikipedia.org/wiki/Heuristics. [Last accessed: 03/07/2006].

[Wol88] S. Wolfram. *Mathematica: A system for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, Inc, 1988.

Appendix A

Module Guide of the System

A Module Guide provides the service and the secret of each module. The service describes the functions the module gives. The secret indicates the likely change encapsulated by the module. In the following, we introduce the module guide of CRCS.

(1) *Name:* **Main Module**

Service: Integrates together all the functions of the system.

Secret: The algorithm to integrate the functions of the system. Secret type: software decision hiding (algorithm).

(2) *Name:* **Operation_Selection Module**

Service: Integrates together all the functions relating to the calculus operations.

Secret: The algorithm to integrate these functions. Secret type: software decision hiding (algorithm).

(3) *Name:* **Setup Module**

Service: Allows users to setup working directories for the system.

Secret: The input formats of the working directories. Secret type: behavior hiding (input formats).

(4) *Name:* **Union Module**

Service: Computes the union of the two relations.

Secret: The algorithm to compute the union of the two arguments. Secret type: software decision hiding (algorithm).

(5) *Name:* **Intersection Module**

Service: Computes the intersection of the two relations.

Secret: The algorithm to compute the intersection of the two arguments. Secret type: software decision hiding (algorithm).

(6) *Name:* **Composition Module**

Service: Computes the composition of the two relations.

Secret: The algorithm to compute the composition of the two arguments. Secret type: software decision hiding (algorithm).

(7) *Name:* **Inverse Module**

Service: Computes the inverse of a relation.

Secret: The algorithm to compute the inverse of the argument. Secret type: software decision hiding (algorithm).

(8) *Name:* **Complement Module**

Service: Computes the complement of a relation.

Secret: The algorithm to compute the complement of the argument. Secret type: software decision hiding (algorithm).

(9) *Name:* **CheckCond1 Module**

Service: Checks whether the union or the intersection operation is defined or not.

Secret: The algorithm to check the condition. Secret type: software decision hiding (algorithm).

(10) *Name:* **CheckCond2 Module**

Service: Checks whether the composition operation is defined or not.

Secret: The algorithm to check the condition. Secret type: software decision hiding (algorithm).

(11) *Name:* **BuildUnion Module**

Service: Builds and produces a new relation which characterises the union of the relations.

Secret: The algorithm to build the union. Secret type: software decision hiding (algorithm).

(12) *Name:* **UnionOutput Module**

Service: Outputs the computed union relation file.

Secret: The screen format to return the relation result. Secret type: behavior hiding (screen formats).

(13) *Name:* **BuildIntersection Module**

Service: Builds and produces a new relation which characterises the intersection of the relations.

Secret: The algorithm to build the intersection. Secret type: software decision hiding (algorithm).

(14) *Name:* **IntersectionOutput Module**

Service: Outputs the computed intersection relation file.

Secret: The screen format to return the relation result. Secret type: behavior.

(15) *Name:* **BuildComposition Module**

Service: Builds and produces a new relation which characterises the composition of the relations.

Secret: The algorithm to build the composition. Secret type: software decision hiding (algorithm).

(16) *Name:* **CompositionOutput Module**

Service: Outputs the computed composition relation file.

Secret: The screen format to return the relation result. Secret type: behavior.

(17) *Name:* **BuildComplement Module**

Service: Builds and produces a new relation which characterises the complement of the relation.

Secret: The algorithm to build the complement. Secret type: software decision hiding (algorithm).

(18) *Name:* **ComplementOutput Module**

Service: Outputs the computed intersection relation file.

Secret: The screen format to return the relation result. Secret type: behavior.

(19) *Name:* **BuildInverse Module**

Service: Builds and produces a new relation which characterises the inverse of the relation.

Secret: The algorithm to build the complement. Secret type: software decision hiding (algorithm).

(20) *Name:* **InverseOutput Module**

Service: Outputs the computed intersection relation file.

Secret: The screen format to return the relation result. Secret type: behavior.

(21) *Name:* **PredicateSimplification Module**

Service: Integrate all the modules related to simplifying an input predicate.

Secret: The sequence of the modules to simplify an input predicate. Secret type: software decision hiding (algorithm).

(22) *Name:* **PredicateHolder Module**

Service: Uses a format file to hold the data structure of the input predicate.

Secret: The algorithm to store the predicate. Secret type: software decision hiding (algorithm).

(23) *Name:* **PVSTranslator Module**

Service: Translates an input predicate into the format which can be read by PVS.

Secret: The syntax of PVS. Secret type: machine hiding (virtual machine).

(24) *Name:* **PVSInterface Module**

Service: Uses PVS to make the derivation of the input predicate and generates the log file for the result proof of the theorem.

Secret: The language to call the batch mode of PVS to make the derivation. Secret type: machine hiding (virtual machine).

(25) *Name:* **PVSInterpretResult Module**

Service: Interprets the log file generated by PVS and provides the PVS theorem result file.

Secret: The algorithm to transfer PVS log file into the result file we need. Secret type: software decision hiding (algorithm).

(26) *Name:* **MaximaTranslator Module**

Service: Translates an input predicate into the format which can be read by Maxima.

Secret: The syntax of Maxima. Secret type: machine hiding (virtual machine).

(27) *Name:* **MaximaInterpretResult Module**

Service: Interprets the log file generated by Maxima and provides the result file in the form we need.

Secret: The algorithm to transfer Maxima log file into the result file we need. Secret type: software decision hiding (algorithm).

(28) *Name:* **MaximaInterface Module**

Service: Uses Maxima to compute the input predicate.

Secret: The language used to invoke Maxima. Secret type: software decision hiding (algorithm).

(29) *Name:* **List Update**

Service: Updates the result returned by Maxima or PVS to the corresponding processed expression or predicate.

Secret: The algorithm to update the result. Secret type: software decision hiding (algorithm).

(30) *Name:* **String Module**

Service: Builds a list of functions for managing strings of the system.

Secret: The data type String. Secret type: software decision hiding (data structure).

(31) *Name:* **MessageDisplay Module**

Service: Displays the contents of the system messages.

Secret: The format of text messages. Secret type: behavior hiding (text message).

Appendix B

Detailed Design of the System

We give the access program and detailed information of a selection of modules that can give a good idea about the overall system.

PVSTranslator Module

The module reads the input predicate file, translates its content, and returns a theory file which can be read by PVS. The content of input file and output file must follow strictly the format of our predicate and PVS theory, respectively. It contains the following main access functions:

1. $predicateToPVSString :: String \rightarrow String$

   It translates an input predicate string into a string which can be read by PVS.

2. $translateProposition :: String \rightarrow String$

   It translates a proposition predicate.

3. $translateComplex :: String \rightarrow String$

   It translates a quantification.

4. $beginTheory :: String$

   It declares theory name which is appropriate to put at the beginning of a PVS theory.

5. $importNewTheory :: String \rightarrow String$

   It displays all the declarations of imported theories.

6. $getAllTheories :: String \rightarrow [String]$

   It gets all theories obtained from the input predicate.

7. $getQuantRange :: String \rightarrow [String]$

   It gets quantification ranges obtained from the input predicate.

8. $displayVarDeclaration :: String \rightarrow String$

   It displays declarations of all variables obtained from the input predicate.

9. $getListOfVarAndType :: String \rightarrow [(String, String)]$

   It gets list of tuples of variable and its corresponding type obtained from the input predicate.

10. $getTupleOfVarAndType :: String \rightarrow (String, String)$

    It gets a tuple of variable and its corresponding type obtained from the input predicate.

11. $translatePropLemma :: String \rightarrow String$

    It translates proposition lemma.

12. $translateComplexLemma :: String \rightarrow String$

    It translates quantification lemma.

13. $translateContextBody :: String \rightarrow String$

    It translates context body of the input predicate.

14. $getQuantifierSymbols :: String \rightarrow [String]$

    It gets all quantifier symbols of the input predicate.

15. $numberOfQuantifierSymbols :: String \rightarrow Int$

    It returns the number of quantifier symbols of the input predicate.

16. $remDuplicate :: Orda => [a] \rightarrow [a]$

    It removes duplicate theory from the theory list.

17. $separateByComma :: String \rightarrow [String]$

    It returns a list of strings separated by commas.

18. $numberOfColons :: String \rightarrow Int$

    It returns the number of colons of the input predicate.

19. $combineList :: [a] \rightarrow [a] \rightarrow [a]$

    It combines two lists which have the same type.

20. $endTheory :: String$

    It displays the ending of a PVS theory.

MaximaTranslator Module

The module reads input predicate file, translates its content, and returns a theory file which can be read by Maxima. The content of input file and output file must follow strictly the format of our predicate and Maxima expressions, respectively. It contains the following main access functions:

1. $predicateToMaximaString :: String \rightarrow String$

   It translates an input predicate string into a string which can be read by Maxima.

2. $translateProposition :: String \rightarrow String$

   It translates an input proposition predicate.

3. $getMaximaKeywords :: String \rightarrow [String]$

   It gets Maxima keywords obtained from the input predicate.

4. $numberOfMaximaKeywords :: String \rightarrow Int$

   It returns the number of Maxima keywords obtained from the input predicate.

5. $numberOfQuantifierSymbols :: String \rightarrow Int$

   It returns the number of quantifier symbols of the input predicate.

6. $numberOfColons :: String \rightarrow Int$

   It returns the number of colons of the input predicate.

7. $getLowerBound :: String \rightarrow String$

   It gets the lower bound of an integration.

8. $getUpperBound :: String \rightarrow String$

   It gets the upper bound of an integration.

9. $separateByComma :: String \rightarrow [String]$

   It returns a list of strings separated by commas.

10. *elementOfTheoryList* :: *String* → *Bool*

    It checks whether a theory obtained from the input predicate is a member of declared PVS theory list or not.

PVSInterface Module

This module provides the interface to communicate with PVS system. It uses the PVS batch mode and the default strategies to verify the submitted theories. These files are passed to PVS which generates ".log" file containing the result.

MaximaInterface Module

This module provides the interface to communicate with Maxima system. It uses the Maxima batch mode to simplify the symbolic expressions or solve the equations. The files containing expressions are passed to it. When Maxima completes its task, it generates ".log" file containing the result.

Predicate Module

This module provides the interface to get or set the relation. The data structure of a relation can be one of three types: a Boolean value, a formula relation represented by a string, or a record which is defined recursively.

The data type of the relation is represented as follows:

```
data Predicate a = Pcmplx                    -complex predicate
{ symbol :: Quantifsymbol
, variables :: [String]
, quantrange :: Pred                         -quantifier range
, cxbody :: Predicate a }                    -context body
| Ppred { smbody :: Pred }                   -proposition predicate
| Pbool { blbody :: Bool } deriving Show     -Boolean predicate
```

This module contains the following access functions:

1. $getVariables :: String \rightarrow String$

   It gets the list of variables obtained from the input predicate string.

2. $getNthVariable :: Int \rightarrow String \rightarrow String$

   It gets the Nth variable of the input predicate string.

3. $getQuantifrange :: String \rightarrow String$

   It returns the quantifier range obtained from the input predicate string.

4. $getQuantifier :: String \rightarrow String$

   It returns the quantifier symbol obtained from the input predicate string.

5. $getPredicate :: String \rightarrow String$

   It returns the sub-Predicate from the input predicate string.

6. $isPBool :: String \rightarrow Bool$

   It determines whether the input is a Boolean predicate or not.

7. $isPpred :: String \rightarrow Bool$

   It determines whether the input is a proposition predicate or not.

8. $isPcmplx :: String \rightarrow Bool$

   It determines whether the input is a quantification or not.

9. $showPredicate :: Predicate\ a \rightarrow String$

   It shows the predicate.

10. $setSymbol :: Predicate\ a \rightarrow Quantifsymbol \rightarrow Predicate\ a$

    It sets a quantifier symbol of the predicate.

11. $setQuantrange :: Predicate\ a \rightarrow Pred \rightarrow Predicate\ a$

    It sets a quantifier range of the predicate.

12. $setCxbody :: Predicate\ a \rightarrow Predicate\ a \rightarrow Predicate\ a$

    It sets a context body of the predicate.

13. $setSmbody :: Predicate\ a \rightarrow Pred \rightarrow Predicate\ a$

    It sets a symbol of the proposition predicate.

14. $setBlbody :: Predicate\ a \rightarrow Bool \rightarrow Predicate\ a$

    It sets Boolean body of the predicate.