Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

DUAN, Zhiping

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.Sc. (Computer Science)

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Proof Search Algorithms for Detecting Interactions in Telecommunication Features

A. Felty

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

L. Bertossi

L. Logrippo

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Proof Search Algorithms for Detecting Interactions in Telecommunication Features

A Thesis

Presented to the Faculty of Graduate and Postdoctoral Studies

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

in Candidacy for the Degree of

Master of Science

Faculty of Engineering

University of Ottawa

By

Zhiping Duan

Thesis Director: Professor Amy Felty

Your file Votre référence
ISBN: 0-612-90061-4
Our file Notre référence
ISBN: 0-612-90061-4

ABSTRACT

Proof Search Algorithms for Detecting Interactions in Telecommunication Features

Zhiping Duan

University of Ottawa

April 2003

This thesis presents a theorem proving approach to automatically detecting feature interactions in telecommunications systems. Feature requirements are formalized as specifications using Linear Temporal Logic (LTL) formulas and feature conflicts are specified by LTL formulas expressing logical inconsistency of specifications of two requirements of two different features. These LTL formulas are translated into First-Order Logic (FOL) formulas. We begin with an existing FOL theorem prover and optimize and enrich this prover for our application. In particular, we develop algorithms for integers and relational expressions which appear in translated FOL formulas and integrate these algorithms into the existing proof search procedure. Implementation and experiments in λProlog demonstrate that our theorem prover is efficient and powerful when applied to the task of feature interaction detection. We also develop a proof checker to verify proofs obtained from our prover.

Acknowledgements

I am deeply indebted to my advisor, Dr. Amy Felty for her constant instructional and financial support. Without her help, this thesis would not be possible. Her ideas, suggestions and patience have been invaluable to this thesis.

Special thanks go to my brother, Zhiyong Duan for his insightful suggestions and assistance at every stage of my work.

Many thanks are sent to Yong Shi, my fiance, for taking care of me. With his love, I have never felt alone.

I thank my eternal friends, Xiulin He, Yong Zhang and the group of 94402 to make my life colorful and full of fun.

Lastly but most important in my heart, I would like to thank my parents for their endless support, love and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

We address a problem in the domain of feature interactions (FIs) in telecommunications systems. During the software development life-cycle of telephony systems, features, such as Call Forwarding and Call Waiting, are designed individually and developed independently to add new functions to the system. When new features are added to meet particular requirements of a system, they may conflict with existing features in some unexpected ways. For example, when a new feature, such as Call Forwarding, which provides a service that a call will be forwarded to another registered phone number if the callee's line is busy, is added to the system including Call Waiting, which gives the callee a signal when he/she is already talking on the line, a conflict happens in the special case when there is a user who subscribes to these two features simultaneously. The reason is that an incoming call cannot be responded to in two different ways, such as both being forwarded and being given a signal, when the callee is already on the line.

Such conflicts are best resolved as early as possible ideally before

such features are implemented. Our work is motivated by the desire to detect feature conflicts at the *specification* stage. This may help prevent time-consuming and high-cost investment in implementations based on incorrect specifications. And this may also help us fix the requirements of features. For example, a solution to solve the conflict above is to give higher priority to Call Forwarding than to Call Waiting.

We describe the general approach and its advantages in section 1.2 and give our contributions in section 1.3. Section 1.4 contains an outline of the rest of this thesis.

1.2 Our Approach

1.2.1 Our Approach in General

In our approach, feature requirements are specified in linear temporal logic (LTL) [3, 11] according to their informal descriptions [1] and feature conflicts are expressed by LTL formulas as logical inconsistency of two LTL formulas representing particular requirements of two different features [8, 9]. Our objective is to develop a theorem prover to search for proofs for LTL formulas. This prover has two important characteristics: it is fully automated and specialized for this particular application.

The background and a general proposal for the approach we follow is outlined by Felty [7]. In this thesis, we explain the overall approach and show how to fulfil the complete design in practice. Our implementation and experiments in λProlog illustrate how the potential of the theorem proving approach is achieved.

1.2.2 Advantages

Related to our approach are a variety of recent approaches to the problem. Temporal logic is sometimes used to specify features [3, 11]. Model checking tools are often used to help in the automatic detection of feature conflicts [9, 4, 13, 19]. Actually, the logical background of our approach originated from the approach in the first reference where conflicts are considered logical inconsistency between two requirements of feature specifications and we use the same feature specification language. The difference between these two approaches is that model checking is applied in that reference but theorem proving is chosen in our approach.

The main novel aspects of our approach in general are:

- A special prover for this application potentially improves on performance. General-purpose provers, such as Otter [16], may come with some additional performance costs because it is not specialized to our particular task. The performance improvement will allow us to consider conflicts possibly caused by features which are more difficult to prove.

- To allow more general uses of quantification. Comparing with the model checking approach which restricts us in some sense to consider particular instantiations of specification formulas, theorem proving can directly handle quantifications in the feature specification language. This is included in our future work.

1.2.3 Description of Our Approach

Following the outline of the theorem proving approach in reference [7], we develop algorithms for handling problems we meet in practice. By

integrating these algorithms into an existing theorem prover [5], we define a proof search algorithm for an automated prover which can detect feature conflicts.

The steps of our approach allow us to reach a point where our specific tool is efficient and complete for the special task we are addressing. They are:

1. Specifying feature requirements formally in LTL from their informal descriptions. There are informal descriptions for features in the telephony system and these descriptions can be specified formally by logical connectives and atomic formulas. For example, any call *always* has a property that the call is *eventually* terminated by *either* the callee's response *or* the caller's phone put back on hook. By using atomic formulas to represent basic behaviors, such as $call(x, y)$ for a call from y to x and $onhook(y)$ for representing the fact that the caller y puts his/her phone on hook, we formalize requirements in LTL which includes connectives, such as *always* represented by the symbol $\Box$, literal *or* specified by $\vee$, etc., to obtain their corresponding LTL formulas.

2. Specifying feature conflicts as logical inconsistency of LTL formulas. According to a general definition of detecting feature conflicts, given by Felty and Namjoshi [9], we give an LTL formula to specify the conflict caused by two feature requirements of a specific form. Therefore, the problem of detecting feature conflicts in the telecommunications systems is transformed to a problem of searching for a proof of the LTL formula by a theorem prover.

3. Translating LTL formulas into FOL formulas based on the semantics of logical connectives. There are many choices in automating temporal logic and we choose to translate LTL into FOL. This

translation function can be viewed as a formalization of the semantics of LTL in FOL with integers. Because we are concentrating on a particular application, our formulas have a specific form which can be decomposed into simpler subformulas given as input to the prover.

4. Specifying particular algorithms to handle the translated FOL formulas. A proof tree for a FOL formula with integers is built by applying inference rules in a sequent calculus. We present special rules for integers and develop algorithms to apply them during proof search. We integrate these algorithms into the main proof search procedure to develop a complete automated theorem prover for detecting feature conflicts.

We implement our approach in λProlog which is shown to be well-suited to the implementation of theorem provers [6]. Our experiments in this language illustrate that our theorem proving approach to detecting feature conflicts is successful.

1.3 Contribution of the Thesis

Among the steps of our approach described in subsection 1.2.3, steps 1, 2 and 3 are previous work which we build on. The main contributions of this thesis are in step 4. They are:

- Optimized the existing theorem prover by finding an order in which to attempt inference rules during proof search which is best for our application.
- Developed the specialized algorithms needed for integer arithmetic and integrated them into the main proof search procedure.

- Implemented our approach and illustrated its potential with experiments in λProlog.

1.4 Organization of the Thesis

The thesis is organized as follows:

1. In Chapter 1, the introduction to the thesis is provided which also includes our motivation and approach to the specific task.

2. Some background on FOL, a sequent proof system, LTL, and λProlog, which are needed for the thesis, is presented in Chapter 2.

3. In Chapter 3, we present the implementation of inference rules for FOL in the sequent proof system, and then we review an algorithm for the existing automated theorem prover which we later build on.

4. In Chapter 4, we introduce the previous work on specifying feature requirements and feature conflicts as logical inconsistency in linear temporal logic respectively. We show how to translate LTL formulas into FOL formulas with integers and how to decompose them into several simpler formulas used later by the prover. This decomposition is important in the general form of proofs for our application. Moreover, we present our representation of these formulas in λProlog which will be used by our prover.

5. The main contributions of this thesis are presented in Chapter 5. We discuss our algorithms for handling relational expressions in the translated FOL formulas. By employing these algorithms, we obtain the new relational expressions which are needed for our proof search procedure. Integrating these algorithms into the existing

theorem prover results in a completely automated proof procedure for our specific task.

6. Results from our implementation and experiments are briefly discussed in Chapter 6.

7. We conclude in Chapter 7 by discussing further performance improvement and future work.

Chapter 2

Background

In this chapter, we provide background on a sequent proof system for first-order classical logic, linear temporal logic and λProlog.

2.1 Sequent Proof System for First-Order Classical Logic

A sequent is an expression $\Gamma \longrightarrow \Delta$ where Γ and Δ are finite sets of formulas $\gamma_1, \gamma_2, \cdots, \gamma_m$ and $\delta_1, \delta_2, \cdots, \delta_n$ respectively [12]. A sequent $\Gamma \longrightarrow \Delta$ in classical logic holds if there is a proof where the *conjunction* of $\gamma_1, \gamma_2, \cdots, \gamma_m$ implies the *disjunction* of $\delta_1, \delta_2, \cdots, \delta_n$. Particularly, if Γ is empty, the sequent asserts that $\delta_1 \vee \delta_2 \vee ... \vee \delta_n$ is true. In contrast, if Δ is empty, it asserts that $\gamma_1 \wedge \gamma_2 \wedge ... \wedge \gamma_m$ is false. Formulas on both sides of a sequent are first-order logic formulas.

Definition 1. *φ is a first-order logic (FOL) formula, defined in Backus Naur Form (BNF) [14]:*

$$\varphi ::= A \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \supset \varphi) \mid (\forall x\varphi) \mid (\exists x\varphi) \qquad (2.1)$$

where A is an atomic formula, which is a formula $p(t_1, t_2, ..., t_n)$ where p

is a predicate of arity n with n terms t_i $(i = 1, \cdots, n)$ as parameters; x is a bound variable for quantifiers; connectives ($\neg$, $\wedge$, $\vee$, and $\supset$) and quantifiers ($\forall$ and $\exists$) have the following binding priorities [14]:

1. $\neg$, $\forall x$ and $\exists x$ bind most tightly;
2. then $\wedge$, $\vee$;
3. then $\supset$.

A proof for a sequent $\Gamma \longrightarrow \Delta$ follows inference rules [5] summarized in Figure 2.1.

$$\frac{\Gamma \longrightarrow A, \Delta \qquad \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \wedge B, \Delta}\ \wedge\text{-R} \qquad \frac{A, B, \Gamma \longrightarrow \Delta}{A \wedge B, \Gamma \longrightarrow \Delta}\ \wedge\text{-L}$$

$$\frac{\Gamma \longrightarrow A, B, \Delta}{\Gamma \longrightarrow A \vee B, \Delta}\ \vee\text{-R} \qquad \frac{A, \Gamma \longrightarrow \Delta \qquad B, \Gamma \longrightarrow \Delta}{A \vee B, \Gamma \longrightarrow \Delta}\ \vee\text{-L}$$

$$\frac{A, \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \supset B, \Delta}\ \supset\text{-R} \qquad \frac{\Gamma \longrightarrow A, \Delta \qquad B, \Gamma \longrightarrow \Delta}{A \supset B, \Gamma \longrightarrow \Delta}\ \supset\text{-L}$$

$$\frac{A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \neg A, \Delta}\ \neg\text{-R} \qquad \frac{\Gamma \longrightarrow A, \Delta}{\neg A, \Gamma \longrightarrow \Delta}\ \neg\text{-L}$$

$$\frac{\Gamma \longrightarrow [y/x]A, \Delta}{\Gamma \longrightarrow \forall x A, \Delta}\ \forall\text{-R} \qquad \frac{[t/x]A, \Gamma \longrightarrow \Delta}{\forall x A, \Gamma \longrightarrow \Delta}\ \forall\text{-L}$$

$$\frac{\Gamma \longrightarrow [t/x]A, \Delta}{\Gamma \longrightarrow \exists x A, \Delta}\ \exists\text{-R} \qquad \frac{[y/x]A, \Gamma \longrightarrow \Delta}{\exists x A, \Gamma \longrightarrow \Delta}\ \exists\text{-L}$$

$$\frac{}{A, \Gamma \longrightarrow A, \Delta}\ \text{initial}$$

The $\forall$-R and $\exists$-L rules have the proviso that the variable y cannot appear free in the lower sequent.

Figure 2.1: The Sequent Proof System for First-Order Classical Logic

There are one left rule and one right rule for each connective and quantifier. $\wedge$-L and $\vee$-R rules express the fact that on the left side of a sequent there are intrinsic "*and*" relations among formulas while on the right side there are intrinsic "*or*" relations. The rules for $\wedge$-R, $\vee$-L,

$\supset$-R, $\supset$-L, $\neg$-R and $\neg$-L similarly express logical consequences that the upper sequent or sequents imply the lower sequent. For example, the rule for $\supset$-R, reading backward, says the lower sequent $(\Gamma \longrightarrow A \supset B, \Delta)$ is true if the upper sequent $(A, \Gamma \longrightarrow B, \Delta)$ is true. In rules for $\forall$-L and $\exists$-R, $[t/x]A$ denotes substitution, which means each *free* occurrence of x is replaced by the term t in A. An expression $[y/x]A$ in rules $\forall$-R and $\exists$-L is also substitution where y replaces x in A with a restriction that y must not be free in the lower sequent. The last rule is the *initial* rule which says a sequent is true if there is a common formula on both sides of the sequent. Among all these rules, the *initial* rule is the only rule which needs to be applied at least once if the sequent is provable.

In proof search, we try to build a proof tree *backward* whose root is the sequent to be proved. At each step, an inference rule is chosen, and the upper sequent or sequents are computed from the lower sequent according to the rule. The formula in the lower sequent to which the rule is applied is removed from the list and the appropriate subformulas are added to form the upper sequents. Removing formulas at each step simplifies sequents and makes them ever easer to read. In this way, the upper sequents are simpler in the sense that they contain one less connective than the lower sequent and all leaves are instances of the *initial* rule. Using a sequent calculus helps us understand proofs and encourages us to study the application by tracing the proof search procedure.

Continuing to employ the above inference rules, the tree grows upwards, branches if needed, until it reaches its leaves, the *initial* sequents which contain a common formula on both sides of a sequent.

We use one sequent

$$\exists x\varphi \vee \exists x\psi \longrightarrow \exists x(\varphi \vee \psi)$$

as an example to show how a proof tree is built in Figure 2.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[x_0/x]\varphi \longrightarrow [x_0/x]\varphi, [x_0/x]\psi}\text{initial}}{[x_0/x]\varphi \longrightarrow [x_0/x]\varphi \vee [x_0/x]\psi}\vee\text{-R}}{[x_0/x]\varphi \longrightarrow \exists x(\varphi \vee \psi)}\exists\text{-R}}{\exists x\varphi \longrightarrow \exists x(\varphi \vee \psi)}\exists\text{-L} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{[x_0/x]\psi \longrightarrow [x_0/x]\varphi, [x_0/x]\psi}\text{initial}}{[x_0/x]\psi \longrightarrow [x_0/x]\varphi \vee [x_0/x]\psi}\vee\text{-R}}{[x_0/x]\psi \longrightarrow \exists x(\varphi \vee \psi)}\exists\text{-R}}{\exists x\psi \longrightarrow \exists x(\varphi \vee \psi)}\exists\text{-L}$$

$$\dfrac{\exists x\varphi \longrightarrow \exists x(\varphi \vee \psi) \qquad \exists x\psi \longrightarrow \exists x(\varphi \vee \psi)}{\exists x\varphi \vee \exists x\psi \longrightarrow \exists x(\varphi \vee \psi)}\vee\text{-L}$$

Figure 2.2: An Example of a Proof Tree Built for a Sequent

In this figure, the root is labelled by the sequent to be proved, the tree grows upwards, branches are increasing when inference rules are applied. The left branch grows upward until reaches its *initial* sequent, which contains a common formula $[x_0/x]\varphi$. Then the right one achieves its *initial* sequent including $[x_0/x]\psi$ on both sides of the sequent.

In general, rule names are given beside the rule application in each step. For example, $\vee$-L appears beside the sequent in the root when $\vee$-L is applied.

2.2 Linear Temporal Logic

2.2.1 Linear Temporal Logic

Linear-time temporal logic (LTL) is a type of modal logic [14, 15] that, by including special temporal connectives, can be applied to describe time dependent properties such as features in the telephony system.

Definition 2. *An LTL formula φ is defined as, in Backus Naur Form [14],*

$$\begin{aligned}\varphi ::= \; & A \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \supset \varphi) \\ & \mid (\Box\varphi) \mid (\Diamond\varphi) \mid (\varphi \,\mathsf{U}\, \varphi) \mid (\varphi \,\mathsf{W}\, \varphi) \mid (\mathsf{X}\varphi)\end{aligned} \tag{2.2}$$

where A is an atomic formula with no logical connectives; in addition to FOL connectives, there are five special temporal connectives $\Box$, $\Diamond$, U, W, and X.

An LTL formula is validated on a path or set of paths in a model [14, 15]. A model is defined as a finite set of states (S), a set of transitions (R) of pairs of states and a set of labels (L) consisting of atomic formulas. Each state s_i in S is labelled by atomic formulas from L. A path π is an infinite sequence of states $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$, in which s_0 is the start state and the token $\longrightarrow$ indicates a transition in R. For example, $s_i \longrightarrow s_j$ is a transition from s_i to s_j.

For a given path, we proceed to explain the semantics for temporal connectives in Figure 2.3 according to explanations in references [14, 15].

LTL formula	Description	Semantics Meaning
$\Box\varphi$	always	φ is satisfied at **every state** on the given path
$\Diamond\varphi$	eventually	φ is satisfied at **some state** on the given path
$\varphi_1 U \varphi_2$	until	following a given path there **exists** a state at which φ_2
		holds and before which φ_1 holds from the start state
$\varphi_1 W \varphi_2$	weak until	$(\Box\varphi_1 \vee (\varphi_1 U \varphi_2))$
$X\varphi$	next	φ is satisfied at the **next state** on the given path

Figure 2.3: Semantics for Temporal Connectives

Among these temporal connectives, new unary connectives have the same priority as $\neg$ and new binary connectives have the same priority as $\wedge$ and $\vee$.

The notation π^i denotes a path π starting from the state s_i. If a path π satisfies an LTL formula φ, which means φ is true on this path π, the satisfaction relation is denoted by $\pi \models \varphi$ which is an abbreviation of $\pi^0 \models \varphi$. In general, $\pi^i \models \varphi$ is defined by induction on the structure of φ as follows [14]:

1. For an atomic formula A, $\pi^i \models A$ iff* A is one of the labels assigned to s_i.

2. $\pi^i \models \neg\varphi$ iff $\pi^i \models \varphi$ does not hold.

3. $\pi^i \models (\varphi_1 \wedge \varphi_2)$ iff both $\pi^i \models \varphi_1$ and $\pi^i \models \varphi_2$.

4. $\pi^i \models (\varphi_1 \vee \varphi_2)$ iff $\pi^i \models \varphi_1$ or $\pi^i \models \varphi_2$.

5. $\pi^i \models (\varphi_1 \supset \varphi_2)$ iff $\pi^i \models \varphi_2$ whenever $\pi^i \models \varphi_1$.

6. $\pi^i \models \Box\varphi$ iff for all $k \geqslant i$, $\pi^k \models \varphi$.

7. $\pi^i \models \Diamond\varphi$ iff for some $k \geqslant i$, $\pi^k \models \varphi$.

8. $\pi^i \models (\varphi_1 \,\mathsf{U}\, \varphi_2)$ iff there exists some $k \geqslant i$ such that $\pi^k \models \varphi_2$ and $\pi^j \models \varphi_1$ for every $j = i, \cdots, k-1$.

9. $\pi^i \models (\varphi_1 \,\mathsf{W}\, \varphi_2)$ iff $\pi^i \models \Box\varphi_1$ or $\pi^i \models (\varphi_1 \,\mathsf{U}\, \varphi_2)$.

10. $\pi^i \models \mathsf{X}\varphi$ iff $\pi^{i+1} \models \varphi$.

An LTL formula is true in a model if it is satisfied by all paths in the model.

2.2.2 Translation from LTL Formulas to FOL Formulas

There are many approaches to validating LTL formulas. The one we adopt is to translate them into first-order logic formulas. There are also several well-known translations and we choose a straightforward one.

The translation function (f) takes two parameters, an LTL formula (φ) and a state number (i), which is added in FOL to describe the state

*If and only if.

s_i, at which φ holds. For example, an atomic LTL formula $p(t_1, t_{2,...}t_n)$ is translated into an atomic FOL formula $p(t_1, t_2, \ldots, t_n, i)$ to describe the fact that $p(t_1, t_2, \ldots, t_n)$ is true at the state s_i. The translation begins by computing $f(\varphi, 0)$, which expresses the fact that the formula holds at the start state. The translation function is defined inductively [7] in Figure 2.4 following the definition of the satisfaction relation shown in subsection 2.2.1.

$$
\begin{array}{rcl}
f(p(t_1, t_2, ..., t_n), i) & := & p(t_1, t_2, ..., t_n, i) \\
f(\neg\varphi, i) & := & \neg f(\varphi, i) \\
f(\varphi_1 \wedge \varphi_2), i) & := & f(\varphi_1, i) \wedge f(\varphi_2, i) \\
f(\varphi_1 \vee \varphi_2), i) & := & f(\varphi_1, i) \vee f(\varphi_2, i) \\
f(\varphi_1 \supset \varphi_2), i) & := & f(\varphi_1, i) \supset f(\varphi_2, i)) \\
f(\Box\varphi, i) & := & \forall j(j \geqslant i \supset f(\varphi, j)) \\
f(\Diamond\varphi, i) & := & \exists j(j \geqslant i \supset f(\varphi, j)) \\
f(\varphi_1 \,\mathsf{U}\, \varphi_2), i) & := & \exists j(j \geqslant i \wedge f(\varphi_2, j) \\
 & & \wedge\forall k(i \leqslant k < j \supset f(\varphi_1, j))) \\
f(\varphi_1 \,\mathsf{W}\, \varphi_2), i) & := & f(\varphi_1 \mathsf{U}\, \varphi_2), i) \vee \ f(\Box\varphi_1, i) \\
f(\mathsf{X}\varphi, i) & := & f(\varphi, i + 1)
\end{array}
$$

Figure 2.4: Translation from LTL Formulas to FOL Formulas

In our application, there are also $\forall$ and $\exists$ in the LTL formulas. However, we do not include translations for them because these quantifiers are used in restricted forms. For example, the universal quantifiers appear in the following two formulas at the top: $\forall x \forall y \forall z(\Box\varphi(x, y, z))$ and $\Box\forall x \forall y \forall z(\varphi(x, y, z))$. We handle them as special cases. In this case the latter formula is equivalent to the former one.

2.3 λProlog

2.3.1 Overview of λProlog

λProlog is a higher-order logic programming language whose formulas are higher-order logic formulas that permit quantifiers over predicate

and function symbols [17]. The logical foundation of this language is an extension of Prolog which has more expressions to allow logical connectives, such as implication and universal quantifiers, to appear in goals and bodies of the program clauses.

Here are features of λProlog used by our later implementation [17]:

- Modular programming

Clauses in λProlog are organized into modules. Each module begins with keyword **module** followed by the module name and is stored in the file with the module name and the extension name **mod**. For example, the file **a1.mod** contains the module **a1**.

There are constants, variables and clauses in a module which may accumulate other modules by the keyword **accumulate**. Thus constants and clauses defined in the accumulated modules can also be used in the current module. We summarize the structure of a module of five parts: a name, a list of accumulated modules (if any), kind declarations, type declarations and definite clauses. For example, Figure 2.5 illustrates how to organize a module.

In this figure, the module **a2** accumulates the module **a1** and **memb_and_rest** declared in **a1** is used in **a2** with no necessary re-declaration.

- Strong typing and polymorphic typing:

λProlog is a strongly typed language which means that all constants must be declared with their types. Variables in λProlog begin with an upper-case letter and constants begin with a lower-case letter. The types of variables in formulas are automatically inferred according to the context and the type for each constant must be declared before it is first used.

```
%% module name
module a1.
%% type declaration
type memb_and_rest A -> list A -> list A -> o.

...

module a2.
%% accumulated module a1
accumulate a1.
%% kind declaration
kind seq type.
...
%% type declaration
type >- lprf -> seq -> o.
...
%% definite clauses
(and_l A B Q) >- (Gamma --> Delta) :-
 memb_and_rest (A and B) Gamma Gamma1,
 Q >- ((A::B::Gamma1) --> Delta).
```

Figure 2.5: Example Modules in Teyjus

λProlog is polymorphic, which allows *variables* to appear in type declarations, called *type variables*. For example, the constant **memb** has the following declaration:

```
type memb A -> list A -> o.
```

where **memb** is a predicate with two parameters of types **A** and **list A**. **A** is a *type variable* which states that this parameter can be instantiated by an instance of any type. Note that the detailed explanation of the constant **memb** is presented later.

- Definite Clauses and Goal formulas

There are two different types of formulas in λProlog, definite clauses (denoted by D) and goal formulas (denoted by G). They can be defined recursively as below:

$G := \mathrm{T} \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G \mid \forall x G$

$D := A \mid G \supset A \mid \forall x D$

where A is an atomic formula.

As we know, λProlog is an extension of Prolog. The important distinction between them is that the connective $\supset$ and the qualifier $\forall$ can be allowed to appear in both goal formulas and definite clauses in λProlog which are forbidden in the goal formulas in Prolog.

Definite clauses and goal formulas have different purposes. On one hand, definite clauses are grouped into modules which are regarded as *facts*. On the other hand, goal formulas are queried in the command line and the query is answered based on those facts stored in the loaded modules. For example, we declare the following definite clauses for the predicate **memb** which are used to decide whether X is an element of a list L:

```
memb X (X :: L).
memb X (Y :: L) :- memb X L.
```

where the token :: is an operator to concatenate an element into a list; :- is an operator which means "implied by" and used to write the top level connective in the definite clause.

In this example, the first clause says that **memb X (X::L)** is true if X is the first element of the list. The second clause states that **memb X (Y::L)** is true if **memb X L** is true where L is a sublist of the given list without the first element Y. Then any query **memb X L** is true if either of these two clauses is satisfied.

Based on the facts above, a goal formula **memb 2 (1::2::nil)**, where **nil** denotes an empty list, is true because 2 is the first element of the sub-list **(2::nil)**. Note that the type variable **A** in the type declaration of **memb** is automatically instantiated by type **int**.

- Simply-typed λ-terms

The terms of λProlog are the simply-typed λ-terms [17] and they are more expressive than first-order terms. The application of λ-terms in λProlog uses β-conversion, a basic operation of λ-calculus. Bound variables can begin with an initial letter in either upper-case or lower-case. In order to represent λxA, the backslash \ is used as an infix operator to represent it as $x\backslash A$ or $X\backslash A$. $((\lambda xA)\ T)$ denotes the application of the term (λxA) to T and β-conversion in λProlog will replace x in A by T.

- Unification of λ-terms

Unification of λ-terms in λProlog is handled by higher-order unification. It is used to instantiate *logical variables*. For example, if there are one formula **(X and y)** on the left side and a formula **(x and y)**

on the right side of a sequent, then variable **X** is instantiated with the constant **x** by unification. This is implemented as a built-in operation by the interpreter/compiler of λProlog. Since higher-order unification is undecidable [10], implementation of any interpreter for λProlog is incomplete. But for the cases in first-order logic, there is no such problem. In addition, implementation and experiments illustrate that this problem does not occur in practice for most applications, including ours.

2.3.2 Teyjus

There are several implementation of λProlog. We choose *Teyjus*, a C-based compiler because it is intended to be efficient and (eventually) robust for λProlog [18] and is appropriate for complex applications. Compared with some other implementations, *Teyjus* greatly improves on performance of our application.

2.3.3 Syntax of λProlog

Because we use *Teyjus* to implement our application, the *Teyjus* syntax of λProlog is provided in this subsection for understanding of our later implementations.

Since we discuss one logic, first-order logic, within another logic, higher-order logic of λProlog, in order to avoid confusion, we differentiate the former logic, called the *object-logic* from the latter one, called the *meta-logic*.

We begin the discussion on syntax with type declarations about constants.

There are two different declarations: **kind** and **type**. The keyword **kind** is used to introduce user-defined type constructors and then they

can be used in type declarations to build user-defined terms. Types are constructed by a binary infix constructor $\longrightarrow$. For example, a predicate p with arity n can be declared with type: $\tau_1 \longrightarrow \tau_2 \longrightarrow ... \longrightarrow \tau_n \longrightarrow \tau_{n+1}$. If $t_1, t_2, \ldots, t_n$ are terms of type $\tau_1, \tau_2, \ldots, \tau_n$ respectively, then $p(t_1, t_2, \ldots, t_n)$ is of type τ_{n+1}. The association of $\longrightarrow$ is to the right, which means $\tau_1 \longrightarrow \tau_2 \longrightarrow ... \longrightarrow \tau_n \longrightarrow \tau_{n+1}$ is an abbreviation of $\tau_1 \longrightarrow (\tau_2 \longrightarrow (... \longrightarrow (\tau_n \longrightarrow \tau_{n+1})..))$. An example of type declarations shown in subsection 2.3.1 has already demonstrated how to construct a type using $\longrightarrow$:

```
type memb A -> list A -> o.
```

where the constant **memb** is a term, which takes 2 parameters of type **A** and **list A** respectively. The formula **memb X Y** is of type **o**; `->` is the token in *Teyjus* for the constructor $\longrightarrow$.

Note that there are two other constants in the above declaration, **list** and **o**, which also need to be declared before they are used. The declarations of these constants are called *kind declarations*, which are used to declare types and *type constructors*. These two constants have the following definitions:

```
kind list type -> type.
kind o  type.
```

where the constant **list** in λProlog is used to represent a *list* which builds a new type by accepting one parameter. For example, **list A** is a type which is built from **list** with one parameter **A**. The constant **o** is used to represent a formula in λProlog.

Theoretically, these two constants need to be declared before **memb**'s declaration. In practice, since they are built-in constants, we can employ them anywhere anytime.

A new binary infix constant :: is used to concatenate an element to a list and a unary constant **nil** is used to indicate an empty list. They have their own explicit declarations as follows although it is not necessary to define them in the application since they are also built-in constants:

```
type :: A -> list A -> list A.
type nil list A.
infixr :: 5.
```

The keyword **infixr** here means that this constant is represented in an infix format and associates to the right. The number in the **infix** declaration represents the binding priority of the constant. In this case the number 5 is the number for ::. Particularly in *Teyjus*, the number for the user's self-defined constants should be $\geqslant 200$ and the larger the number is, the tighter the constant's binding priority is.

Therefore, we can construct lists by these constants. For example, $(1 :: 2 :: 3 :: nil)$ is a list in λProlog. $(2 :: L)$ is another example of a list whose first element is 2 and whose tail is L.

Besides these built-in constants, there are others we use in our application. We merely express their functions without presenting their explicit declarations. The first two, infix tokens , and ; represent connectives $\wedge$ and $\vee$ in λProlog respectively. The last two are universal and existential quantifiers in the meta-logic, constants **pi** and **sigma** which are represented in conjunction with a λ-term. For example, the constant **pi** is declared:

```
type pi (A -> o) -> o.
```

where **pi** takes a predicate of type **A -> o** as a parameter; the bound variable in the predicate can be of any type, represented by **A**. For example, we can have a meta-logic formula **pi x\ F** which represents $\forall x F$.

Then user-defined terms may be built by built-in type constructors and pre-declared type constructors. All declarations for constants needed for the application are grouped together into a signature[†].

We declare definite clauses after type declarations and continue to use the same example of definite clauses for the constant **memb** explained in subsection 2.3.1:

```
memb X (X :: L).
memb X (Y :: L) :- memb X L.
```

We have explained these two clauses above.

There are variables in the definite clauses and goal formulas, such as X, Y and L in the above example. The convention for these variables is: variables in the definite clauses are assumed to be universally quantified and variables in the goal formula are assumed to be existentially quantified [17]. Therefore, the above definite clauses are equivalent to the following clauses:

```
pi X\ pi L\ (memb X (X :: L)).
pi X\ pi Y\ pi L\ (memb X (Y :: L) :- memb X L).
```

which mean variables X, Y and L can be instantiated by any terms.

A goal formula **memb X (1::2::nil)**, as an example, is equivalent to **sigma X\ (memb X (1::2::nil))**, which means that the formula is true as long as there is one instantiation of **X**. We prefer to use the concise formats to simplify the presentations of definite clauses and goal formulas.

[†]Particularly, in *Teyjus*, there is a signature file which has the extension name of .sig

Chapter 3

Theorem Proving and Proof Checking in λProlog

This chapter describes an approach to developing an automated theorem prover and a proof checker for FOL formulas in the sequent proof system. Since our proofs for FOL formulas use inference rules, we present our approach by implementing them in λProlog and then provide strategies to organize and control these implementations in order to achieve a prover for automatically searching for proofs for FOL formulas. Moreover, an example in FOL is demonstrated to explain how the prover works as a successfully automated proof search tool. Finally, we discuss briefly a complementary tool, a proof checker, which is used to verify the proofs that we have obtained from the prover. Discussions on this topic are illustrated by implementations in λProlog. They are based on an existing prover and checker presented in reference [5].

3.1 An Automated Prover for FOL Formulas

As we know, a given FOL formula A with no assumptions can be written as a sequent: $\longrightarrow A$ whose left side is null. The procedure to find a proof for a sequent, called "proof search", is based on fairly employing inference rules presented in Figure 2.1. In order to achieve proof search, basically, we implement sequents and inference rules in λProlog. Moreover, we figure out strategies which can lead to an automatic proof search procedure. Finally, we present an example to illustrate how this prover works.

3.1.1 Implementation of Inference Rules for First-Order Classical Logic

In order to implement FOL formulas in λProlog, one primitive type **form** is declared to construct object-logic formulas.

```
kind form type.
```

By using the binary infix arrow constructor `->`, all connectives and quantifiers can be declared by using **form** in Figure 3.1.

```
type and form -> form -> form.
type or form -> form -> form.
type imp form -> form -> form.
type neg form -> form.
type forall (A -> form) -> form.
type exists (A -> form) -> form.
infixl and 250.
infixl or 250.
infixr imp 240.
```

Figure 3.1: Declarations for FOL Connectives and Quantifiers in λProlog

In the above declarations, constants **and**, **or**, **imp**, **neg**, **forall** and **exists** are used to represent logical connectives $\wedge$, $\vee$, $\supset$, $\neg$ and quantifiers $\forall$ and $\exists$ respectively. The constant **and**, for example, takes 2 parameters of type **form** representing two formulas, and constructs the conjunction of these two formulas. The notation of **and** is infix and associates to the left. Thus, **(X and Y and Z)** represents **((X and Y) and Z)**. For the other connectives, **or**, **imp** and **neg**, we have similar explanations for them. For example, $(X \vee Y), (X \supset Y)$ and $(\neg X)$ are represented by **(X or Y)**, **(X imp Y)** and **(neg X)** respectively in λProlog. The slight differences among them are that **neg** has the highest binding priority and **imp** is right associative and has the lowest binding priority. Quantifiers $\forall$ and $\exists$ have functional parameters. For example, the constant **forall**, takes a λ-term as a parameter of type **(A -> form)** inside which a bound variable is of any type **A**. Therefore, $\forall x A$ is represented in the object-logic as **forall X\ A** or **forall x\ A**.

Besides these declarations for connectives and quantifiers, predicates in FOL also need to be defined. For example, predicates **p** and **q** are declared:

```
type p A -> form.
type q A -> form.
```

which are two unary predicates accepting one parameter of any type.

We have declarations for FOL formulas below:

```
kind name type.
type formula name -> form -> o.
formula pre_1 (forall X\ exists Y\
                ((p X) or (q Y)) imp
                 (exists Y\ forall X\ ((p X) or (q Y)))).
```

where the predicate **formula** has two parameters, a special formula name of type **name** and the value for the formula of type **form**; **pre_1** is a name for the formula $\forall x \exists y(p(x) \vee q(y)) \supset \exists y \forall x(p(x) \vee q(y))$.

Now we are going to implement inference rules as follows.

Inference rules in the sequent proof system for first-order classical logic are written using the tree structures. Each node represents a sequent. A new primitive type **seq** is introduced to represent the object-logic sequent. Lists of elements of type **form** are used to represent formulas in a sequent. The constant **seq** , by using the infix binary operator **-->**, is constructed by two parameters of type **list form** as below:

```
kind seq type.
type --> (list form) -> (list form) -> seq.
```

Thus a sequent ($\Gamma \longrightarrow \Delta$) is presented in λProlog as (**Gamma - ->Delta**), where **Gamma** and **Delta** are lists of formulas of type **list form**. Generally speaking, during the proof search procedure, it is better to store the proof for a sequent to review and later check. So the primitive type **lprf** is used to build the sequent proof and another infix binary operator >- is used to express the relation between a sequent and its proofs.

```
kind lprf type.
type >- lprf -> seq -> o.
```

Therefore if we have **Q** as a proof for the sequent ($\Gamma \longrightarrow \Delta$), this can be presented as **Q >- (Gamma - -> Delta)** in λProlog. Our objective is to automatically search for **Q** by applying inference rules to the given sequent.

The inference rules are declarative expressing the logical relation that the lower sequent is true if the upper one or ones are true. The deterministic logic programming compiler *Teyjus* provides us implementation of a depth-first proof search by applying these rules in a backward direction.

The constant declarations for the λProlog data structures representing each rule are in Figure 3.2:

```
type and_l form -> form -> lprf -> lprf.
type and_r form -> form -> lprf -> lprf -> lprf.
type or_l  form -> form -> lprf -> lprf -> lprf.
type or_r  form -> form -> lprf -> lprf.
type imp_r form -> form -> lprf -> lprf.
type imp_l form -> form -> lprf -> lprf -> lprf.
type neg_r form -> lprf -> lprf.
type neg_l form -> lprf -> lprf.
type exists_r A -> lprf -> lprf.
type exists_l (A -> lprf) -> lprf.
type forall_r (A -> lprf) -> lprf.
type forall_l A -> lprf -> lprf.
```

Figure 3.2: Declarations for Inference Rules for FOL in λProlog

We explain the constant **and_l** as an example. It is used to store a proof employing the ∧-L rule, which takes the first two parameters of type **form** to store A and B, the subformulas of $(A \wedge B)$ and the third parameter of type **lprf**, which is used to record the proof of the upper sequent. If **Q** is such a proof of this sequent, then **(and_l A B Q)** is a proof for the lower sequent.

We have the following definite clause for the rule **and_l**:

```
(and_l A B Q) >- (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1,
  Q >- ((A::B::Gamma1) --> Delta).
```

where a new λProlog predicate **memb_and_rest** is used to take out a formula from a list if this formula is a member of this list. This element-extracting operation is used to guarantee termination of the proof search procedure because in each step after employing one rule, the upper sequent is simpler than the lower sequent since it contains one less connective or quantifier. Then there are appropriate subformulas added to the upper sequent in accordance with the logical relation between the upper and lower sequents. In this example, **Gamma1** is a new list without element $(A \wedge B)$ and **(A::B::Gamma1)** is a list obtained by adding two elements A and B to **Gamma1**.

Operationally, this clause for the constant **and_l** says: if a sequent has the form $(A \wedge B, \Gamma \longrightarrow \Delta)$ and **Q** is the proof for the upper sequent, then **(and_l A B Q)** is the proof for the lower sequent. Notice that the order of elements in the sequent is not important at all during implementation of all rules. For example, in this case, we do not care whether $(A \wedge B)$ is the first element of the list or not as long as it is a member of this list.

Now, consider the rule ∧-R:

```
(and_r A B Q1 Q2) >- (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >- (Gamma --> (A::Delta1)),
  Q2 >- (Gamma --> (B::Delta1)).
```

Since the rule ∧-R separates the lower sequent into two upper sequents, the predicate **and_r** takes 4 parameters: Two of them indicate A and B which are subformulas of $(A \wedge B)$ in the lower sequent, and the other two are proofs for each separate subgoal. The clause can be read as: if **Q1** is a proof of $(\Gamma \longrightarrow A, \Delta)$ and **Q2** is a proof of $(\Gamma \longrightarrow B, \Delta)$, then **(and_r A B Q1 Q2)** is a proof of $(\Gamma \longrightarrow A \wedge B, \Delta)$. Note that we also remove $(A \wedge B)$ from Δ and add A and B separately to Δ in the

upper sequents.

We have very similar explanations for $\vee$-L, $\vee$-R, $\supset$-L, $\supset$-R, $\neg$-L and $\neg$-R rules.

Let us look at the implementation of quantifiers which are more complex than the previous ones.

The rule $\exists$-R, for example, is captured by the following definite clause:

```
(exists_r T Q) >- (Gamma --> Delta) :-
  memb_and_rest (exists A) Delta Delta1,
  Q >- (Gamma --> ((A T)::Delta1)).
```

Since bound variables for quantifiers in FOL are represented by bound variables in simply-typed λ-terms, term **A** is such a λ-term in this example. **(A T)** substitutes **T** for the bound variable in **A**. Substitution is done automatically by λProlog .

We can read this clause as: If **Q** is a proof of $\Gamma \longrightarrow [T/x]A, \Delta$, then **(exists_r T Q)** is a proof of $\Gamma \longrightarrow \exists x A, \Delta$. In λProlog, **T** is a *logic variable* and will be instantiated by higher-order unification later by applying the **initial** rule.

Also, we have a similar explanation for the rule $\forall$-L.

```
(forall_l T Q) >- (Gamma --> Delta) :-
  memb_and_rest (forall A) Gamma Gamma1,
  Q >- (((A T)::Gamma1) --> Delta).
```

This clause says: if there is $\forall x A$ in Γ and **Q** is a proof of $\Gamma, [T/x]A \longrightarrow \Delta$, then **(forall_l T Q)** is a proof of $\Gamma, \forall x A \longrightarrow \Delta$.

Implementation of rules $\forall$-R and $\exists$-L are complex because we need to implement the proviso for these two rules. Here is the implementation of the $\forall$-R rule:

```
(forall_r Q) >- (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  pi T \ ((Q T) >- (Gamma --> ((A T)::Delta1))).
```

Because the additional proviso for this rule says: y is not free in the lower sequent, we use the universal quantifier **pi** in λProlog to implement the proviso that **T** is not free in λ-term **A**, or in **Gamma** or **Delta1**. **(Q T)** is a proof of the upper sequent where **Q** is the part of the proof that does not contain **T**. We omit the details about how the λ-term **Q** is formed. The universal quantifier **pi** is used so that **T** is an arbitrary term which can be treated as a *fresh constant*. Because **Q** does not contain **T**, the proof **(forall_r Q)** also does not contain it. For this case, the clause can be read: if we have a proof function **Q** such that for any **T**, **(Q T)** is a proof of $\Gamma \longrightarrow [T/x]A, \Delta$, then **(forall_r Q)** is the proof for $\Gamma \longrightarrow \forall x A, \Delta$.

The last example is the **initial** rule.

```
(initial A) >- (Gamma --> Delta) :-
  memb A Gamma,
  memb A Delta.
```

The **initial** rule is applied when there is a common formula on both sides of a sequent. In some cases, when the sequent contains *logic variables*, higher-order unification for instantiation of logic variables to make both occurrences of **A** the same is applied here. This operation is built into the implementation of *Teyjus*. The proof search procedure for a sequent is terminated by the **initial** rule if this clause succeeds.

So far, we have already implemented all inference rules in λProlog. In order to develop an automated prover, we need to impose strategies on these rules' applications.

3.1.2 Goal-directed Automated Prover

In the last subsection, we mentioned that once one inference rule is applied, there is one connective or quantifier less in the upper sequent or sequents because we take it out after applying the corresponding rule. This approach guarantees termination of proof search. But in some cases when formulas include $\forall$ on the left side or $\exists$ on the right side of a sequent, our method for implementing a complete proof search procedure must allow these formulas to be used more than once with different values for substitution term t. The strategy that we adopt is to *amplify* the universally quantified formulas on the left and existentially quantified formulas on the right of a sequent by duplicating them more than one time [2].

Using amplification, we present the main idea of our approach to develop an automated prover with a proof tree whose root is labelled by $\Gamma \longrightarrow \Delta$ as follows: at each step, an inference rule is chosen, and then the upper sequent or sequents are computed from the lower sequent according to the rule. Repeat to attempt to apply all possible rules and exit whenever there is a proof. Otherwise, it may be necessary to allow $\forall$-L and $\exists$-R to be applied more than once to the same formula. We restrict search so that all others are applied at most once. Keep on searching until there is a common formula on both sides of a sequent.

In practice, the duplicated quantified formulas are instantiated possibly by different values. This kind of amplification may cause an infinite loop because it keeps adding formulas indefinitely even if the sequent is not provable.

Experiments help us to figure out how to arrange the implementations of these inference rules in an order which is best for the efficiency of the prover for our specific task. The following proof search algorithm is

designed for our automated prover:

Algorithm 1. *Automated Proof Search for FOL Formulas*

Input parameter: **A sequent**

Return: **A Proof** (if there is one, an infinite loop otherwise)

1. Attempt to apply rules in the following order: initial, $\wedge$-L, $\supset$-R, $\exists$-L, $\forall$-R, $\vee$-L, $\wedge$-R, $\supset$-L, $\neg$-R, $\vee$-R and $\neg$-L. Exit if a proof is found. Note that there are no rules $\exists$-R or $\forall$-L in this list.

2. Continue search, attempting all the rules in the following order: initial, $\wedge$-L, $\supset$-R, $\exists$-L, $\forall$-R, $\vee$-L, $\wedge$-R, $\supset$-L, $\neg$-R, $\vee$-R, $\neg$-L, $\exists$-R, $\forall$-L. Note that this list includes rules $\exists$-R and $\forall$-L.

3. Amplification step: If a proof is not found, then using the sequent that is obtained at the end of step 1, add a new copy of all universally quantified formulas on the left and existentially quantified formulas on the right and go to step 2.

In order to completely implement this algorithm, two predicates, **amplify_forall** and **amplify_exists**, are provided to implement amplification for universally and existentially quantified formulas respectively. The main idea of amplification for the rule $\forall$-L as an example is that if there is $\forall xA$ on the left side of a sequent, and if there is not a proof by applying $\forall$-L once, then copy this formula to replace it by $\forall xA \wedge \forall xA$. Attempt to apply all rules again and exit proof search if there is a proof. Otherwise, possibly iterate to replace the formula $\forall xA$ with $\forall xA \wedge \forall xA \wedge ... \wedge \forall xA$ which has one more conjunct than the previous step, until there is a successful termination of proof search. This could lead to an infinite loop.

We have a similar explanation for amplification for ∃-R with the slight difference that $\exists xA$ is replaced with $\exists xA \vee \exists xA... \vee \exists xA \vee \exists xA$ on the right side of a sequent.

Clauses of amplification for rules ∀-L and ∃-R in the module named as **lc_iter.mod** are shown in Figure 3.4. A new predicate **add_copies** is used to add an additional copy of the quantified formula to the appropriate side of a sequent and **N** is used to store how many times amplification is applied.

We explain **amplify_forall** as an example. There are 3 definite clauses for this predicate. The first clause is satisfied if Γ is null which means there is no more universally quantified formulas on the left side of a sequent. The second says, if $\forall xA$ is the first element of Γ, then take it out from Γ and duplicate it in Γ N times . The last clause recursively calls the clause itself using Γ without its first element. This clause is used when the second one fails because the first element in Γ does not have the form $\forall xA$.

Since we need to implement the inference rules in two steps, constants **>-1** and **>-2** are introduced to represent the relations between proofs and the sequents for these two steps respectively. All rules implemented by **>-1** are put into a module, **lc_auto.mod**, in Figure 3.5 and rules implemented by **>-2** are placed into a module, **lc_prover.mod**, in Figure 3.6. Moreover, implementation of amplification in **lc_iter.mod** in Figure 3.4 is a link between **lc_auto.mod** and **lc_prover.mod** because **nprove** defined in **lc_iter.mod** is called in **lc_auto** and **>-2** defined in **lc_prover.mod** is called in **lc_iter**.

The algorithm always terminates with a proof if there is one. If there is none, the amplification step will cause an infinite loop [7].

We can run our prover on the concrete FOL sequent $\exists x\varphi \vee \exists x\psi \longrightarrow$

```
formula pre_1 ((exists x\ (p x)) or (exists x\ (q x))
              imp (exists x\ ((p x) or (q x)))).
formula pre_1 F.


?- Q >- (nil --> F::nil).

Q = imp_r (exists (x3\ p x3) or exists (x3\ q x3))
          (exists (x3\ p x3 or q x3))
          (or_l (exists (x3\ p x3)) (exists (x3\ q x3))
          (exists_l (W1\ exists_r W1
          (or_r (p W1) (q W1) (initial (p W1)))))
          (exists_l (W1\ exists_r W1
          (or_r (p W1) (q W1) (initial (q W1))))))
```

Figure 3.3: An Example of a Proof for a FOL Formula in λProlog

$\exists x(\varphi \vee \psi)$, which is the root of a tree in Figure 2.2 and assigned to **pre_1** in Figure 3.3. We query using **Q >- (nil - -> F::nil)** (The clause begins with a $?-$) where **F** is assigned by matching **formula pre_1 F**. It finds the proof which is represented by **Q** of type **lprf** in Figure 3.3.

In this figure, the formula **pre_1** is the formula to be proved. **Q** is the result we obtain from the prover. **W1** is the *fresh constant* from **exists_l** also used by **exists_r**. Note that the bound variable **x** in the goal formula gets renamed during proof search to **x3**.

This proof search algorithm will be modified to obtain an automated prover for detecting feature interactions in Chapter 5.

3.2 A Proof Checker for FOL Formulas

In this section, based on the same inference rules, we discuss a proof checker.

As we know, the top-level clause $(Q >- (\Gamma \longrightarrow \Delta))$ behaves as a prover if **Q** is available and Γ and Δ are given which means we search

for proof **Q** for the sequent ($\Gamma \longrightarrow \Delta$). Also, we can give a similar query where Q already has a value to a "*proof checker*", which is used to check whether Q is a valid proof for the given sequent ($\Gamma \longrightarrow \Delta$). Furthermore, if the value for Q is coming from the prover that we have developed above, we can use this checker to verify whether our prover runs correctly or not.

Let us explain how to develop a proof checker.

Most implementations of the inference rules for the checker are similar to but simpler than those used for the theorem prover. For the theorem prover, we need strategies to make it automated, but for the checker, we implement the rules directly without any strategy. For example, we do not care about the order of the inference rules and we do not need amplification in the checker.

We use the following example to illustrate how to implement the rule $\wedge$-R in the checker.

```
(and_r A B Q1 Q2) >- (Gamma --> Delta) :-
  memb (A and B) Delta,
  Q1 >- (Gamma --> (A::Delta)),
  Q2 >- (Gamma --> (B::Delta)).
```

A significant distinction between this example and the one we described for the prover is to use the predicate **memb** instead of **memb_and_rest**, which means after applying this rule, the formula ($A \wedge B$) does not need to be removed from Δ. The reason is that in the checker, the *head* of the proof Q which is already assigned determines which rule will be applied. Then the contents of this rule decide whether this rule is appropriately applied or not.

Similarly, we use **memb** in the checker unlike **memb_and_rest** in

the prover for implementation of all other rules. The implementation for the checker is presented in the module **lc_checker.mod** in Figure 3.7.

From now on, we can use the prover to automatically search for proofs. Proof search is successfully terminated if the sequent is provable. Furthermore, we can put this proof with the sequent to the checker to verify the correctness of this proof. Otherwise, there is an infinite loop caused by amplification. For this case, we do not know whether this sequent is true or not.

```
module lc_iter.
accumulate  lc_prover.
type add_copies int -> form -> (list form)
            -> (list form) -> o.
type amplify_forall int -> (list form)
            -> (list form) -> o.
type amplify_exists int -> (list form)
            -> (list form) -> o.
type amplify int -> seq -> seq -> o.
type nprove int -> lprf -> seq -> o.
add_copies 1 A List (A::List).
add_copies N A List (A::List1) :-
  (N > 1), M is (N - 1),
  add_copies M A List List1.
amplify_forall N nil nil.
amplify_forall N ((forall A)::Gamma) Gamma2 :-
  amplify_forall N Gamma Gamma1,
  add_copies N (forall A) Gamma1 Gamma2.
amplify_forall N (A::Gamma) (A::Gamma1) :-
  amplify_forall N Gamma Gamma1.
amplify_exists N nil nil.
amplify_exists N ((exists A)::Delta) Delta2 :-
  amplify_exists N Delta Delta1,
  add_copies N (exists A) Delta1 Delta2.
amplify_exists N (A::Delta) (A::Delta1) :-
  amplify_exists N Delta Delta1.
amplify 1 Seq Seq :- !.
amplify N (Gamma1 --> Delta1) (Gamma2 --> Delta2) :-
  amplify_forall N Gamma1 Gamma2,
  amplify_exists N Delta1 Delta2.
nprove N Q Seq1 :-
  print "\nAttempting to prove the following
  sequent at amplification  ",
  term_to_string N NS, print NS, print "\n",
  term_to_string Seq1 NSeq1, print NSeq1, print "\n",
  amplify N Seq1 Seq2,
  Q >-2 Seq2.
nprove N Q Seq :-
  M is (N + 1),
  nprove M Q Seq.
```

Figure 3.4: Implementation of Amplification for $\forall$-R and $\exists$-L in λProlog

```
module lc_auto.
accumulate  lc_iter.
type    >-1 lprf -> seq -> o.
infixl  >-1 200.
(initial A) >-1 (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(and_l A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1,
  Q >-1 ((A::B::Gamma1) --> Delta).
(imp_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Delta Delta1,
  Q >-1 ((A::Gamma) --> (B::Delta1)).
(exists_l Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (exists A) Gamma Gamma1,
  pi T \ ((Q T) >-1 (((A T)::Gamma1) --> Delta)).
(forall_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  pi T \ ((Q T) >-1 (Gamma --> ((A T)::Delta1))).
(or_l A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Gamma Gamma1,
  Q1 >-1 ((A::Gamma1) --> Delta),
  Q2 >-1 ((B::Gamma1) --> Delta).
(and_r A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >-1 (Gamma --> (A::Delta1)),
  Q2 >-1 (Gamma --> (B::Delta1)).
(imp_l A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Gamma Gamma1,
  Q1 >-1 (Gamma1 --> (A::Delta)),
  Q2 >-1 ((B::Gamma1) --> Delta).
(neg_r A Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Delta Delta1,
  Q >-1 ((A::Gamma) --> Delta1).
(or_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Delta Delta1,
  Q >-1 (Gamma --> (A::B::Delta1)).
(neg_l A Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Gamma Gamma1,
  Q >-1 (Gamma1 --> (A::Delta)).
Q1 >-1 (Gamma --> Delta)  :-
  nprove 1 Q1 (Gamma --> Delta).
```

Figure 3.5: lc_auto.mod in λProlog

```
module lc_prover.
kind seq type.
type --> (list form) -> (list form) -> seq.
infixl --> 210.
type >-2 lprf -> seq -> o.
infixl >-2 200.
(initial A) >-2 (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(imp_l A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Gamma Gamma1,
  Q2 >-2 ((B::Gamma1) --> Delta),
  Q1 >-2 (Gamma1 --> (A::Delta)).
(and_l A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1,
  Q >-2 ((A::B::Gamma1) --> Delta).
(imp_r A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Delta Delta1,
  Q >-2 ((A::Gamma) --> (B::Delta1)).
(exists_l  Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (exists A) Gamma Gamma1,
  pi T \ ((Q T) >-2 (((A T)::Gamma1) --> Delta)).
(forall_r Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  pi T \ ((Q T) >-2 (Gamma --> ((A T)::Delta1))).
(or_l A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A or B) Gamma Gamma1,
  Q1 >-2 ((B::Gamma1) --> Delta),
  Q2 >-2 ((A::Gamma1) --> Delta).
(and_r A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >-2 (Gamma --> (A::Delta1)),
  Q2 >-2 (Gamma --> (B::Delta1)).
(neg_r A Q ) >-2 (Gamma --> Delta) :-
  memb_and_rest (neg A) Delta Delta1,
  Q >-2 ((A::Gamma) --> Delta1).
(or_r A B Q ) >-2 (Gamma --> Delta) :-
  memb_and_rest (A or B) Delta Delta1,
  Q >-2 (Gamma --> (A::B::Delta1)).
(neg_l A Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (neg A) Gamma Gamma1,
  Q >-2 (Gamma1 --> (A::Delta)).
(exists_r T Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (exists A) Delta Delta1,
  Q >-2 (Gamma --> ((A T)::Delta1)).
(forall_l T Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (forall A) Gamma Gamma1,
  Q >-2 (((A T)::Gamma1) --> Delta).
```

Figure 3.6: lc_prover.mod in λProlog

```
module lc_checker.
accumulate  basic,lprf.
kind seq type.
type --> (list form) -> (list form) -> seq.
type >-  lprf -> seq -> o.
infixl --> 210.
infixl >-  200.
(initial A) >- (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(and_r A B Q1 Q2) >- (Gamma --> Delta) :-
  memb (A and B) Delta,
  Q1 >- (Gamma --> (A::Delta)),
  Q2 >- (Gamma --> (B::Delta)).
(or_r A B Q) >- (Gamma --> Delta) :-
  memb (A or B) Delta,
  Q >- (Gamma --> (A::B::Delta)).
(imp_r A B Q) >- (Gamma --> Delta) :-
  memb (A imp B) Delta,
  Q >- ((A::Gamma) --> (B::Delta)).
(neg_r A Q) >- (Gamma --> Delta) :-
  memb (neg A) Delta,
  Q >- ((A::Gamma) --> Delta).
(exists_r T Q) >- (Gamma --> Delta) :-
  memb (exists A) Delta,
  Q >- (Gamma --> ((A T)::Delta)).
(forall_r Q) >- (Gamma --> Delta) :-
   memb (forall A) Delta,
   pi Y\ ((Q Y) >- (Gamma --> ((A Y)::Delta))).
(and_l A B Q) >- (Gamma --> Delta) :-
  memb (A and B) Gamma,
  Q >- ((A::B::Gamma) --> Delta).
(imp_l A B Q1 Q2) >- (Gamma --> Delta) :-
  memb (A imp B) Gamma,
  Q1 >- (Gamma --> (A::Delta)),
  Q2 >- ((B::Gamma) --> Delta).
(forall_l T Q) >- (Gamma --> Delta) :-
  memb (forall A) Gamma,
  Q >- (((A T)::Gamma) --> Delta).
(neg_l A Q) >- (Gamma --> Delta) :-
  memb (neg A) Gamma,
  Q >- (Gamma --> (A::Delta)).
(or_l A B Q1 Q2) >- (Gamma --> Delta) :-
  memb (A or B) Gamma,
  Q1 >- ((A::Gamma) --> Delta),
  Q2 >- ((B::Gamma) --> Delta).
(exists_l Q) >- (Gamma --> Delta) :-
  memb (exists A) Gamma,
  pi Y\ ((Q Y) >- (((A Y)::Gamma) --> Delta)).
```

Figure 3.7: Implementation of A Proof Checker in λProlog

Chapter 4

Feature Interactions

In this chapter, we illustrate how to specify the requirements of telephony features, such as Call Waiting and Call Forwarding, in linear temporal logic. And we also express an LTL formula, which if provable, means that a feature conflict has been detected. This LTL formula representing the conflict expresses logical inconsistency of LTL formulas for two feature requirements. In this way, detecting feature conflicts in the telephony system becomes searching for a proof for this LTL formula in theorem proving.

More precisely, we consider two features as examples to present our specification language: Anonymous Call Rejection (ACR) and Call Forwarding Busy Line (CFBL). First, we explain atomic formulas in the requirements of these two features. And then, according to their informal descriptions and one general form used to represent features in our specification language, we specify two LTL formulas to represent requirements of ACR and CFBL separately. Second, because these two features, as well as many others (additional examples are discussed in Chapter 6), fit a specific specification pattern we intend to focus on, we formalize

their feature conflict as an LTL formula which expresses logical inconsistency of the two LTL feature formulas. Third, based on the semantics of LTL connectives, we translate this LTL formula into a FOL formula whose proof can be decomposed into several cases. These simple cases help us generate a general proof structure used for detecting conflicts between requirements of two features that fit the specification pattern described above. Finally, we explain how we implement these cases in λProlog.

4.1 Specifying Features by LTL Formulas

The features we consider come from Telcordia (Bellcore) standards [1]. These features have many different properties and even for one feature, there may be different properties specifying its different requirements in most cases. For example, in our specification, the feature ACR has 6 different properties and the feature CFBL has 3 different properties. These properties can be informally described according to a sequence of conditions over time. For example, an informal expression of one requirement of ACR states that *if* the entity x is an ACR user who denies calls when the caller, for example, the entity y, does not allow his/her phone number to be displayed on x's device, and *furthermore* if there is a call from y to x, *then eventually* y hears an ACR denial announcement, *unless* y hangs up his/her phone to terminate the call before it is completed. As another example, we present one requirement of CFBL, whose informal description states: an entity x subscribes to CFBL forwarding his/her calls to z when his/her line is busy. If there is a call from y to x, meanwhile, x is not idle and no other calls to x are being forwarded to z, then the call from y is eventually forwarded to z unless y hangs up his/her phone before that happens. Such informal feature

descriptions help us construct formal feature specifications in LTL. This can be successfully done by the following two steps by using ACR and CFBL as examples.

First, from their two informal descriptions, we define the following atomic formulas in terms of entities x, y and z, which are needed in features ACR and CFBL in Figure 4.1:

Atomic formulas	Explanations	ACR	CFBL
$acr(x)$	x is an ACR user	✓	
$dn\_allowed(y)$	y allows to display his/her name	✓	
$call\_req(x, y)$	there is a call from y to x	✓	✓
$acr\_annc(y, x)$	y hears the denial announcement in response to a call to x	✓	
$onhook(y)$	y puts his/her phone back onhook	✓	✓
$cfbl(x)$	x is a CFBL user		✓
$idle(x)$	x's line is idle		✓
$forwarding(x, y, z)$	x forwards y's call to z		✓

Figure 4.1: Atomic Formulas in Features ACR and CFBL

Note that some atomic formulas, such as $call\_req(x, y)$ and $onhook(y)$, in ACR are the same as those in CFBL because they represent common actions in the telephony system.

Second, we use boolean and temporal connectives to represent logical relations, particularly the time-dependent properties, such as *furthermore, eventually* and *unless* in the feature requirements.

The general form for feature specifications in Figure 4.2 has been successfully employed in model checking [8, 9]. This form is divided into two parts, separated by a dash line. The "precondition" is above the line and the "postcondition" is under the line. The postcondition should hold after the precondition becomes true.

The precondition consists of series of events (e_0, e_1, ..., e_n) and the persisting conditions (p_0, p_1, ..., p_{n-1}). These events and persisting conditions consequently hold one by one, such as e_0, p_0, e_1, p_1,..., e_n,

property <Name>
{
  event: e_0 persists: p_0
  event: e_1 persists: p_1
  ...
  event e_n
- - - - - - - - - - - - - - - - - - -
  persists: p until: r discharge: d
}

Figure 4.2: A General Form for the Features Specification

which means that initially, e_0 holds with a period of $(p_0 \wedge \neg e_1)$ until e_1 is true, then e_1 holds for a while with $(p_1 \wedge \neg e_2)$ holding until e_2; ..., etc, until e_n holds.

The postcondition contains one persisting condition (p), one resolution condition (r) and one discharge condition (d). The postcondition means that after the persisting condition p stops holding, either the resolution r or the discharge d holds. This can be represented by an LTL formula $(p \mathsf{U} (r \vee d))$ where the resolution r is an expected response to the persisting condition p and the discharge d is an exception to it. For some features, keyword *unless* is used to represent a *weaker* condition for the resolution r instead of *until*, and then the formula of the postcondition is changed to $(p \mathsf{W} (r \vee d))$ which expresses the fact that p could hold forever, but if it does not, then it must be resolved by either r or d.

All conditions described above in the general form are FOL formulas. Note that we could distinguish between atomic formulas that hold over a period of time, such as $acr(x)$ holding for the entire time, and external events or triggers, such as $call\_req(x, y)$ which becomes true as soon as a call is initiated. In our case, $call\_req(x, y)$ is also viewed as occurring over a period of time starting from when a call is initiated until the next phase of the call starts. Although an intuitive understanding of the difference between events and conditions which persist over time is useful, we do not

property <Name>
{
  event: e
- -
  persists: p until: r discharge: d
}

Figure 4.3: A Simple Specification Form for one Kind of Features

need to distinguish these two kinds of atomic formulas when expressing requirements in temporal logic. For example, $call\_req(x, y)$ above the line can be viewed as an event (at the point in time it starts to hold) and below the line can be viewed as a persisting condition over time.

Although this form was originally designed for model checking [8, 9], for the case we consider, this form can also be used in theorem proving since we use the same logic.

Specifying the general form in Figure 4.2 in LTL is challenge. For our implementation and experiments in theorem proving, we focus on a special case of the general form with just one event condition (e) in the precondition shown in Figure 4.3.

In linear temporal logic, this simple specification form, which represents a special feature specification pattern, can be represented by the LTL formula below [7]:

$$\Box[e \supset (p \,\mathsf{U}\, (r \vee d))]. \tag{4.1}$$

This formula can be understood that it is always true that event e implies that condition p persists until either resolution r or discharge d becomes true.

Many examples of features, including ACR and CFBL described at the beginning of this section, are fitting into this pattern, on which our current case study focuses.

Based on the atomic formulas defined in Figure 4.1, the ACR requirement discussed above can be represented using the special general form below [8, 9]:

property ACR_Normal_Operation_3

{

event: $acr(x) \wedge \neg dn\_allowed(y) \wedge call\_req(x, y)$

- -

persists: $call\_req(x, y)$

until: $acr\_annc(y, x)$

discharge: $onhook(y)$

}

The precondition of this ACR requirement is $(acr(x) \wedge \neg dn\_allowed(y) \wedge call\_req(x, y))$ which says that x is an ACR subscriber, y is an entity preventing his/her name from being displayed on x's device when there is a call from y to x. The postcondition here has a form $(call\_req(x, y)\ \mathsf{U}(acr\_annc(y, x) \vee onhook(y)))$, which tells us that if y calls x, then this call is resolved by either y hearing a denial announcement or y putting his/her phone back onhook.

By illustrating the precondition and postcondition in this example, we have explained how the simple general form in Figure 4.3 can be used to formally represent a requirement of a feature.

Similarly, we can also specify the CFBL requirement, our other example, in this special pattern as follows:

property CFBL_Normal_Operation_1

{

event: $cfbl(x) \wedge \neg idle(x) \wedge \neg \exists y\ forwarding(x, y, z) \wedge call\_req(x, y)$

- -

persists: $call\_req(x, y)$

until: $forwarding(x, y, z)$

discharge: $onhook(y)$

}.

This form is also a formalization of the informal requirement of the CFBL feature.

Since both of these two examples fit the pattern in Figure 4.3 and it can be expressed by an LTL formula (4.1), we use the following two LTL formulas to abbreviate them. We use subscript 1 for ACR and 2 for CFBL respectively:

$$\Box[e_1 \supset (p_1 \mathsf{U} (r_1 \vee d_1))], \tag{4.2}$$

$$\Box[e_2 \supset (p_2 \mathsf{U} (r_2 \vee d_2))]. \tag{4.3}$$

where e_1, p_1, r_1, d_1, e_2, p_2, r_2 and d_2 are specialized below:

$e_1 := acr(x) \wedge \neg dn\_allowed(y) \wedge call\_req(x, y)$;

$p_1 := call\_req(x, y)$;

$r_1 := acr\_annc(y, x)$;

$d_1 := onhook(y)$;

$e_2 := cfbl(x) \wedge \neg idle(x) \wedge \neg \exists y\ forwarding(x, y, z) \wedge call\_req(x, y)$;

$p_2 := call\_req(x, y)$;

$r_2 := forwarding(x, y, z)$;

$d_2 := onhook(y)$.

By using ACR and CFBL as examples, we have shown how we specify features in the special pattern (4.1) by LTL formulas over a set of atomic formulas.

4.2 Specifications for Detecting Feature Conflicts by LTL Formulas

Feature conflict considered in our approach means that it is not possible to build a system which satisfies both feature specifications. This corresponds to logical inconsistency of two feature specification formulas. In other words, a conflict between two features means their requirements cannot hold simultaneously on any computation path in a system. This can be represented by an LTL formula:

$$\neg(\Box A \wedge \Box B) \tag{4.4}$$

where we use $\Box A$ and $\Box B$ to present the two feature specifications respectively.

Basically, if this formula is true, then there is a conflict between features A and B. But in practice, it is insufficient to detect the conflicts directly by our prover. Instead, we consider the following LTL formula by adding "SA" and "Constraint":

$$\neg(\Box A \wedge \Box B \wedge \Box[SA] \wedge \Diamond[Constraint]). \tag{4.5}$$

Considering the conflict between ACR and CFBL described above, these four conjuncts have the following meanings: the first two are the two feature specifications for the ACR and CFBL requirements, represented as described earlier:

$A := [e_1 \supset (p_1 \,\mathsf{U}\, (r_1 \vee d_1))]$;

$B := [e_2 \supset (p_2 \,\mathsf{U}\, (r_2 \vee d_2))]$.

The third one is SA, "System Axioms", which has the following particular value for this example [7]:

$$SA := \forall x \forall y \forall z (\neg((acr\_annc(y,x) \wedge forwarding(x,y,z))$$
$$\wedge \neg(acr\_annc(y,x) \wedge call\_req(x,y))$$
$$\wedge \neg(forwarding(x,y,z) \wedge call\_req(x,y))$$
$$\wedge \neg(onhook(y) \wedge call\_req(x,y))$$
$$\wedge \neg(onhook(y) \wedge forwarding(x,y,z)))).$$

Inside which, pairs of atomic formulas, such as $dn\_allowed(y,x)$ and $forwarding(x,y,z)$, never hold simultaneously in any reasonable implementation of a telephony system because, using this pair of atomic formulas as an example, a call cannot be resolved both by playing ACR announcement and being forwarded in the ACR and CFBL examples.

Actually, SA is a long formula with many pairs of atomic formulas that cannot hold simultaneous as well as many other constraints. We only present some pairs to simplify the example. When new features are added, more formulas are inserted which we will illustrate in chapter 6.

The last one is the most complicated and specified by [7]:

$$Constraint := \Diamond[e_1 \wedge e_2 \wedge g]$$

where "Constraint" imposes our attention on the particular paths where the conflict occurs only when both preconditions of features can hold simultaneously. This is what the first 2 conjuncts mean. In addition, the last conjunct can be specified by [7]:

$$g := (p_1 \wedge p_2 \,\mathsf{U}\, (\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2)).$$

This formula g can be understood as: it eliminates the computation paths where all calls are discharged before they are even completed.

φ below by instantiating variables with the exact values from ACR and CFBL, is an example to illustrate how to specify an LTL formula to represent conflict between two features fitting the same special specification pattern (4.1).

$$\varphi := \neg(\Box[e_1 \supset (p_1 \mathsf{U} (r_1 \vee d_1))] \wedge \Box[e_2 \supset (p_2 \mathsf{U} (r_2 \vee d_2))] \\ \wedge \Box[SA] \wedge \Diamond[e_1 \wedge e_2 \wedge g]) \tag{4.6}$$

This formula is the one which we intend to prove by using our prover.

4.3 Transforming Specifications From LTL to FOL

Among the approaches to proving LTL formulas, we choose to translate them into FOL formulas. According to the translation functions in subsection 2.2.2, φ can be translated to its corresponding FOL formula by computing $f(\varphi, 0)$.

$$\begin{aligned} f(\varphi, 0) := \neg(f((&\Box[e_1 \supset (p_1 \mathsf{U}(r_1 \vee d_1))] \\ &\wedge \Box[e_2 \supset (p_2 \mathsf{U}(r_2 \vee d_2))] \wedge \Box[SA] \wedge \Diamond[e_1 \wedge e_2 \wedge g]), 0)) \\ := \neg(&f(\Box[e_1 \supset (p_1 \mathsf{U}(r_1 \vee d_1))], 0) \\ &\wedge f(\Box[e_2 \supset (p_2 \mathsf{U}(r_2 \vee d_2))], 0) \\ &\wedge f(\Box[SA], 0) \\ &\wedge f(\Diamond[e_1 \wedge e_2 \wedge g]), 0)) \end{aligned}$$

We discuss the four conjuncts inside the negation of the above formula separately.

$f(\Box[e_1 \supset (p_1 \mathsf{U}(r_1 \vee d_1))], 0)$,

$f(\Box[e_2 \supset (p_2 \mathsf{U}(r_2 \vee d_2))], 0)$,

$f(\Box[SA], 0)$,

$f(\Diamond[e_1 \wedge e_2 \wedge g]), 0)$

They are translated to the following FOL formulas respectively.

$$\forall k(k \geqslant 0 \supset (f(e_1,k) \supset \exists i_1(i_1 \geqslant k \wedge f((r_1 \vee d_1), i_1) \wedge \forall j^{'}(k \leqslant j^{'} < i_1 \supset f(p_1, j^{'}))))), \tag{4.7}$$

$$\forall k(k \geqslant 0 \supset (f(e_2,k) \supset \exists i_2(i_2 \geqslant k \wedge f((r_2 \vee d_2), i_2) \wedge \forall j^{'}(k \leqslant j^{'} < i_2 \supset f(p_2, j^{'}))))), \tag{4.8}$$

$$\begin{aligned}\forall k(k \geqslant 0 \supset (f(\neg((&acr\_annc(y,x,k) \wedge forwarding(x,y,z,k)) \\ &\wedge \neg(acr\_annc(y,x,k) \wedge call\_req(x,y,k)) \\ &\wedge \neg(forwarding(x,y,z,k) \wedge call\_req(x,y,k)) \\ &\wedge \neg(onhook(y,k) \wedge call\_req(x,y,k)) \\ &\wedge \neg(onhook(y,k) \wedge forwarding(x,y,z,k))))),\end{aligned} \tag{4.9}$$

$$\exists k(k \geqslant 0 \supset (f(e_1,k) \wedge f(e_2,k) \wedge f(g,k))). \tag{4.10}$$

Formula (4.10) is further represented by the one below:

$$\begin{aligned}\exists k(k \geq 0 \supset (&f(e_1,k) \wedge f(e_2,k) \wedge \\ &\exists j(j \geq k \ \wedge f(\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2, j) \wedge \\ &\forall j^{'}(k \leq j^{'} < j \supset f(p_1 \wedge p_2, j^{'}))))).\end{aligned} \tag{4.11}$$

By employing the translation functions, we have obtained the translated FOL formulas which have not only FOL formulas with integers, such as $acr\_annc(y,x,k)$ and $forwarding(x,y,z,k)$, but also relational expressions on integers. For example, there are $k \geqslant 0$ and $j \geqslant k$ in the above formula. These integers, such as k and j, represent the important states along the computation paths where the feature conflicts may occur.

4.4 Decomposing the Proof into Simpler Cases

Proving the truth of the translated FOL formula means finding a contradiction among the four conjuncts. In order to obtain the general proof structure, we instantiate the existential quantifiers with constants. For example, in this case study, we instantiate $\exists i_1$, $\exists i_2$, $\exists j$ and $\exists k$ with constants i_1, i_2, j and k. This is an important step in the proof structure for this particular application.

Following an analysis of these four conjuncts, we illustrate the important states represented by constants i_1, i_2, j and k on a given path by tables below (0 is an integer indicating the start state in all tables):

- From Formula (4.10), we get an integer k. There is a $k \geqslant 0$, such that $(e_1 \wedge e_2 \wedge g)$ holds at s_k, shown in Table 4.1:

0	k
$\downarrow$	$\downarrow$
	$\uparrow$
	$(e_1 \wedge e_2 \wedge g)$

Table 4.1: Path with an Integer k

- Since e_1 holds at the state s_k, from Formula (4.7), we have an integer i_1 such that $(i_1 \geqslant k)$ and $(r_1 \vee d_1)$ holds at s_{i_1} and p_1 holds between s_k and s_{i_1}, shown in Table 4.2:

- Also, since e_2 holds at the state s_k, from Formula (4.8), we have an integer i_2 in Table 4.3 where there is an $(i_2 \geqslant k)$, $(r_2 \vee d_2)$ holds at s_{i_2} and p_2 holds between s_k and s_{i_2}:

0	k				i_1
f ↓	↓				↓
	↑	↑	⋯	↑	↑
	$(e_1 \wedge e_2 \wedge g)$	p_1	⋯	p_1	$(r_1 \vee d_1)$

Table 4.2: Path with an Integer i_1

0	k				i_2
↓	↓				↓
	↑	↑	⋯	↑	↑
	$(e_1 \wedge e_2 \wedge g)$	p_2	⋯	p_2	$(r_2 \vee d_2)$

Table 4.3: Path with an Integer i_2

- From Formula (4.10), g holds at the state s_k, and furthermore, from Formula (4.11), there is a $(j \geqslant k)$ and $(\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2)$ holds at s_j and $(p_1 \wedge p_2)$ holds between s_k and s_j, shown in Table 4.4:

0	k				j
↓	↓				↓
	↑	↑	⋯	↑	↑
	$(e_1 \wedge e_2 \wedge g)$	$(p_1 \wedge p_2)$	⋯	$(p_1 \wedge p_2)$	$(\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2)$

Table 4.4: Path with an Integer j

So far, we have obtained integers i_1, i_2, j and k and know that they are all $\geqslant 0$ and i_1, i_2 and j are $\geqslant k$. Among integers i_1, i_2 and j, we must consider the different relations and show all cases which lead to contradiction. For example, we know that Table 4.2 and Table 4.4 are satisfied at the same time. If we also assume $(i_1 > j)$, we get Table 4.5.

From this table, we have the obvious contradiction at state s_j where p_1 holds meanwhile $\neg p_1$ also holds.

Or if we assume $(i_1 < j)$, then we get Table 4.6.

Table 4.6 says the formula $(p_1 \wedge p_2)$ holds at the state s_{i_1} where $(r_1 \vee d_1)$ holds as well. Then $call\_req(x, y) \wedge (acr\_annc(y, x) \vee onhook(y))$ holds, which means either $(call\_req(x, y) \wedge acr\_annc(y, x))$ or $(call\_req(x, y) \wedge$

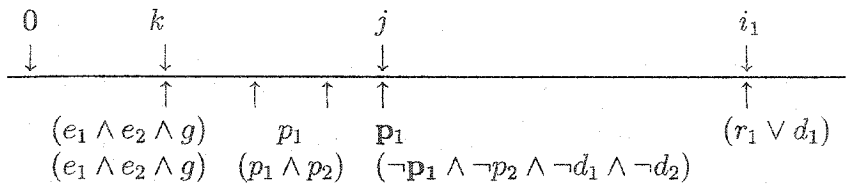

Table 4.5: Contradiction for Case 1

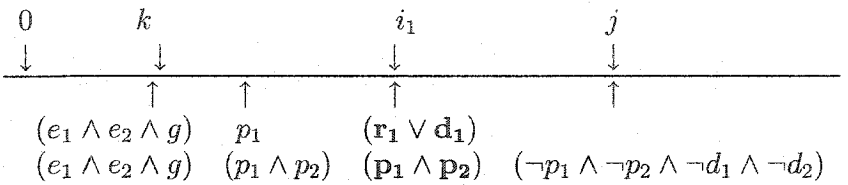

Table 4.6: Contradiction for Case 2

onhook(y)) holds. Formula (4.9) tells us SA holds everywhere and states *call_req*(x, y) and *acr_annc*(y, x) never hold and also *call_req*(x, y) and *onhook*(y) never hold at the same time. This is a contradiction between these two statements.

Consequently, we reach the similar contradictions when $(i_2 > j)$ and $(i_2 < j)$ are assumed.

The last case is when $(i_1 = i_2 = j)$ is true, shown in Table 4.7:

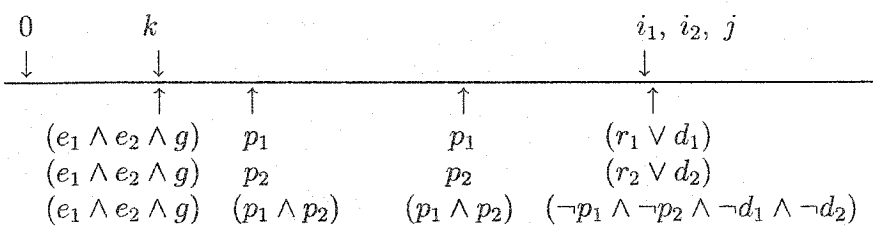

Table 4.7: Contradiction for Case 5

From Table 4.2, we know that at the state s_{i_1}, $(r_1 \vee d_1)$ holds as well as $(r_2 \vee d_2)$ from Table 4.3. We also get the fact from Table 4.4 that at the state s_{i_1}, $(\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2)$ holds. By logical reasoning, we conclude $(r_1 \wedge r_2)$ holds here, which contradicts SA states that *acr_annc*(y, x) and $forwarding(x, y, z)$ cannot hold simultaneously.

Since there is a contradiction in all 5 cases, this completes the proof. We describe how our prover finds the proof in Chapter 6.

In fact, there are 13 different possible cases among 3 integers i_1, i_2 and j: there are 6 cases where all 3 integers are different; there are 6 other cases in which 2 of these integers are same and the other is different; the last case is when all 3 integers are the same. But experiments demonstrate that only 5 cases we discussed above by tables are enough to cover all the other cases. For example, Table 4.5, which assumes $(i_1 > j)$, covers cases: $(i_2 < j < i_1)$, $(i_1 = i_2 > j)$, $(i_1 > i_2 > j)$, $(i_1 > j > i_2)$ and $(i_1 > j = i_2)$.

We conclude by summarizing all these 5 cases in Table 4.8 to illustrate the general proof structure used in searching for contradictions in this case study whose two feature specifications fit the special pattern (4.1).

Case	Conclusion from analysis	Contradiction	SA
$i_1 > j$	$p_1 \wedge \neg p_1$	✓	
$i_2 > j$	$p_2 \wedge \neg p_2$	✓	
$i_1 < j$	$p_1 \wedge (r_1 \vee d_1)$	✓	If $\neg(p_1 \wedge r_1)$ and $\neg(p_1 \wedge d_1))$ are in SA
$i_2 < j$	$p_2 \wedge (r_2 \vee d_2)$	✓	If $\neg(p_2 \wedge r_2)$ and $\neg(p_2 \wedge d_2))$ are in SA
$i_1 = i_2 = j$	$r_1 \wedge r_2$	✓	If $\neg(r_1 \wedge r_2)$ is in SA

Table 4.8: General Proof Structure of Detecting Conflicts in this Case Study

Actually, the first 2 cases are even general enough to establish contradictions between any two feature specifications fitting the specification pattern (4.1). If the conditions written in the column SA in the table 4.8 are true for two other feature specifications, we can also draw the contradictions in a similar way for the last 3 cases. Otherwise, we cannot. For example, in the last case, if we have $(r_1 = r_2)$, then the condition $(\neg(r_1 \wedge r_2))$ is equivalent to $(\neg r_1)$ or $(\neg r_2)$, which are not in SA. So we

do not have contradictions as we show in this table. Proof search will be important in such cases to find a contradiction if there is one.

In order to represent these 5 cases, we use F_1, F_2, F_3 and F_4 to represent formulas (4.7), (4.8), (4.9) and (4.11) respectively and use $case_1$, $case_2$, $case_3$, $case_4$ and $case_5$ to represent each case. These are presented in Figure 4.4.

$case_1 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_1 > j))$
$case_2 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_1 < j))$
$case_3 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_2 > j))$
$case_4 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_2 < j))$
$case_5 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_1 = i_2 = j))$

Figure 4.4: Formulas for the Decomposed Cases

From the tables illustrating how we draw the contradictions, we notice that not all these four formulas are always involved in finding the contradictions. We further simplify each case and include only the important facts which affect the conflicts in Figure 4.5.

$case_1 := \neg(F_1 \wedge F_4 \wedge (i_1 > j))$
$case_2 := \neg(F_1 \wedge F_3 \wedge F_4 \wedge (i_1 < j))$
$case_3 := \neg(F_2 \wedge F_4 \wedge (i_2 > j))$
$case_4 := \neg(F_2 \wedge F_3 \wedge F_4 \wedge (i_2 < j))$
$case_5 := \neg(F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge (i_1 = i_2 = j))$

Figure 4.5: Formulas for the Simpler Decomposed Cases

4.5 Implementations of Cases in λProlog

In this section, we illustrate how to specify the five cases in λProlog. Basic predicates for each case are defined in Figure 4.6.

In this figure, **tm** is a type to represent terms x, y and z. **nat** is a type to represent integers indicating the different states on a computation path. The predicate **acr**, as an example, takes two parameters, an ACR user of type **tm**, and the state integer of type **nat**. Then **(acr x k)**, which

```
kind nat type.
kind tm type.
type acr tm -> nat -> form.
type acr_annc tm -> tm -> nat -> form.
type call_req tm -> tm -> nat -> form.
type dn_allowed tm -> nat -> form.
type onhook tm -> nat -> form.
type cfbl tm -> nat -> form.
type idle tm -> nat -> form.
type forwarding tm -> tm -> tm -> nat -> form.
```

Figure 4.6: Declarations for Atomic Formulas in the Telephony System in λProlog

represents the FOL formula $acr(x, k)$, is an atomic formula in λProlog. We have similar explanations for other predicates.

Based on these declarations of atomic formulas, we define predicates **alwaysSpec1** for Formula (4.7), **alwaysSpec2** for Formula (4.8), **alwaysSA** for Formula (4.9) and **eventuallyE1E2** for Formula (4.11) respectively. A new predicate **fact** is designed to represent each of those four formulas. From the tables shown to illustrate contradictions, we know these formulas are true at some special states. For example, there is an integer k and $k \geqslant 0$ by Formula (4.10). We instantiate $\forall k$ in Formulas (4.7), (4.8) and (4.9) with this k. In Formulas (4.7), (4.8) and (4.11), there are formulas including $\exists i_1, \exists i_2$ and $\exists j$, which we name and treat as constants i_1, i_2 and j. We use λ-binding $\mathbf{I}\backslash$ or $\mathbf{J}\backslash$, where variables $\mathbf{I}$ and $\mathbf{J}$ serve as parameters which can be flexibly assigned to different actual values. For example, $\mathbf{I}$ is instantiated sometimes by i_1 and sometimes by i_2. Then we represent each formula in Figure 4.7 where all constants **alwaysSpec1, alwaysSpec2, eventuallyE1E2** and **alwaysSA** have one parameter that always has one bound variable of type **nat**; **x1**, **y1** and **z1** are three arbitrary telephony users who are involved in the features ACR and CFBL.

```
type  x1 ,y1, z1 tm.
type  k  nat.
type  gt  nat -> nat -> form.
type  lt  nat -> nat -> form.
type  ge  nat -> nat -> form.
type  le  nat -> nat -> form.
type  eq  nat -> nat -> form.
type  fact  form -> o.
type  alwaysSpec1 (nat -> form) -> form.
type  alwaysSpec2 (nat -> form) -> form.
type  eventuallyE1E2 (nat -> form) -> form.
type  alwaysSA (nat -> form) -> form.

fact (alwaysSpec1 I\ ((ge k zero) imp
   ((acr x1 k) and (call_req x1 y1 k) and
   (neg (dn_allowed y1 k))) imp
     ((ge I k ) and
               ((acr_annc y1 x1 I) or (onhook y1 I)) and
               (forall I0\ (le k I0) imp (lt I0 I)
                        imp (call_req x1 y1 I0))))).
fact   (alwaysSpec2 I\ ((ge k zero) imp
    ((cfbl x1 k) and (neg (idle x1 k)) and
    (neg (exists Y\ (forwarding x1 Y z1 k))) and
       (call_req x1 y1 k)) imp
           ((ge I k) and
               ((forwarding x1 y1 z1 I) or (onhook y1 I)) and
               (forall I0\ (le k I0) imp (lt I0 I) imp
                           (call_req x1 y1 I0))))).
fact  (alwaysSA I\ ((ge I zero) imp
 ((neg ((acr_annc y1 x1 I ) and (forwarding x1 y1 z1 I))) and
  (neg ((acr_annc y1 x1 I) and (call_req x1 y1 I))) and
  (neg ((forwarding x1 y1 z1 I) and (call_req x1 y1 I))) and
  (neg (onhook y1 I) and (call_req x1 y1 I)) and
  (neg (onhook y1 I) and (forwarding x1 y1 z1 I))))).
fact  (eventuallyE1E2 J\ ((ge k zero) and
  ((acr x1 k ) and (call_req x1 y1 k ) and
  (neg (dn_allowed y1 k ))) and
              ((cfbl x1 k ) and (neg (idle x1 k )) and
               (neg (exists Y\ (forwarding x1 Y z1 k ))) and
                    (call_req x1 y1 k )) and
        ((ge J k ) and (neg (call_req x1 y1 J)) and
                   (neg (onhook y1 J)) and
  (forall J0\ (le k J0) imp (lt J0 J) imp (call_req x1 y1 J0))))).
```

Figure 4.7: Declarations for Specifications of ACR and CFBL in λProlog

Since we decompose the complicated FOL formula into 5 simpler cases as we assume $(i_1 > j)$, $(i_1 < j)$, $(i_2 > j)$, $(i_2 < j)$ and $(i_1 = i_2 = j)$, we assign **case1**, **case2**, **case3**, **case4** and **case5** to describe each of them in Figure 4.8 where formulas correspond to those we obtain in Figure 4.5.

```
type i1,i2,j  nat.
type case1 name.
type case2 name.
type case3 name.
type case4 name.
type case5 name.
formula case1 F :-
 fact (alwaysSpec1 F1),
 fact (eventuallyE1E2 F4),
 F = (neg ((F1 i1) and (F4 j) and (gt i1 j))).
formula case2 F :-
 fact (alwaysSpec1 F1),
 fact (alwaysSA F3),
 fact (eventuallyE1E2 F4),
 F = (neg ((F1 i1) and (F3 i1) and (F4 j)  and (lt i1 j))).
formula case3 F :-
 fact (alwaysSpec2 F1),
 fact (eventuallyE1E2 F2),
 F = (neg ((F1 i2) and (F2 j) and (gt i2 j))).
formula case4 F :-
 fact (alwaysSpec2 F1),
 fact (eventuallyE1E2 F2),
 fact (alwaysSA F3),
 F = (neg ((F1 i2) and (F2 j) and (F3 i2) and (lt i2 j))).
formula case5 F :-
  fact (alwaysSpec1 F1),
  fact (alwaysSpec2 F2),
  fact (eventuallyE1E2 F3),
  fact (alwaysSA F4),
  F = (neg ((F1 i1) and (F2 i1)  and (F3 i1)  and (F4 i1))).
```

Figure 4.8: Implementation of Simpler Cases in λProlog

Subsequently, we ask the prover to search for proofs for these cases. Since there are relational expressions which cannot be handled by the existing prover, we provide strategies to handle them in the next chapter.

Chapter 5

Algorithms for Relational Expressions

Translating an LTL formula (4.6) into FOL gives a translated formula consisting of not only FOL formulas with integers, such as $(acr\_annc(y, x, k))$, but also relational expressions, such as $(i_1 \geqslant k)$. The existing prover described in chapter 3 cannot completely search for proof for these formulas because there are no rules for the prover to handle the relational expressions. For example, the prover fails when searching for a proof for a sequent $(x_1 > x_2) \longrightarrow (x_2 < x_1)$ because it cannot find the same formula on both sides of this sequent. The reason for this failure is that the prover does not know the equivalence between $(x_1 > x_2)$ and $(x_2 < x_1)$. New rules are needed to enhance the prover's capability.

In this chapter, we present our algorithms to handle relational expressions. First, we reduce 5 relational operators to 2 operators. Second, we show mathematical properties about these two operators and their expressions and express the properties as additional rules in sequent format. Third, we present our algorithms to implement these rules, and

then illustrate examples to explain them. Finally, we integrate the algorithms into our prover to build a powerful automated prover to handle these special FOL formulas.

5.1 Reducing Relational Operators Using $=$ and $<$

Although there are five relational operators on equalities and inequalities, $>, \geqslant, =, <$ and $\leqslant$ between two integers, two of them, $<$ and $=$, are enough to represent all the others. Table 5.1 illustrates how to do so.

expression	represented by	$<$	and	$=$
$A > B$	$\Leftrightarrow$	$B < A$		
$A \geqslant B$	$\Leftrightarrow$	$B < A$	$\vee$	$A = B$
$A = B$	$\Leftrightarrow$			$A = B$
$A < B$	$\Leftrightarrow$	$A < B$		
$A \leqslant B$	$\Leftrightarrow$	$A < B$	$\vee$	$A = B$

Table 5.1: Reducing Relational Operators using $=$ and $<$

After reducing relational operators, we can focus only on $=$ and $<$ once $>$, $\geqslant$ and $\leqslant$ are transformed to either $<$ or compound expressions of $=$ and $<$. We declare these transformations as additional rules for the sequent proof system in Figure 5.1.

$$\frac{\Gamma \longrightarrow B < A, \Delta}{\Gamma \longrightarrow A > B, \Delta} >\text{-R} \qquad \frac{B < A, \Gamma \longrightarrow \Delta}{A > B, \Gamma \longrightarrow \Delta} >\text{-L}$$

$$\frac{\Gamma \longrightarrow B < A, A = B, \Delta}{\Gamma \longrightarrow A \geqslant B, \Delta} \geqslant\text{-R} \qquad \frac{(B < A) \vee (A = B), \Gamma \longrightarrow \Delta}{A \geqslant B, \Gamma \longrightarrow \Delta} \geqslant\text{-L}$$

$$\frac{\Gamma \longrightarrow A < B, A = B, \Delta}{\Gamma \longrightarrow A \leqslant B, \Delta} \leqslant\text{-R} \qquad \frac{(A < B) \vee (A = B), \Gamma \longrightarrow \Delta}{A \leqslant B, \Gamma \longrightarrow \Delta} \leqslant\text{-L}$$

Figure 5.1: Additional Rules about Relational Operators

For operators $>$, $\geqslant$ and $\leqslant$, a pair of rules are defined in terms of $=$ and $<$ since each of them may appear on either side of a sequent.

New constants are introduced in λProlog for each additional rule to keep track of the transformations in Figure 5.2.

```
type gt_r  nat -> nat -> lprf -> lprf.
type gt_l  nat -> nat -> lprf -> lprf.
type ge_r  nat -> nat -> lprf -> lprf.
type ge_l  nat -> nat -> lprf -> lprf.
type le_r  nat -> nat -> lprf -> lprf.
type le_l  nat -> nat -> lprf -> lprf.
```

Figure 5.2: Declarations for Additional Rules of Transformations

Taking constant **gt_r** as an example, it is used to construct a proof with three input parameters, two of type **nat** and one of type **lprf**. **(gt_r A B Q)** of type **lprf** is a proof of employing the rule $>$-R. Inside this proof, **A** and **B** are two integers of type **nat** which come from $(A > B)$ in the lower sequent and **Q** of type **lprf** is a proof of the upper sequent where a formula $(A > B)$ on the right side of the lower sequent is transformed to the formula $(B < A)$ on the same side. Other constants are defined similarly.

We have the definite clauses for each rule in Figure 5.3.

As we have explained for the implementations of the inference rules, we have very similar explanations for these additional rules. For example, the rule $>$-R says that if **Q** is a proof of $(\Gamma \longrightarrow B < A, \Delta)$, then **(gt_r A B Q)** is a proof of $(\Gamma \longrightarrow A > B, \Delta)$.

After attempting all these rules, expressions on $>, \geqslant$ and $\leqslant$ are transformed to either expressions on $<$ solely or compound expressions on $=$ and $<$.

```
(gt_r A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (gt A B) Delta Delta1,
    Q >- (Gamma --> ((lt B A)::Delta1)).
(gt_l A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (gt A B) Gamma Gamma1,
    Q >- (((lt B A)::Gamma1) --> Delta).
(ge_r A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (ge A B) Delta Delta1,
    Q >- (Gamma --> ((lt B A) ::( eq A B):: Delta1)).
(ge_l A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (ge A B) Gamma Gamma1,
    Q >- ((((lt B A) or (eq A B))::Gamma1) --> Delta).
(le_r A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (le A B) Delta Delta1,
    Q >- (Gamma --> ((lt A B)::(eq A B)::Delta1)).
(le_l A B Q) >- (Gamma --> Delta) :-
    memb_and_rest (le A B) Gamma Gamma1,
    Q >- ((((lt A B) or (eq A B))::Gamma1) --> Delta).
```

Figure 5.3: Implementation of Additional Rules of Transformations

5.2 Properties of Relational Operators of $=$ and $<$

By reducing five relational operators to only $=$ and $<$, we have simplified the relational expressions. In this section we concentrate on the discussions of these two operators.

Now, our prover still has difficulties to complete proof search in some cases. Here is an example. The prover cannot search for a proof of a sequent $(x_1 = x_2) \longrightarrow (x_2 = x_1)$ because there is not an exact formula on both sides of this sequent although these two formulas are equivalent. More mathematical properties of $=$ and $<$ are needed to fulfil the proof search procedure. For this example, this sequent is provable by adding $(x_2 = x_1)$ to the left side of this sequent after applying symmetry of equality.

As we know, expressions on $=$ and $<$ have the following properties:

Axiom. *Reflexivity of Equality*

$$\forall x\ (x = x) \tag{5.1}$$

Axiom. *Symmetry of Equality*

$$\forall x\ \forall y\ ((x = y) \supset (y = x)) \tag{5.2}$$

Axiom. *Transitivity of Equality*

$$\forall x\ \forall y\ \forall z\ ((x = y \wedge y = z)\ \supset (x = z)) \tag{5.3}$$

Axiom. *Transitivity of "Equality-Less-than"*

$$\forall x\ \forall y\ \forall z\ ((x = y \wedge y < z)\ \supset (x < z)) \tag{5.4}$$

Axiom. *Transitivity of "Less- than-Equality"*

$$\forall x\ \forall y\ \forall z\ ((x < y \wedge y = z)\ \supset (x < z)) \tag{5.5}$$

Axiom. *Transitivity of "Less-than(<)"*

$$\forall x\ \forall y\ \forall z\ ((x < y \wedge y < z) \supset (x < z)) \tag{5.6}$$

By applying these axioms on formulas which are on the left side of a sequent, we obtain more consequences which are also regarded as premises. We write down the following new rules, called auxiliary rules, to represent the axioms in the sequent format in Figure 5.4.

Our prover is enhanced by adding these rules so that if there is a proof, the prover finds it.

5.3 Algorithms and Implementations of Properties

We present our algorithms for these rules and implement them straightforwardly in λProlog in this section.

$$\frac{A = A, B = B, \Gamma \longrightarrow \Delta}{A = B, \Gamma \longrightarrow \Delta} \text{ =-Ref} \qquad \frac{A = B, B = A, \Gamma \longrightarrow \Delta}{A = B, \Gamma \longrightarrow \Delta} \text{ =-Sym}$$

$$\frac{A = B, B = C, A = C, \Gamma \longrightarrow \Delta}{A = B, B = C, \Gamma \longrightarrow \Delta} \text{ =-Tran}$$

$$\frac{A < B, B < C, A < C, \Gamma \longrightarrow \Delta}{A < B, B < C, \Gamma \longrightarrow \Delta} \text{ <-Tran}$$

$$\frac{A = B, B < C, A < C, \Gamma \longrightarrow \Delta}{A = B, B < C, \Gamma \longrightarrow \Delta} \text{ =<-Tran}$$

$$\frac{A < B, B = C, A < C, \Gamma \longrightarrow \Delta}{A < B, B = C, \Gamma \longrightarrow \Delta} \text{ <=-Tran}$$

Figure 5.4: Auxiliary Rules about Relational Expressions

All these rules are necessary to implement automatic proof search for our application.

There are two types of expressions in the translated formulas: one, called Class 1, are atomic formulas appearing in system axioms and feature specifications; the other called Class 2 are relational expressions of $=$ and $<$. The auxiliary rules do not change formulas of Class 1, but they do influence the formulas of Class 2. Thus, for a given sequent $\Gamma \longrightarrow \Delta$, we separate Γ into sublists Γ_1 and Γ_2 respectively, where Γ_1 contains formulas of Class 2 without *duplicated* forms while Γ_2 contains formulas of Class 1.

We choose to apply rules on Γ_1 according to the following orders: rules of equality, rules of transitivity of "Less-than-Equality" and "Equality-less-than", and transitivity of "Less-than". Actually, the order does not matter as long as we apply all rules about equality before transitivity of "Less-than-Equality" and transitivity of "Equality-Less-than" because there are new equality expressions appended to Γ_1 after applying the former rules which may influence the applications of the latter two rules.

In order to obtain all potential consequences on the left side of a sequent, each auxiliary rule is applied on Γ_1 to acquire a new result sublist Γ_1' which is then passed into the next rule. Finally, after employing all auxiliary rules, we obtain a new sequent $\Gamma_1', \Gamma_2 \longrightarrow \Delta$.

We present abstract descriptions of algorithms for the auxiliary rules in pseudo-code first, and then show and explain how we implement these algorithms in λProlog. After implementation of each rule, an example is presented to give a clear understanding. The following conventions are used in the descriptions of all algorithms in this chapter: **InputList** is the input list Γ_1 for each algorithm; **N** is the length of **InputList** and elements in a list are numbered from 0; **MidList** is used locally in each algorithm during intermediate stages of the algorithm; **OutputList** is the return value of Γ_1'; **InputList[k]** means the k*th* element of **InputList**; variables with the suffix **Rest** are new lists obtained by removing a single element from the given list; variables with the suffix **Append** are new lists obtained by appending new elements to the end of the given list; variables with the suffix **ND** are new lists with no duplicated elements inside.

5.3.1 Algorithm and Implementation for Symmetry of Equality

The algorithm for symmetry of equality $\forall x \forall y((x = y) \longrightarrow (y = x))$ is based on the fact that we can certainly add $(y = x)$ to the left side of a sequent if there is $(x = y)$ there.

Algorithm 2. *Pseudo-code Description of Algorithm for Symmetry of Equality:*

Input parament: **InputList**;

```
type math_eq_sym  (list form) -> (list form)
          -> (list form) -> lprf -> lprf -> o.
type eq_sym  nat -> nat -> lprf -> lprf.
math_eq_sym nil L2 L2 Q2 Q2 :- !.
math_eq_sym ((eq A B)::L1Rest)
    L2 L3 (eq_sym A B Q1) Q2 :-
  append L2 ((eq B A)::nil) L2Append,
  math_eq_sym L1Rest L2Append L3 Q1 Q2.
math_eq_sym (A::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_sym L1Rest L2 L3 Q1 Q2.
```

Figure 5.5: Definite Clauses for Symmetry of Equality

Return: **OutputList**;

1. **MidList** is initialized to be the value of **InputList**;
2. while (**InputList** is not null)
   (a) Extract the first element from **InputList** and the rest of **InputList** is **InputListRest**; if the first element has a form $(A = B)$ and if $(B = A)$ is not already in **MidList**, then append $(B = A)$ to **MidList**;
   (b) **InputList** = **InputListRest**;
3. **OutputList** =**MidList**;

A new predicate **math_eq_sym** is introduced to implement this algorithm in λProlog and we have following type declarations and definite clauses in Figure 5.5*. In this figure, lines 1, 2 and 3 are type declarations for constants **math_eq_sym** and **eq_sym**, which are used to represent the rule name and the proof respectively. There are 5 parameters for **math_eq_sym**: 4 input parameters include **InputList**, **MidList**, **OutputList** and one proof for the upper sequent in this rule; the output is

*We simply use L1, L2 and L3 to represent lists **InputList**, **MidList** and **OutputList**.

the proof by applying this rule. The constant **eq_sym** is the head of a proof and it has 3 input parameters which represent the two integers in $(A = B)$ and one proof for the upper sequent to construct the new proof of type **lprf** for the lower sequent.

The main definite clause from line 5 to line 8 in Figure 5.5 for **math_eq_sym** says: if the first element of **InputList** is of form $(A = B)$ and **Q1** is a proof of the sequent with the new symmetric equality appended to the list, then **Q2** is a proof which save final proof. More precisely, if **Q1** is a proof of $(\Gamma,\ A = B,\ B = A \longrightarrow \Delta)$, then **(eq_sym A B Q1)** is a proof of $(\Gamma, A = B \longrightarrow \Delta)$. **InputList** (first represented by ((eq A B)::L1Rest)) is changed in the recursive call to **L1Rest** since we extract the first element from it every time.

The algorithm stops when **InputList** is null which is shown by the first definite clause for **math_eq_sym**. The last clause for **math_eq_sym** is used when its first element does not have a form of $(A = B)$. For example, it has a form of $(A < B)$. It recursively invokes itself with a new **InputList** by taking out the first element from it.

We illustrate with one example to see how this algorithm works for symmetry of equality in Table 5.2.

FirstElement	InputList	MidList
	$(x_1 = x_2,\ x_2 < x_4, x_2 = x_3)$	$(x_1 = x_2,\ x_2 < x_4, x_2 = x_3)$
$x_1 = x_2$	$(x_2 < x_4, x_2 = x_3)$	$(x_1 = x_2, x_2 < x_4, x_2 = x_3, x_2 = x_1)$
$x_2 < x_4$	$(x_2 = x_3)$	$(x_1 = x_2, x_2 < x_4,\ x_2 = x_3, x_2 = x_1)$
$x_2 = x_3$	()	$(x_1 = x_2, x_2 < x_4, x_2 = x_3, x_2 = x_1,$ $x_3 = x_2)$

Note that: the final value for **MidList** is **OutputList**.

Table 5.2: An Example of Symmetry of Equality

This algorithm is simple and straightforward.

5.3.2 Algorithm and Implementation for Transitivity of Equality

Comparing to the algorithm of symmetry of equality, the algorithm of transitivity of equality is much more complicated.

Using Axiom (5.3) $\forall x\, \forall y\, \forall z\, ((x = y \wedge y = z) \supset (x = z))$, if there are two equality expressions $(x = y)$ and $(y = z)$ in the input list, by applying this rule, then $(x = z)$ is appended to this list if it is not already in it. The complications occur because this new expression $(x = z)$ may also be used with existing elements in the list to conclude new consequences from the transitivity axiom. We present our main idea for this axiom as follows: take out each element from **InputList** as **KeyElement** one by one. Every time compare one **KeyElement** with **TakenOutElement**. **TakenOutElement** comes from the list **InputListRest** obtained by getting rid of **KeyElement** from **InputList**. Each element in this list is considered one by one as **TakenOutElement**. If there is a transitive relation between these two, a new equality expression is added to the original **InputList** if it is not already in it. Continue to compare this **KeyElement** with the other elements from **InputListRest** and add new expressions if possible. Assign **InputList** with the extended list with all new expressions and replace **KeyElement** with the next one from the new **InputList** and repeat until we choose all elements from **InputList** as **KeyElement** once. We achieve this algorithm in two steps: the first step is used to choose **KeyElement**. The second one is to compare it with **TakenOutElements**. Variables with the suffix **Sub** in this step are used to contrast them with those in the first step.

Algorithm 3. *Pseudo-code Description of Algorithm of Transitivity of Equality:*

Input parameters: **InputList**, **N**;

Return: **OutputList**;

1. $i = 0$;
2. while $(i < N)$
   (a) if **InputList**$[i]$ has a form $(A = B)$ then
      i. **KeyElement** = **InputList**$[i]$; **InputListRest** is assigned to be **InputList** without **KeyElement**.
      ii. Call Algorithm 4 with parameters **KeyElement**, **InputListRest**, **InputList**. Assign **OutputListSub** to the return result.
      iii. **InputList** =**OutputListSub**; N is modified by the new length of **InputList**.
   (b) $i = i + 1$;

Algorithm 4. *Pseudo-code Description of Sub-algorithm of Transitivity of Equality:*

Input parameters: **KeyElement**, **InputListRest**, **InputList**

Return: **OutputListSub**;

1. **KeyElementSub** = **KeyElement** (note that it has the form $(A = B)$);
   InputListSub = **InputListRest**; **MidListSub** = **InputList**;
2. while (**InputListSub** is not null)
   (a) Extract the first element from **InputListSub** as **TakenOutElement**, the rest of **InputListSub** is **InputListRestSub**;

(b) if **TakenOutElement** has a form $(B = C)$ and if $(A = C)$ is not already in **MidListSub**, then append $(A = C)$ to **MidListSub**;

(c) **InputListSub=InputListRestSub**;

3. **OutputListSub** = **MidListSub**;

There are several important points in implementation of transitivity of equality:

1. There are a finite number of variables in the given **InputList**. Thus there are a finite number of equalities that can possibly be added by this algorithm. This fact guarantees termination of the algorithm.

2. We do not need to compare new elements chosen from **MidListSub** as **TakenOutElement** with **KeyElement** because the transitivity rule is never applied in these cases. For example, a new expression $(x = z)$ is added by transitivity between $(x = y)$ and $(y = z)$ and it is impossible to obtain new expressions either between $(x = z)$ and $(x = y)$ or between $(x = z)$ and $(y = z)$ by transitivity. But we do change **InputList** by adding new expressions if there are any after employing the sub algorithm.

3. Because we implement Algorithm 2 first, we have all consequences of symmetry already in **InputList.** After applying these two algorithms, all possible equality expressions including the expressions of reflexivity of equality appear explicitly. We explain this in the next subsection.

4. The order of elements in a list is so important that we can not change it when extracting elements from or appending a sublist to

```
type math_eq_tran  int -> (list form) -> (list form)
         -> (list form) -> lprf -> lprf -> o.
type math_eq_tranSub form ->(list form) ->(list form)
         ->(list form) ->lprf ->lprf ->o.
type eq_tran  nat -> nat -> nat -> lprf -> lprf.
math_eq_tran N L1 L2 L3 Q2 Q2 :-
  memCount L1 N1,
  N > N1,
  filter L2 L3, !.
math_eq_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1,
  (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_eq_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub  L2 L2Append,
  filter L2Append L2AppendND,
  math_eq_tran Next L2AppendND L2AppendND L3 Q11 Q2.
math_eq_tranSub X nil L3 L3 Q2 Q2 :- !.
math_eq_tranSub (eq A B)
((eq B C)::L1Rest) L2 L3 (eq_tran A B C Q1) Q2 :-
  append L2 ((eq A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_tranSub X1 L1Rest L2 L3 Q1 Q2 .
```

Figure 5.6: Definite Clauses for Transitivity of Equality

the list because we choose elements as **KeyElement** and **TakenOutElement** one by one in order.

We implement these two algorithms in λProlog in Figure 5.6[†].

Two predicates **math_eq_tran** and **math_eq_tranSub** are declared to implement the first step and the second step of the algorithms respectively. The constant **eq_tran** is used to store proofs in **math_eq_tranSub** when there are new elements added to the sequent by transitivity of equality.

[†]We also simply employ L1, L2 and L3 to represent lists **InputList**, **MidList** and **OutputList**.

We explain the constants in Figure 5.6 as follows:

The predicate **memCount** is used to count the length of a list **L1** as **N1** of type **int**. We use the predicate **pickup** to choose the **N***th* element from a list **L1** as **KeyElement (E1)** of type **form**. **nth_and_rest** is a predicate to construct a new list **L1Rest** as the rest of **L1** by extracting out the **N***th* element. The predicate **filter** helps us delete the redundant elements in a list.

math_eq_tran has one more input parameter than **math_eq_sym** which is a counter for **KeyElement**. Basically, we invoke **math_eq_tran** by assigning this parameter with 1. There are 2 definite clauses for **math_eq_tran**. One is to terminate the algorithm if no more elements can be chosen as **KeyElement** from **InputList** if $(N > N1)$ where **N** is the index of the current number and **N1** is the length of the list. This clause is shown in Figure 5.6 from line 6 to line 9. The second clause for **math_eq_tran** is used to implement step 2 in Algorithm 3 where **KeyElement** of the form $(A = B)$ is taken out by the predicate **pickup** to obtain **L1Rest** (**InputListRest** by extracting **KeyElement** from **L1** (**InputList**), and then call **math_eq_tranSub** to compare it with other elements in **L1Rest**. An extended list with new elements is returned back from the sub algorithm as the new list **L1**. Finally the clause recursively invokes itself with the new **KeyElement** and the new **L1**.

In Figure 5.6, line 3 is the type declaration for **math_eq_tranSub**. The first input parameter of type **form** represents **KeyElement** and we have similar explanations for the rest parameters as we have explained for **math_eq_sym**. The constant **eq_tran** takes three parameters of type **nat** since the transitive property involves three variables in expressions $(A = B)$ and $(B = C)$. **(eq_tran A B C Q1)** is a proof of $(\Gamma,\ A =$

$C \longrightarrow \Delta$) if **Q1** is a proof of (Γ, $A = B$, $B = C \longrightarrow \Delta$).

We demonstrate on a list with only equality expressions to simplify the presentation of the execution of transitivity of equality in Table 5.3.

KeyElement	InputList
	$(x_1 = x_2, x_2 = x_3, x_2 = x_1, x_3 = x_2)$
$x_1 = x_2$	$(\{x_1 = x_2\}, \text{“}x_2 = x_3\text{”}, x_2 = x_1, x_3 = x_2, x_1 = x_3)$
$x_1 = x_2$	$(\{x_1 = x_2\}, x_2 = x_3, \text{“}x_2 = x_1\text{”}, x_3 = x_2, x_1 = x_3, x_1 = x_1*)$
$x_1 = x_2$	$(\{x_1 = x_2\}, x_2 = x_3, x_2 = x_1, \text{“}x_3 = x_2\text{”}, x_1 = x_3, x_1 = x_1)$
$x_2 = x_3$	$(\text{“}x_1 = x_2\text{”}, \{x_2 = x_3\}, x_2 = x_1, x_3 = x_2, x_1 = x_3, x_1 = x_1)$
$x_2 = x_3$	$(x_1 = x_2, \{x_2 = x_3\}, \text{“}x_2 = x_1\text{”}, x_3 = x_2, x_1 = x_3, x_1 = x_1)$
$x_2 = x_3$	$(x_1 = x_2, \{x_2 = x_3\}, x_2 = x_1, \text{“}x_3 = x_2\text{”}, x_1 = x_3, x_1 = x_1, x_2 = x_2*)$
$x_2 = x_3$	$(x_1 = x_2, \{x_2 = x_3\}, x_2 = x_1, x_3 = x_2, \text{“}x_1 = x_3\text{”}, x_1 = x_1, x_2 = x_2)$
$x_2 = x_3$	$(x_1 = x_2, \{x_2 = x_3\}, x_2 = x_1, x_3 = x_2, x_1 = x_3, \text{“}x_1 = x_1\text{”}, x_2 = x_2)$
$x_2 = x_1$	$(\text{“}x_1 = x_2\text{”}, x_2 = x_3, \{x_2 = x_1\}, x_3 = x_2, x_1 = x_3, x_1 = x_1, x_2 = x_2, [x_2 = x_2])$
$x_2 = x_1$	$(x_1 = x_2, \text{“}x_2 = x_3\text{”}, \{x_2 = x_1\}, x_3 = x_2, x_1 = x_3, x_1 = x_1, x_2 = x_2)$
$x_2 = x_1$	$(x_1 = x_2, x_2 = x_3, \{x_2 = x_1\}, \text{“}x_3 = x_2\text{”}, x_1 = x_3, x_1 = x_1, x_2 = x_2)$
$x_2 = x_1$	$(x_1 = x_2, x_2 = x_3, \{x_2 = x_1\}, x_3 = x_2, \text{“}x_1 = x_3\text{”}, x_1 = x_1, x_2 = x_2, [x_2 = x_3])$
$x_2 = x_1$	$(x_1 = x_2, x_2 = x_3, \{x_2 = x_1\}, x_3 = x_2, x_1 = x_3, \text{“}x_1 = x_1\text{”}, x_2 = x_2, [x_2 = x_1])$
$x_2 = x_1$	$(x_1 = x_2, x_2 = x_3, \{x_2 = x_1\}, x_3 = x_2, x_1 = x_3, x_1 = x_1, \text{“}x_2 = x_2\text{”})$
$x_3 = x_2$	$(\text{“}x_1 = x_2\text{”}, x_2 = x_3, x_2 = x_1, \{x_3 = x_2\}, x_1 = x_3, x_1 = x_1, x_2 = x_2)$
$x_3 = x_2$	$(x_1 = x_2, \text{“}x_2 = x_3\text{”}, x_2 = x_1, \{x_3 = x_2\}, x_1 = x_3, x_1 = x_1, x_2 = x_2, x_3 = x_3*)$
$x_3 = x_2$	$(x_1 = x_2, x_2 = x_3, \text{“}x_2 = x_1\text{”}, \{x_3 = x_2\}, x_1 = x_3, x_1 = x_1, x_2 = x_2, x_3 = x_3, x_3 = x_1)$
$x_3 = x_2$	$(x_1 = x_2, x_2 = x_3, x_2 = x_1, \{x_3 = x_2\}, \text{“}x_1 = x_3\text{”}, x_1 = x_1, x_2 = x_2, x_3 = x_3, x_3 = x_1)$
$x_3 = x_2$	$(x_1 = x_2, x_2 = x_3, x_2 = x_1, \{x_3 = x_2\}, x_1 = x_3, \text{“}x_1 = x_1\text{”}, x_2 = x_2, x_3 = x_3, x_3 = x_1)$
$x_3 = x_2$	$(x_1 = x_2, x_2 = x_3, x_2 = x_1, \{x_3 = x_2\}, x_1 = x_3, x_1 = x_1, \text{“}x_2 = x_2\text{”}, x_3 = x_3, x_3 = x_1, [x_3 = x_2])$
$x_1 = x_3$	$(\text{“}x_1 = x_2\text{”}, x_2 = x_3, x_2 = x_1, x_3 = x_2, \{x_1 = x_3\}, x_1 = x_1, x_2 = x_2, x_3 = x_3, x_3 = x_1)$
$x_1 = x_3$	$(x_1 = x_2, \text{“}x_2 = x_3\text{”}, x_2 = x_1, x_3 = x_2, \{x_1 = x_3\}, x_1 = x_1, x_2 = x_2, x_3 = x_3, x_3 = x_1)$
	.....

Note that:

1. {} is used to represent the expression which is chosen as **KeyElement**
2. “” is used to represent **TakenOutElement**
3. [ ] is used to represent duplicated elements which are not appended to **InputList**
4. * is used to represent reflexive equalities.

Table 5.3: An Example of Transitivity of Equality

In this example, although we do not illustrate all comparisons, at the point we stop in Table 5.3, we have already obtained all possible equalities. Note that there are added reflexive equalities marked by *.

5.3.3 Implementation for Reflexivity of Equality

By the following theorem, if an integer n appears in any equality relation, the consequence of reflexivity ($n = n$) appears after Algorithm 2 and 3[‡]. So implementations of these two axioms are enough to implement all these properties on equality.

Theorem.

$$\begin{aligned} & \forall x_1\ \forall x_2\ (x_1 = x_2 \supset x_2 = x_1) \\ \wedge\ & \forall x_1\ \forall x_2\ \forall x_3\ (x_1 = x_2 \wedge x_2 = x_3 \supset x_1 = x_3) \\ \supset\ & \forall x\ \forall y\ (x = y \supset (x = x \wedge y = y)) \end{aligned}$$

Our prover has automatically proved this theorem. Moreover, we illustrate the proof in Natural Deduction [14] with rules in Appendix A.

From the example shown in the above subsection, we notice that $(x_1 = x_1), (x_2 = x_2)$ and $(x_3 = x_3)$ are already in the list.

5.3.4 Algorithm and Implementation for Transitivity between Equality and "Less-than(<)"

Implementations of the other transitivity axioms are very similar to the one for transitivity of equality.

We begin with Axioms (5.4) and (5.5). In the following algorithms, we represent equality expressions as **KeyElement** and compare it with other "less-than" expressions as **TakenOutElement**. The main idea of the algorithms is similar to that of algorithms for transitivity of equality in some sense. This algorithm is also achieved by two steps.

We have the following algorithm of transitivity between $=$ and $<$ expressions.

[‡]In our application, any integers that come from the translation from LTL to FOL always appear in at least one equality expression.

Algorithm 5. *Pseudo-code Description of Algorithm of Transitivity between $=$ and $<$ Expressions:*

Input parameters: **InputList**, N;

Return: **OutputList**;

1. $i = 0$;
2. while $(i < N)$
   (a) if **InputList**$[i]$ has a form $(A = B)$ then
      i. **KeyElement** = **InputList**$[i]$; **InputListRest** is assigned to be **InputList** without **KeyElement**.
      ii. Call Algorithm 6 with parameters **KeyElement**, **InputListRest**, **InputList**. Assign **OutputListSub** to the return result.
      iii. **InputList** =**OutputListSub**; N is modified by the new length of **InputList**.
   (b) $i = i + 1$;

Algorithm 6. *Pseudo-code Description of Sub-algorithm of Transitivity between $=$ and $<$ Expressions*

Input parameters: **KeyElement**, **InputListRest**, **InputList**

Return: **OutputListSub**;

1. **KeyElementSub** = **KeyElement** (note that it has the form $(A = B)$);
   InputListSub = **InputListRest**; **MidListSub** = **InputList**;
2. while (**InputListSub** is not null)

(a) Extract the first element from **InputListSub** as **TakenOutElement**, the rest of **InputListSub** is **InputListRestSub**;

(b) if **TakenOutElement** has the form $(B < C)$ and if $(A < C)$ is not already in **MidListSub**, then append $(A < C)$ to **MidListSub**;

(c) Or if **TakenOutElement** has the other form $(C < A)$ and if $(C < B)$ is not already in **MidListSub**, then append $(C < B)$ to **MidListSub**;

(d) **InputListSub=InputListRestSub**;

3. **OutputListSub** = **MidListSub**;

These two algorithms are implemented in Figure 5.7.

We also have similar explanations for the definite clauses in this figure as we have explained for Figure 5.6. The difference is discussed below:

The big distinction exists in the sub algorithm **math_eq_lt_tranSub** which is more complex than **math_eq_tranSub** because we combine implementations of Axiom (5.4) and Axiom (5.5) together where we have one more clause for **math_eq_lt_tranSub**: when **KeyElement** has the form $(A = B)$, it can possibly be used in applying transitivity between it and **TakenOutElement** of either the form $(B < C)$ for Axiom (5.4) or the form $(C < A)$ for Axiom (5.5).

An example for this algorithm is demonstrated in Table 5.4.

In this example, new expressions $(x_1 < x_6)$ and $(x_6 < x_3)$ as examples, are appended to **InputList**. The former one is appended due to transitivity of "Equality-Less-than" and the latter one is appended due to transitivity of "Less-than-Equality".

```
type math_eq_lt_tran  int -> (list form) -> (list form)
  -> (list form) -> lprf -> lprf -> o.
type math_eq_lt_tranSub form -> (list form) -> (list form)
  -> (list form) -> lprf -> lprf -> o.
type eq_lt_tran nat -> nat -> nat -> nat
  -> lprf -> lprf.
math_eq_lt_tran N L1 L2 L3 Q1 Q1 :-
  memCount L1 N1,  N > N1,
  filter L2 L3,!.
math_eq_lt_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1, (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_eq_lt_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub L2 L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tran Next L2AppendND L2AppendND L3 Q11 Q2.
math_eq_lt_tranSub X nil L3 L3 Q1 Q1 :- !.
math_eq_lt_tranSub (eq A B) ((lt B C)::L1Rest) L2 L3
                   (eq_lt_tran A B B C Q1) Q2 :-
  append L2 ((lt A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_lt_tranSub (eq A B) ((lt C A)::L1Rest) L2 L3
                   (eq_lt_tran A B C A Q1) Q2 :-
  append L2 ((lt C B)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_lt_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_lt_tranSub X1 L1Rest L2 L3 Q1 Q2.
```

Figure 5.7: Definite Clauses for Transitivity between = and < Expressions

KeyElement	InputList
	$(x_1 = x_2, x_3 = x_4, x_5 < x_3, x_6 < x_4, x_2 < x_6)$
$x_1 = x_2$	$(\{x_1 = x_2\}, \text{“}x_3 = x_4\text{”}, x_5 < x_3, x_6 < x_4, x_2 < x_6)$
$x_1 = x_2$	$(\{x_1 = x_2\}, x_3 = x_4, \text{“}x_5 < x_3\text{”}, x_6 < x_4, x_2 < x_6)$
$x_1 = x_2$	$(\{x_1 = x_2\}, x_3 = x_4, x_5 < x_3, \text{“}x_6 < x_4\text{”}, x_2 < x_6)$
$x_1 = x_2$	$(\{x_1 = x_2\}, x_3 = x_4, x_5 < x_3, x_6 < x_4, \text{“}x_2 < x_6\text{”}, x_1 < x_6)$
$x_3 = x_4$	$(\text{“}x_1 = x_2\text{”}, \{x_3 = x_4\}, x_5 < x_3, x_6 < x_4, x_2 < x_6, x_1 < x_6)$
$x_3 = x_4$	$(x_1 = x_2, \{x_3 = x_4\}, \text{“}x_5 < x_3\text{”}, x_6 < x_4, x_2 < x_6, x_1 < x_6, x_5 < x_3)$
$x_3 = x_4$	$(x_1 = x_2, \{x_3 = x_4\}, x_5 < x_3, \text{“}x_6 < x_4\text{”}, x_2 < x_6, x_1 < x_6, x_5 < x_3, x_6 < x_3)$
$x_3 = x_4$	$(x_1 = x_2, \{x_3 = x_4\}, x_5 < x_3, x_6 < x_4, \text{“}x_2 < x_6\text{”}, x_1 < x_6, x_5 < x_3, x_6 < x_3)$
$x_3 = x_4$	$(x_1 = x_2, \{x_3 = x_4\}, x_5 < x_3, x_6 < x_4, x_2 < x_6, \text{“}x_1 < x_6\text{”}, x_5 < x_3, x_6 < x_3)$

Note that:

1. {} is used to represent the expression which is chosen as KeyElement
2. “” is used to represent TakenOutElement

Table 5.4: An Example of Transitivity between = and < Expressions

5.3.5 Algorithm and Implementation for Transitivity of “Less-than(<)”

The last algorithm we present is for transitivity between “Less-than” expressions only. This algorithm is also derived from the algorithm of transitivity of equality. In order to adopt same conventions from **math_eq_tran**, two new predicates for transitivity of “Less-than” are **math_lt_tran** and **math_lt_tranSub** respectively. Changes for **math_lt_tran** from **math_eq_tran** are related to two things. One is the format of **KeyElement** that has a form $(A < B)$ and the other is **TakenOutElement** that has a form $(B < C)$.

Algorithm 7. *Pseudo-code Description of Algorithm of Transitivity of “Less-than”:*

Input parameters: **InputList**, **N**;

Return: **OutputList**;

1. $i = 0$;

2. while $(i < N)$

   (a) if **InputList**$[i]$ has a form $(A < B)$ then

      i. **KeyElement** = **InputList**$[i]$; **InputListRest** is assigned to be **InputList** without **KeyElement**.

      ii. Call Algorithm 8 with parameters **KeyElement**, **InputListRest**, **InputList**. Assign **OutputListSub** to the return result.

      iii. **InputList** =**OutputListSub**; N is modified by the new length of **InputList**.

   (b) $i = i + 1$;

Algorithm 8. *Pseudo-code Description of Sub-algorithm of Transitivity of "Less-than":*

Input parameters: **KeyElement**, **InputListRest**, **InputList**

Return: **OutputListSub**;

1. **KeyElementSub** = **KeyElement** (note that it has the form $(A < B)$);

   InputListSub = **InputListRest**; **MidListSub** = **InputList**;

2. while (**InputListSub** is not null)

   (a) Extract the first element from **InputListSub** as **TakenOutElement**, the rest of **InputListSub** is **InputListRestSub**;

   (b) if **TakenOutElement** has a form $(B < C)$ and if $(A < C)$ is not already in **MidListSub**, then append $(A < C)$ to **MidListSub**;

```
type math_lt_tran int -> (list form) -> (list form)
                  -> (list form) -> lprf -> lprf -> o.
type math_lt_tranSub  form -> (list form) -> (list form)
                  -> (list form) -> lprf -> lprf -> o.
type lt_tran  nat -> nat -> nat -> lprf -> lprf.
math_lt_tran N L1 L2 L3 Q2 Q2 :-
  memCount L1 N1,
  N > N1,
  filter L2 L3,!.
math_lt_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1,
  (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_lt_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub L2 L2Append,
  filter L2Append L2AppendND,
  math_lt_tran Next L2AppendND L2AppendND L3 Q11 Q2.
math_lt_tranSub X nil L3 L3 Q2 Q2 :- !.
math_lt_tranSub (lt A B) ((lt B C)::L1Rest)
          L2 L3 (lt_tran A B C Q1) Q2 :-
  append L2 ((lt A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_lt_tranSub (lt A B) L1Rest L2AppendND L3 Q1 Q2.
math_lt_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_lt_tranSub X1 L1Rest L2 L3 Q1 Q2.
```

Figure 5.8: Definite Clause for Transitivity of "Less-than(<)"

(c) **InputListSub=InputListRestSub**;

3. **OutputListSub = MidListSub**;

Clauses are defined for this algorithm in Figure 5.8.

Implementations for these two algorithms are similar to those we described for transitivity of equality except for the different forms of **KeyElement** and **TakenOutElement**.

An example for transitivity of "Less-than" is demonstrated in Table 5.5.

KeyElement	InputList
	$(x_1 < x_2, x_2 < x_3, x_2 < x_4, x_3 < x_4)$
$x_1 < x_2$	$(\{x_1 < x_2\}, \text{“}x_2 < x_3\text{”}, x_2 < x_4, x_3 < x_4, x_1 < x_3)$
$x_1 < x_2$	$(\{x_1 < x_2\}, x_2 < x_3, \text{“}x_2 < x_4\text{”}, x_3 < x_4, x_1 < x_3, x_1 < x_4)$
$x_1 < x_2$	$(\{x_1 < x_2\}, x_2 < x_3, x_2 < x_4, \text{“}x_3 < x_4\text{”}, x_1 < x_3, x_1 < x_4)$
$x_2 < x_3$	$(\text{“}x_1 < x_2\text{”}, \{x_2 < x_3\}, x_2 < x_4, x_3 < x_4, x_1 < x_3, x_1 < x_4)$
$x_2 < x_3$	$(x_1 < x_2, \{x_2 < x_3\}, \text{“}x_2 < x_4\text{”}, x_3 < x_4, x_1 < x_3, x_1 < x_4)$
$x_2 < x_3$	$(x_1 < x_2, \{x_2 < x_3\}, x_2 < x_4, \text{“}x_3 < x_4\text{”}, x_1 < x_3, x_1 < x_4, [x_2 < x_4])$
$x_2 < x_3$	$(x_1 < x_2, \{x_2 < x_3\}, x_2 < x_4, x_3 < x_4, \text{“}x_1 < x_3\text{”}, x_1 < x_4)$
$x_2 < x_3$	$(x_1 < x_2, \{x_2 < x_3\}, x_2 < x_4, x_3 < x_4, x_1 < x_3, \text{“}x_1 < x_4\text{”})$
	.....

Note that:

1. {} is used to represent the expression which is chosen as KeyElement
2. “” is used to represent TakenOutElement
3. [ ] is used to represent the duplicated elements and is not be appended to InputList

Table 5.5: An Example of Transitivity of “Less-than(<)”

With the assistance of these auxiliary rules, all relational expressions appear explicitly on the left side of a sequent which greatly help our prover to complete the proof search procedure and increase the prover's capability. Our algorithms are straightforward and work in practice for our application.

5.4 Algorithm for An Automated Prover for FOL with Integers and Relational Expressions

Based on the new added algorithms, the complete algorithm for searching for proofs for FOL with integers and relational expressions is described

below. Since we have 5 simpler subformulas in Figure 4.8, we start by putting them into the prover and employ the following proof search algorithm.

Algorithm 9. *Automated Proof Search for FOL with Integers and Relational Expressions:*

Input parameter: **A sequent** representing one of the 5 simpler subformulas

Return: **A proof** (if there is one, infinite loop otherwise)

1. Attempt to apply rules in the following order: initial, $\leqslant$-L, $\leqslant$-R, $\geqslant$-L, $\geqslant$-R, $>$-L, $>$-R, $\wedge$-L, $\supset$-R, $\exists$-L, $\forall$-R, $\vee$-L, $\wedge$-R, $\supset$-L, $\neg$-R, $\vee$-R, $\neg$-L, and exit if a proof is found. Note that there are no rules $\exists$-R or $\forall$-L in this list.

2. Apply the following algorithms to the lists of formulas on the left side of the sequent obtained from Step 1.

   (a) Algorithm 2 for Axiom 5.2;

   (b) Algorithm 3 for Axiom 5.3;

   (c) Algorithm 5 for Axiom 5.4 and 5.5;

   (d) Algorithm 7 for Axiom 5.6;

   Repeat the following steps. Exit whenever a proof is found:

3. Continue search, attempting all the rules in the following order: initial,$\leqslant$-L, $\leqslant$-R, $\geqslant$-L, $\geqslant$-R, $>$-L, $>$-R, $\wedge$-L, $\supset$-R, $\exists$-L, $\forall$-R, $\vee$-L, $\wedge$-R, $\supset$-L, $\neg$-R, $\vee$-R, $\neg$-L, $\exists$-R, $\forall$-L. Note that this list includes rules $\exists$-R and $\forall$-L.

4. Amplification step: If a proof is not found, then using the sequent that is obtained at the end of Step 2, add a new copy of universally

quantified formulas on the left and existentially quantified formulas on the right and go to Step 3.

Note that although the order in which inference rules are attempted is not important for completeness, it is certainly important for efficiency. For example, it is better to put the **initial** rule as early as possible. Also, our experiments have shown that for our application, it is better to put the ∃-R and ∀-L rules in the end of the list in Step 3.

Implementation for this algorithm is presented in Appendix B.

5.5 A Proof Checker for FOL with Integers and Relational Expressions

Integrating these additional and auxiliary rules into the checker is much easier than integrating them into the prover.

We still need to replace the predicate **memb_and_rest** by **memb** when we implement these rule in the checker. For example, implementation of the rule >-**R** is presented below:

```
(gt_r A B Q) >- (Gamma --> Delta) :-
  memb (gt A B) Delta ,
  Q >- (Gamma --> ((lt B A)::Delta)).
```

where as long as there is $(A > B)$ on the right side of the sequent, we add $(B > A)$.

We also take the transitivity of equality as an example of the implementation of the auxiliary rules.

```
(eq_tran A B C Q) >- (Gamma --> Delta) :-
```

```
memb (eq A B) Gamma,
memb (eq B C) Gamma,
Q >- (((eq A C)::Gamma) --> Delta).
```

This definite clause says that if there are $(A = B)$ and $(B = C)$ on the left side of the lower sequent, there is also $(A = C)$ inside the upper sequent.

Implementation of the checker is shown in Appendix C.

Chapter 6

Summary of Experimental Results

In this chapter, we present the experiments of finding the conflict caused by requirements of ACR and CFBL that we discussed in Chapter 4. We also present and prove two other examples of conflicts caused by other requirements of ACR and CFBL which also fit into the same pattern. We summarize the experimental results at the end of this section.

6.1 Experimental Results

In previous chapters, according to the automated proof search algorithm, we developed and implemented a successful prover in *Teyjus*.

In order to prove the FOL formulas for each case described in Chapter 4, we declare a new predicate **prove** as a top-level predicate used for queries in Figure 6.1.

The predicate **prove** takes two parameters, a formula's name of type **name** and a proof of type **lprf**, to construct a goal formula in λProlog. In *Teyjus*, after loading all necessary modules, we can query (**prove**

```
type prove name -> lprf -> o.
type check name -> o.
type proof name -> lprf -> o.

prove Name Q :-
  formula Name Formula,
  Q >-1 (nil --> (Formula::nil)).
check Name :-
 formula Name Formula,
 proof Name Q,
 Q >- (nil --> (Formula::nil)).

?- prove case1 Q.
?- prove case2 Q.
?- prove case3 Q.
?- prove case4 Q.
?- prove case5 Q.
```

Figure 6.1: Examples of Proof Queries in λProlog

case1 Q), **(prove case2 Q)**, **(prove case3 Q)**, **(prove case4 Q)** and **(prove case5 Q)** respectively to ask for proofs for each case. The goal formulas are illustrated in Figure 6.1 which follow the token **?-** and they can be invoked on the command line.

We query **(prove case1 Q)** as an example. From the definite clause for the predicate **prove**, this goal formula is changed to the query **(Q >- 1 (nil -> (Formula::nil)))** where the variable **Formula** is assigned to be an exact formula by matching **(formula case1 Formula)** based on facts in Figure 4.8. The proof search procedure begins with querying **>- 1** clauses, which achieve a sequent with atomic formulas and universally quantified formulas which are on the left side and existentially quantified formulas which are on the right side. After running amplification, **>-2** clauses are invoked to achieve a successful result for **Q** from the prover in one second.

The proofs obtained from the prover are complex and not easy to read.

To help us justify the results, we define a new predicate **check** to verify whether these results satisfy the proved formulas or not. The clause for this predicate is also shown in Figure 6.1. The other new predicate **proof** is used to store the concrete value of a formula's proof coming from the prover. For this example, we copy the proof resulting from the prover for **case1** with the predicate **proof** and **case1** as a fact. Thus the query **check case1** invokes the query **Q >- (nil -> (Formula::nil))** which runs the checker described in chapter 5 to decide whether **Q** is a correct proof of **case1**. We get successful responses from the checker for these cases.

Also, we notice that cost of proof search for the first four cases is less than that of the last case by about one third. The reason for the longer time cost for the last case is that all four specification formulas are involved in this case as we described in Figure 4.5.

Now we conclude for this example: there is a proof for each of these 5 cases, which means there is a conflict between the two requirements, both fitting the feature pattern we focus on.

6.2 Detecting Feature Conflicts Caused by Other Examples

To further illustrate our theorem proving approach, we make an effort to apply it to other pairs of feature specifications which also fit the specific pattern described in Chapter 4.

We examined two other pairs of requirements of features ACR and CFBL.

In addition to CFBL_Normal_Operation_1 described in Chapter 4

which specifies the normal operation of CFBL, there are two other requirements, which describe exceptions to its normal operation, represented by the general forms below:

property CFBL_Exception_Operation_1

{

event: $cfbl(x) \wedge \neg idle(x) \wedge \exists y\ forwarding(x, y, z) \wedge call\_req(x, y)$

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

persists: $call\_req(x, y)$

until: $busy\_tone(y) \wedge \neg forwarding(x, y, z)$

discharge: $onhook(y)$

}

This exception expresses that when y calls x who is not idle, if there is already a call to x which is being forwarded to z at the same time, then y hears a busy tone represented by $busy\_tone(y) and$ his/her call cannot be forwarded.

The other exception captures a forwarding loop when the incoming call from y has been already forwarded to z more than 5 times. (The requirements [1] assume 5 is the maximum time that a call can be forwarded.)

property CFBL_Exception_Operation_2

{

event: $cfbl(x) \wedge \neg idle(x) \wedge \neg le\_five\_forwards(y) \wedge call\_req(x, y)$

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

persists: $call\_req(x, y)$

until: $busy\_tone(y) \wedge \neg forwarding(x, y, z)$

discharge: $onhook(y)$

}

The new atomic formula $le\_five\_forward(y)$ expresses that y's call has been forwarded no more than 5 consecutive times. If y's call has been forwarded to z more than 5 times, then his/her call cannot be forwarded and he/she receives a busy tone.

These two properties also fit the pattern (4.6). We specify the following two LTL formulas by using subscript 3 for the first exception requirement and subscript 4 for the second exception requirement respectively:

$$\Box[e_3 \supset (p_3 \mathsf{U} (r_3 \vee d_3))] \tag{6.1}$$

where e_3, p_3, r_3 and d_3 are:

$e_3 := cfbl(x) \wedge \neg idle(x) \wedge \exists y\ forwarding(x, y, z) \wedge call\_req(x, y)$

$p_3 := call\_req(x, y)$

$r_3 := busy\_tone(y) \wedge \neg forwarding(x, y, z)$

$d_3 := onhook(y)$

$$\Box[e_4 \supset (p_4 \mathsf{U} (r_4 \vee d_4))] \tag{6.2}$$

where e_4, p_4, r_4 and d_4 are:

$e_4 := cfbl(x) \wedge \neg idle(x) \wedge \neg le\_five\_forwards(y) \wedge call\_req(x, y)$

$p_4 := call\_req(x, y)$

$r_4 := busy\_tone(y) \wedge \neg forwarding(x, y, z)$

$d_4 := onhook(y)$

Both of these two exceptions conflict with ACR_Normal_Operation_3 because both resolutions in the exception requirements request that y's call should not be forwarded and y hears a busy tone but this ACR property requires that y receives the ACR denial announcement.

As we said before, when adding new feature requirements, there are

formulas that may be added to SA. It is obvious that $busy\_tone(y)$ and $acr\_annc(y, x)$ cannot hold simultaneously in these two cases. By adding $\neg(busy\_tone(y) \wedge acr\_annc(y, x))$ to SA, the following two LTL formulas are used to detect conflicts between the two different kinds of CFBL exception requirements and ACR_Normal_Opertion_3.

$$\neg(\Box[e_1 \supset (p_1 \,\mathsf{U}\, (r_1 \vee d_1)] \wedge \Box[e_3 \supset (p_3 \,\mathsf{U}(r_3 \vee d_3)] \\ \wedge \Box[SA] \wedge \Diamond[e_1 \wedge e_3 \wedge g]) \tag{6.3}$$

Particularly, for the case of requirements between CFBL_Exception_Operation_1 and ACR_Normal_Opertion_3, SA and g are specified:

$$SA := \neg((acr\_annc(y, x) \wedge forwarding(x, y, z)) \\ \wedge\neg(acr\_annc(y, x) \wedge call\_req(x, y)) \\ \wedge\neg(forwarding(x, y, z) \wedge call\_req(x, y)) \\ \wedge\neg(onhook(y) \wedge call\_req(x, y)) \\ \wedge\neg(onhook(y) \wedge forwarding(x, y, z)) \\ \wedge\neg(busy\_tone(y) \wedge acr\_annc(y, x))$$

$$g := (p_1 \wedge p_3) \,\mathsf{U}\, (\neg p_1 \wedge \neg p_3 \wedge \neg d_1 \wedge \neg d_3)$$

$$\neg(\Box[e_1 \supset (p_1 \,\mathsf{U}\, (r_1 \vee d_1)] \wedge \Box[e_4 \supset (p_4 \,\mathsf{U}(r_4 \vee d_4)] \\ \wedge \Box[SA] \wedge \Diamond[e_1 \wedge e_4 \wedge g]) \tag{6.4}$$

We have the same SA in formula (6.4) for detecting conflicts between CFBL_Exception_Operation_2 and ACR_Normal_Opertion_3, but we have a different value for g as follows:

$$g := (p_1 \wedge p_4) \,\mathsf{U}\, (\neg p_1 \wedge \neg p_4 \wedge \neg d_1 \wedge \neg d_4)$$

We implemented these two LTL formulas in λProlog, followed the

same steps we designed to handle the previous example in chapter 4, constructed queries of the similar 5 simpler subformulas for each of these two LTL formulas respectively and succeeded in searching for proofs using our prover and verifying the proofs using our checker. All of these cases cost approximately the same amount of time as spent for the previous cases.

6.3 Comparison of Implementation in Teyjus and Terzo

There are several existing interpreters and compilers for λProlog. Before considering implementations of our application in *Teyjus*, we began with them in *Terzo* [20]. But for our complicated application, the application of detecting feature conflicts, implementation in *Terzo* is not tolerable because of the slow speed of this interpreter written in Standard ML. This is the reason that we then shifted all implementations to *Teyjus* and modified code to fit syntax requirements of *Teyjus* and achieved a powerful and efficient prover with one or two orders of magnitude improvement in proof search time.

Chapter 7

Conclusion

7.1 Summary of Contributions

In this thesis we illustrated an application of detecting feature conflicts in telecommunications systems by an approach in theorem proving. We addressed this problem of finding an automated prover to search for contradictions which reveal the fact that some independent feature requirements cannot hold simultaneously in a telephony system. Previous research helped us specify feature requirements and feature conflicts by LTL formulas. These specifications were translated into FOL formulas with integers. New algorithms about relational expressions were developed and integrated into an existing prover. We expressed our strategies on how to organize the inference rules, the additional and the auxiliary rules, and then developed an automated prover for FOL formulas with integers to complete and optimize the proof search procedure. This approach was fully demonstrated by the experimental results in λProlog. Examples were also presented during our discussion.

The major contributions of this thesis are:

- Algorithms for dealing with relational expressions. There are relational expressions in the translated FOL formulas and the existing prover did not have the capability to handle them. We presented specialized algorithms for properties of relational expressions containing "Equality" and "Less-than", implemented these algorithms in λProlog, and then integrated them into the prover to develop a specialized proof search procedure.

- Reorganization of the order of application of the inference rules. The existing prover works well for general FOL formulas. From the viewpoint of logic programming, the order of the inference rules is not important for completeness of the proof search procedure. But from experiments, we noticed that arranging the order of these inference rules improves on efficiency and effectiveness of our prover in proof search time.

- Complete implementation and full illustration of the possibility of the theorem proving potential with experiments in detecting feature conflicts.

The minor contributions of this thesis are:

- Implementation of the inference rules in the sequent proof system in *Teyjus*. Since we begin with an existing prover, which handles general FOL formulas [5], we migrated the whole implementation from *Terzo* to *Teyjus* and improved upon performance of this prover.

- Implementation of a translation function from LTL formulas into FOL formulas. We implemented a translation function to transform LTL formulas into FOL formulas with integers in *Teyjus*. We presented the results of applying this function on examples in Chapter 4.

7.2 Future Work

7.2.1 Specifications of Features and Feature Conflicts in Other Patterns

In Chapter 4, we presented a general form to represent feature requirements. The easiest way to specify the general form by an LTL formula is to consider its negation: the property is false if and only if there is a point where the precondition pattern holds, but is not followed by the postcondition pattern [8, 9].

In the next step, we will try to implement specification formulas in more complicated patterns. For example, a property of the following general form:

property <Name>

{

event: e_0 persists: p_0

event: e_1

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

persists: p until: r discharge: d

}

can be specified by an LTL formula according to the general form as below [7]:

$$\neg\Diamond(e_0 \wedge \mathsf{X}\,((p_0 \wedge \neg e_0)\ \mathsf{U}\ (e_1 \wedge \neg(p\ \mathsf{U}\ (r \vee d))))). \tag{7.1}$$

Formula (7.1) expresses a property that must hold after two enabling events e_0 and e_1 happen in sequence. In particular, the precondition on the property is that e_0 is followed by e_1 with p_0 persisting between them.

The postcondition is $(p \mathsf{U} (r \vee d))$, which states after this series of events, a second persisting condition p holds until there is either a resolution r or a discharge d.

We are also interested in a common form with **unless** in the postcondition rather then **until**.

property <Name>

{

event: e

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

persists: p **unless**: r

}

We specify this property by Formula (7.2) [7].

$$\Box[e \supset \mathsf{X}(\neg p \mathsf{W} r)] \tag{7.2}$$

This formula is often used to express a safety property: if there is some enabling event e, in the next state, then some property p should not hold between e and r. For example, if the callee x is idle, the caller y should not hear a busy tone between duration of the call.

By appropriately adjusting g in Formula (4.6) and adding more system axioms to SA, we believe our prover must certainly handle these formulas similarly to the cases studied in this thesis.

7.2.2 Comparison with Existing Formal Method Tools

So far, we have not been able to do a detailed comparison between the model checking approach [8, 9] and theorem proving approach [7] because all examples we tried in theorem proving are the same as those in model checking. We are able to conclude that our theorem prover is at least

as efficient as our model checking approach by the experimental results. But since we have not tried feature conflicts in other complex patterns, we have not challenged our prover with formulas with more general use of quantifiers. Our future work includes more thorough comparison to the model checking approach and adopting examples with more sophisticated use of quantification in formulas specifying feature requirements.

Appendix A

Rules and Proofs in Natural Deduction

Natural Deduction (ND) is another proof system for first-order classical Logic. Proving a formula A from the assumptions Γ in ND corresponds to building a proof tree for $\Gamma \longrightarrow (A :: nil)$ in the sequent proof system. We have inference rules in ND, shown in Figure A.1 [14].

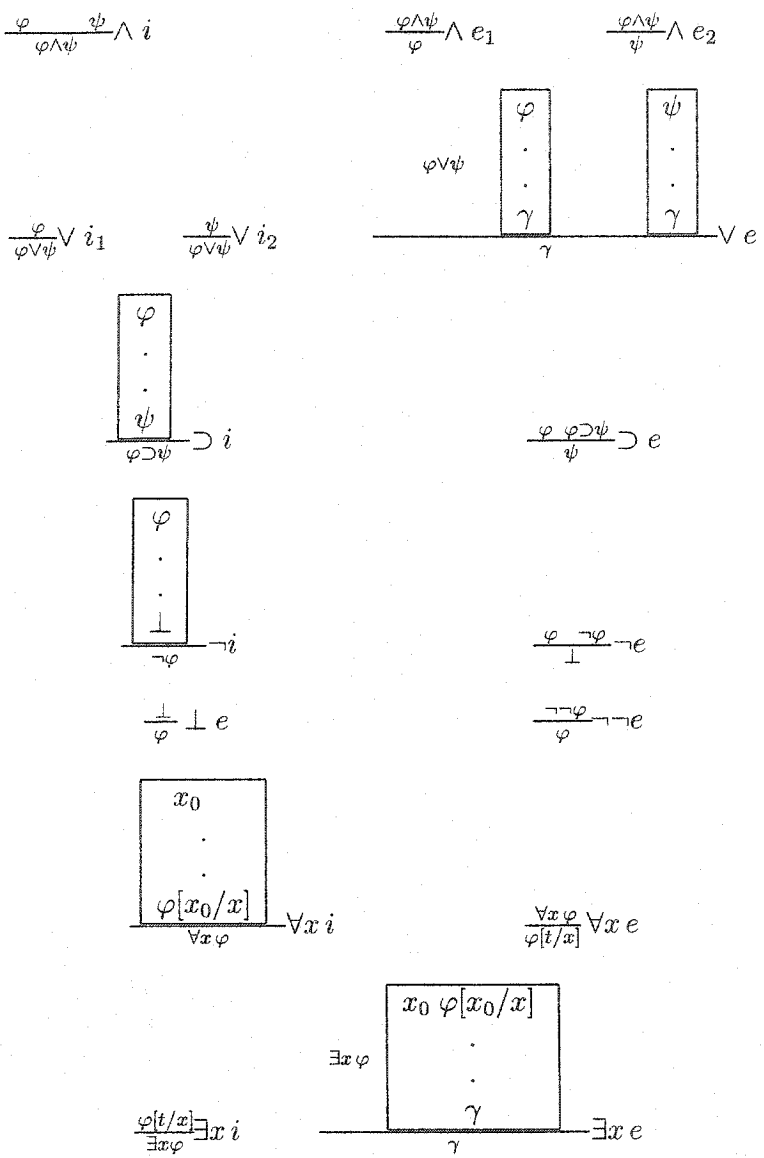

Figure A.1: Natural Deduction Rules for First-Order Classical Logic

Theorem.

$$\begin{aligned} & \forall x_1\, \forall x_2\, (x_1 = x_2 \supset x_2 = x_1) \\ \wedge\ & \forall x_1\, \forall x_2\, \forall x_3\, (x_1 = x_2 \wedge x_2 = x_3 \supset x_1 = x_3) \\ \supset\ & \forall x\, \forall y\, (x = y \supset (x = x \wedge y = y)) \end{aligned}$$

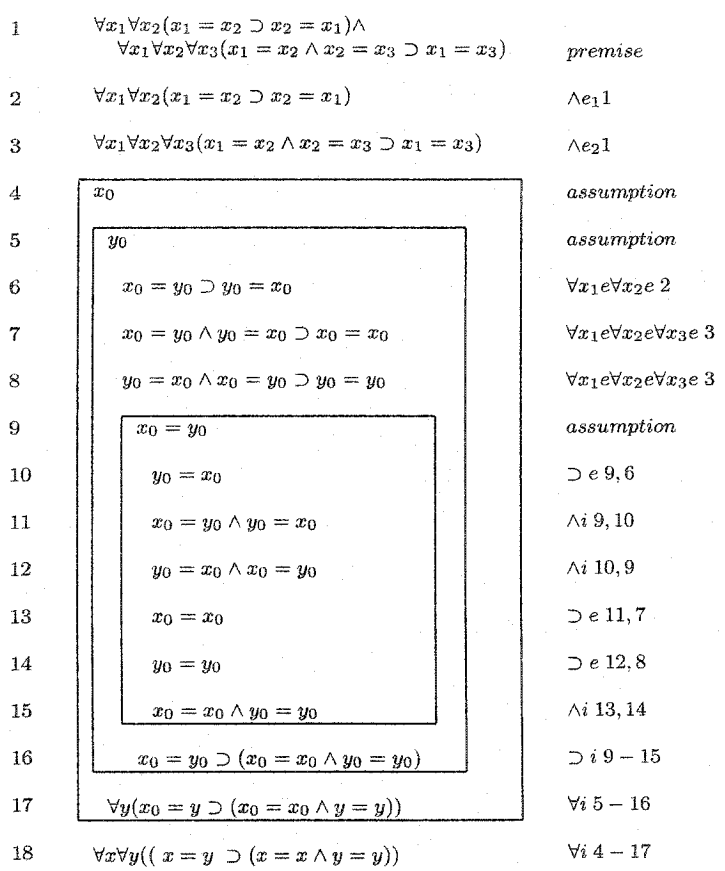

Figure A.2: A Proof for the Theorem in ND

Appendix B

Implementation of A Theorem Prover for FOL Formulas with Relational Expressions

We present related modules for a theorem prover for FOL formulas with integers and relational expressions.

```
% This module provides basic declarations.
module  basic.
type    memb        A -> list A -> o.
type    member      A -> list A -> o.
type    append      list A -> list A -> list A -> o.
type    append_item     list A -> A -> list A -> o.
type    join        list A -> list A -> list A -> o.
type    memb_and_rest   A -> list A -> list A -> o.
type    nth_item        int -> A -> list A -> o.
type    nth_and_rest    int -> A -> list A -> list A -> o.
memb X (X :: L).
memb X (Y :: L) :- memb X L.
member X (X :: L) :- !.
member X (Y :: L) :- member X L.
append nil K K.
append (X :: L) K (X :: M) :- append L K M.
append_item K L M  :- append K (L::nil) M.
join nil K K.
join (X :: L) K M :- memb X K, !, join L K M.
join (X :: L) K (X :: M) :- join L K M.
memb_and_rest A (A :: Rest) Rest.
memb_and_rest A (B :: Tail) (B :: Rest) :- memb_and_rest A Tail Rest.
nth_item 0 A List :- !, memb A List.
nth_item 1 A (B::Rest) :- !, A = B.
nth_item N A (B::Tail) :- M is (N - 1), nth_item M A Tail.
nth_and_rest 0 A List Rest :- !, memb_and_rest A List Rest.
nth_and_rest 1 A (B::Rest) Rest :- !, A = B.
nth_and_rest N A (B::Tail) (B::Rest) :-
  M is (N - 1), nth_and_rest M A Tail Rest.
```

Figure B.1: basic.mod

```
% This module provides declarations for connectives, quantifiers,
% temporal connectives and relational operators in first-order logic
module  fol.
kind form type.
kind nat type.
type and form -> form -> form.
type or  form -> form -> form.
type imp form -> form -> form.
type neg form -> form.
type forall  (A -> form) -> form.
type exists (A -> form) -> form.
type box form -> form.
type diamond form -> form.
type until form -> form -> form.
type weak  form -> form -> form.
type next  form -> form.
type gt  nat -> nat -> form.
type lt  nat -> nat -> form.
type ge  nat -> nat -> form.
type le  nat -> nat -> form.
type eq  nat -> nat -> form.
infixl  or      250.
infixl  and     250.
infixr  imp     240.
infixl  until   240.
infixl  weak    240.
```

Figure B.2: fol.mod

```
% This module is the "signature" file for building sequent-style proofs.
module      lprf.
accumulate  fol.
kind lprf         type.
type initial      form -> lprf.
type and_l        A -> A -> lprf -> lprf.
type and_r        A -> A -> lprf -> lprf -> lprf.
type or_l         A -> A -> lprf -> lprf -> lprf.
type or_r         A -> A -> lprf -> lprf.
type imp_r        A -> A -> lprf -> lprf.
type imp_l        A -> A -> lprf -> lprf -> lprf.
type neg_r        A -> lprf -> lprf.
type neg_l        A -> lprf -> lprf.
type exists_r     A -> lprf -> lprf.
type exists_l     (A -> lprf) -> lprf.
type forall_r     (A -> lprf) -> lprf.
type forall_l     A -> lprf -> lprf.
type eq_sym       nat -> nat -> lprf -> lprf.
type eq_tran      nat -> nat -> nat -> lprf -> lprf.
type lt_tran      nat -> nat -> nat -> lprf -> lprf.
type eq_lt_tran   nat -> nat -> nat -> nat -> lprf -> lprf.
type gt_r         nat -> nat -> lprf -> lprf.
type gt_l         nat -> nat -> lprf -> lprf.
type ge_r         nat -> nat -> lprf -> lprf.
type ge_l         nat -> nat -> lprf -> lprf.
type le_r         nat -> nat -> lprf -> lprf.
type le_l         nat -> nat -> lprf -> lprf.
```

Figure B.3: lprf.mod

```
% This module provides implementation of auxiliary rules
module lc_math.
accumulate  basic, lprf.
type    rearrange   (list form) -> (list form) -> lprf -> lprf -> o.
type    separate    (list form) -> (list form) -> (list form) -> o.
type    separateSub (list form) -> (list form) -> (list form)
          -> (list form) -> (list form) -> o.
type    math_components (list form) -> (list form) -> lprf -> lprf -> o.
type    math_eq_sym (list form) -> (list form)
                    -> (list form) -> lprf -> lprf -> o.
type    math_eq_tran    int -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    math_lt_tran    int -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    math_eq_lt_tran int -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    math_eq_tranSub form -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    math_lt_tranSub form -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    math_eq_lt_tranSub form -> (list form) -> (list form)
                     -> (list form) -> lprf -> lprf -> o.
type    pickup      int -> (list form) -> form -> o.
type    filterSub   (list A) -> (list A) -> (list A) -> o.
type    filter      (list A) -> (list A) -> o.
type    memCount    (list form) -> int -> o.
% rearrange all components
rearrange L1 L2 Q1 Q2 :-
  separate L1 Lmath Lnon,
  math_components Lmath LmathOutput Q1 Q2,
  append Lnon LmathOutput L2.
% separate math components and non-math components
separate L1 Lmath Lnon :-
  separateSub L1 nil nil Lmath Lnon.
separateSub nil Lmath Lnon Lmath Lnon :- !.
separateSub ((eq A B)::L1Rest) LmathSub LnonSub Lmath Lnon :-
  append LmathSub ((eq A B)::nil) LmathSubAppend,
  separateSub L1Rest LmathSubAppend LnonSub Lmath Lnon,!.
separateSub ((lt A B)::L1Rest) LmathSub LnonSub Lmath Lnon :-
  append LmathSub ((lt A B)::nil) LmathSubAppend,
  separateSub L1Rest LmathSubAppend LnonSub Lmath Lnon,!.
separateSub (A::L1Rest) LmathMid LnonMid Lmath Lnon :-
  append LnonMid (A::nil) LnonMidAppend,
  separateSub L1Rest LmathMid LnonMidAppend Lmath Lnon, !.
% eq_tran, lt_tran, eq_sym, eq_rel to gather all math components
math_components Lmath LmathOutput Q1 Q2 :-
  math_eq_sym Lmath Lmath LeqSym Q1 QeqSym,
  math_eq_tran 1 LeqSym nil LeqTran QeqSym QeqTran,
  math_eq_lt_tran 1 LeqTran nil LeqltTran QeqTran QeqltTran,
  math_lt_tran 1 LeqltTran nil LmathOutput QeqltTran Q2.
% algorithm for math_eq_sym
math_eq_sym nil L2 L2 Q2 Q2 :- !.
math_eq_sym ((eq A B)::L1Rest) L2 L3 (eq_sym A B Q1) Q2 :-
  append L2 ((eq B A)::nil) L2Append,
  math_eq_sym L1Rest L2Append L3 Q1 Q2.
math_eq_sym (A::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_sym L1Rest L2 L3 Q1 Q2.
```

```
% algorithm for meth_eq_tran
math_eq_tran N L1 L2 L3 Q2 Q2 :-
  memCount L1 N1,
  N > N1,
  filter L2 L3, !.
math_eq_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1,
  (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_eq_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub  L2 L2Append,
  filter L2Append L2AppendND,
  math_eq_tran Next L2AppendND L2AppendND L3 Q11 Q2.
% algorithm for math_eq_tranSub
math_eq_tranSub X nil L3 L3 Q2 Q2 :- !.
math_eq_tranSub (eq A B) ((eq B C)::L1Rest) L2 L3
    (eq_tran A B C Q1) Q2 :-
  append L2 ((eq A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_tranSub X1 L1Rest L2 L3 Q1 Q2 .
% algorithm for math_lt_tran
math_lt_tran N L1 L2 L3 Q2 Q2 :-
  memCount L1 N1,
  N > N1,
  filter L2 L3,!.
math_lt_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1,
  (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_lt_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub L2 L2Append,
  filter L2Append L2AppendND,
  math_lt_tran Next L2AppendND L2AppendND L3 Q11 Q2.
% algorithm for math_lt_tranSub
math_lt_tranSub X nil L3 L3 Q2 Q2 :- !.
math_lt_tranSub (lt A B) ((lt B C)::L1Rest) L2 L3
                (lt_tran A B C Q1) Q2 :-
  append L2 ((lt A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_lt_tranSub (lt A B) L1Rest L2AppendND L3 Q1 Q2.
math_lt_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_lt_tranSub X1 L1Rest L2 L3 Q1 Q2.
% algorithm for math_eq_lt_tran
math_eq_lt_tran N L1 L2 L3 Q1 Q1 :-
  memCount L1 N1,
  N > N1,
  filter L2 L3.
math_eq_lt_tran N L1 L2 L3 Q1 Q2 :-
  memCount L1 N1,
  (N < N1; N = N1),
  pickup N L1 E1,
  nth_and_rest N E1 L1 L1Rest,
  math_eq_lt_tranSub E1 L1Rest L1 L1Sub Q1 Q11,
  Next is N + 1,
  append  L1Sub L2 L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tran Next L2AppendND L2AppendND L3 Q11 Q2.
```

```
% algorithm for math_eq_lt_tran
math_eq_lt_tranSub X nil L3 L3 Q1 Q1 :- !.
math_eq_lt_tranSub (eq A B) ((lt B C)::L1Rest) L2 L3
        (eq_lt_tran A B B C Q1) Q2 :-
  append L2 ((lt A C)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_lt_tranSub (eq A B) ((lt C A)::L1Rest) L2 L3
        (eq_lt_tran A B C A Q1) Q2 :-
  append L2 ((lt C B)::nil) L2Append,
  filter L2Append L2AppendND,
  math_eq_lt_tranSub (eq A B) L1Rest L2AppendND L3 Q1 Q2.
math_eq_lt_tranSub X1 (X2::L1Rest) L2 L3 Q1 Q2 :-
  math_eq_lt_tranSub X1 L1Rest L2 L3 Q1 Q2.
% E is one element which is the N item in a list
pickup 1 (E::L) E.
pickup N (E1::LRest) E :-
  Next is N - 1,
  pickup Next LRest E.
% eliminate the duplicate elements in a list
filter L1 L2 :- filterSub L1 nil L2.
% a help clause for filter
filterSub nil L L.
filterSub (A::L1) L2 L3 :-
  memb_and_rest A L1 L1first,
  append L2 (A::nil) L2first,
  filterSub L1first L2first L3,!.
filterSub (A::L1) L2 L3 :-
  append L2 (A::nil) L2first,
  filterSub L1 L2first L3,!.
% Count the number of a list
memCount nil 0.
memCount (A::L) C1 :-
  memCount L C2,
  C1 is C2 + 1.
```

Figure B.4: lc_math.mod

```
% This module contains the >-1 program for performing the first step
% of an automatic theorem prover for
% a sequent system for first-order classical logic.
module lc_auto.
accumulate  lc_iter.
type >-1 lprf -> seq -> o.
infixl >-1 200.
(initial A) >-1 (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(le_l A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (le A B) Gamma Gamma1,
  Q >-1 ((((lt A B) or (eq A B))::Gamma1) --> Delta).
(le_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (le A B) Delta Delta1,
  Q >-1 (Gamma --> ((lt A B)::(eq A B)::Delta1)).
(ge_l A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (ge A B) Gamma Gamma1,
  Q >-1 ((((lt B A) or (eq A B))::Gamma1) --> Delta).
(ge_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (ge A B) Delta Delta1,
  Q >-1 (Gamma --> ((lt B A)::(eq A B)::Delta1)).
(gt_l A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (gt A B) Gamma Gamma1,
  Q >-1 (((lt B A)::Gamma1) --> Delta).
(gt_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (gt A B) Delta Delta1,
  Q >-1 (Gamma --> ((lt B A)::Delta1)).
(and_l A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1,
  Q >-1 ((A::B::Gamma1) --> Delta).
(imp_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Delta Delta1,
  Q >-1 ((A::Gamma) --> (B::Delta1)).
(exists_l Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (exists A) Gamma Gamma1,
  pi T \ ((Q T) >-1 (((A T)::Gamma1) --> Delta)).
(forall_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  pi T \ ((Q T) >-1 (Gamma --> ((A T)::Delta1))).
(or_l A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Gamma Gamma1,
  Q1 >-1 ((A::Gamma1) --> Delta),
  Q2 >-1 ((B::Gamma1) --> Delta).
(and_r A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >-1 (Gamma --> (A::Delta1)),
  Q2 >-1 (Gamma --> (B::Delta1)).
(imp_l A B Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Gamma Gamma1,
  Q1 >-1 (Gamma1 --> (A::Delta)),
  Q2 >-1 ((B::Gamma1) --> Delta).
(neg_r A Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Delta Delta1,
  Q >-1 ((A::Gamma) --> Delta1).
(or_r A B Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Delta Delta1,
  Q >-1 (Gamma --> (A::B::Delta1)).
(neg_l A Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Gamma Gamma1,
  Q >-1 (Gamma1 --> (A::Delta)).
Q1 >-1 (Gamma --> Delta)  :-
  rearrange Gamma Gamma1 Q1 Q2,
  nprove 1 Q2 (Gamma1 --> Delta).
```

Figure B.5: lc_auto.mod

```
% This module performs the iteration step for an automatic theorem prover for
% a sequent system for first-order classical logic.
module lc_iter.
accumulate  lc_prover.
type add_copies int -> form
       -> (list form) -> (list form) -> o.
type amplify_forall int -> (list form) -> (list form) -> o.
type amplify_exists int -> (list form) -> (list form) -> o.
type amplify int -> seq -> seq -> o.
type nprove int -> lprf -> seq -> o.
add_copies 1 A List (A::List).
add_copies N A List (A::List1) :-
  (N > 1), M is (N - 1),
  add_copies M A List List1.
amplify_forall N nil nil.
amplify_forall N ((forall A)::Gamma) Gamma2 :-
  amplify_forall N Gamma Gamma1,
  add_copies N (forall A) Gamma1 Gamma2.
amplify_forall N (A::Gamma) (A::Gamma1) :-
  amplify_forall N Gamma Gamma1.
amplify_exists N nil nil.
amplify_exists N ((exists A)::Delta) Delta2 :-
  amplify_exists N Delta Delta1,
  add_copies N (exists A) Delta1 Delta2.
amplify_exists N (A::Delta) (A::Delta1) :-
  amplify_exists N Delta Delta1.
amplify 1 Seq Seq :- !.
amplify N (Gamma1 --> Delta1) (Gamma2 --> Delta2) :-
  amplify_forall N Gamma1 Gamma2,
  amplify_exists N Delta1 Delta2.
nprove N Q Seq1 :-
  print "\nAttempting to prove the following sequent at amplification  ",
  term_to_string N NS, print NS, print "\n",
  term_to_string Seq1 NSeq1, print NSeq1, print "\n",
  amplify N Seq1 Seq2,
  Q >-2 Seq2,
  trace_term "successful".
nprove N Q Seq :-
  M is (N + 1), nprove M Q Seq.
```

Figure B.6: lc_iter.mod

```
% This module is the main prover for an automatic theorem prover for
% a sequent system for first-order classical logic.
module lc_prover.
accumulate  lc_math.
kind  seq type.
type --> (list form) -> (list form) -> seq.
infixl  -->     210.
type    >-2     lprf -> seq -> o.
infixl  >-2 200.
(initial A) >-2 (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(le_r A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (le A B) Delta Delta1,
  Q >-2 (Gamma --> ((lt A B)::(eq A B)::Delta1)).
(le_l A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (le A B) Gamma Gamma1,
  Q >-2 ((((lt A B) or (eq A B))::Gamma1) --> Delta).
(ge_r A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (ge A B) Delta Delta1,
  Q >-2 (Gamma --> ((lt B A)::(eq A B)::Delta1)).
(ge_l A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (ge A B) Gamma Gamma1,
  Q >-2 ((((lt B A) or (eq A B))::Gamma1) --> Delta).
(gt_l A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (gt A B) Gamma Gamma1,
  Q >-2 (((lt B A)::Gamma1) --> Delta).
(gt_r A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (gt A B) Delta Delta1,
  Q >-2 (Gamma --> ((lt B A)::Delta1)).
(imp_l A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Gamma Gamma1,
  Q2 >-2 ((B::Gamma1) --> Delta),
  Q1 >-2 (Gamma1 --> (A::Delta)).
(and_l A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1,
  Q >-2 ((A::B::Gamma1) --> Delta).
(imp_r A B Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Delta Delta1,
  Q >-2 ((A::Gamma) --> (B::Delta1)).
(exists_l  Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (exists A) Gamma Gamma1,
  pi T \ ((Q T) >-2 (((A T)::Gamma1) --> Delta)).
(forall_r Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  pi T \ ((Q T) >-2 (Gamma --> ((A T)::Delta1))).
(or_l A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A or B) Gamma Gamma1,
  Q1 >-2 ((B::Gamma1) --> Delta),
  Q2 >-2 ((A::Gamma1) --> Delta).
(and_r A B Q1 Q2) >-2 (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >-2 (Gamma --> (A::Delta1)),
  Q2 >-2 (Gamma --> (B::Delta1)).
(neg_r A Q ) >-2 (Gamma --> Delta) :-
  memb_and_rest (neg A) Delta Delta1,
  Q >-2 ((A::Gamma) --> Delta1).
(or_r A B Q ) >-2 (Gamma --> Delta) :-
  memb_and_rest (A or B) Delta Delta1,
  Q >-2 (Gamma --> (A::B::Delta1)).
(neg_l A Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (neg A) Gamma Gamma1,
  Q >-2 (Gamma1 --> (A::Delta)).
(exists_r T Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (exists A) Delta Delta1,
  Q >-2 (Gamma --> ((A T)::Delta1)).
(forall_l T Q) >-2 (Gamma --> Delta) :-
  memb_and_rest (forall A) Gamma Gamma1,
  Q >-2 (((A T)::Gamma1) --> Delta).
```

Figure B.7: lc_prover.mod

Appendix C

Implementation of A Proof Checker for FOL Formulas with Relational Expressions

We present a module lc_checker.mod to implement rules for a proof checker for FOL formulas with integers and relational expressions.

```
% This module contains a specification for a theorem prover/proof
% checker for first-order classical logic.
module lc_checker.
accumulate  basic.
accumulate  lprf.
kind seq type.
type --> (list form) -> (list form) -> seq.
type >-  lprf -> seq -> o.
infixl --> 210.
infixl >-  200.
(initial A) >- (Gamma --> Delta) :-
  memb A Gamma, memb A Delta.
(gt_r A B Q) >- (Gamma --> Delta) :-
  memb (gt A B) Delta ,
  Q >- (Gamma --> ((lt B A)::Delta)).
(gt_l A B Q) >- (Gamma --> Delta) :-
  memb (gt A B) Gamma,
  Q >- (((lt B A)::Gamma) --> Delta).
(le_r A B Q) >- (Gamma --> Delta) :-
  memb (le A B) Delta,
  Q >- (Gamma --> ((lt A B)::(eq A B)::Delta)).
(le_l A B Q) >- (Gamma --> Delta) :-
  memb (le A B) Gamma,
  Q >- ((((lt A B) or (eq A B))::Gamma) --> Delta).
(ge_r A B Q) >- (Gamma --> Delta) :-
  memb (ge A B) Delta,
  Q >- (Gamma --> ((lt B A)::(eq A B)::Delta)).
(ge_l A B Q) >- (Gamma --> Delta) :-
  memb (ge A B) Gamma ,
  Q >- ((((lt B A) or (eq A B))::Gamma) --> Delta).
(and_r A Q1 B Q2) >- (Gamma --> Delta) :-
  memb (A and B) Delta,
  Q1 >- (Gamma --> (A::Delta)),
  Q2 >- (Gamma --> (B::Delta)).
(or_r A B Q) >- (Gamma --> Delta) :-
  memb (A or B) Delta,
  Q >- (Gamma --> (A::B::Delta)).
(imp_r A B Q) >- (Gamma --> Delta) :-
  memb (A imp B) Delta,
  trace (Q >- ((A::Gamma) --> (B::Delta))).
(neg_r A Q) >- (Gamma --> Delta) :-
  memb (neg A) Delta,
  Q >- ((A::Gamma) --> Delta).
(exists_r T Q) >- (Gamma --> Delta) :-
  memb (exists A) Delta,
  Q >- (Gamma --> ((A T)::Delta)).
(forall_r Q) >- (Gamma --> Delta) :-
   memb (forall A) Delta,
   pi Y\ ((Q Y) >- (Gamma --> ((A Y)::Delta))).
(and_l A B Q) >- (Gamma --> Delta) :-
  memb (A and B) Gamma,
  Q >- ((A::B::Gamma) --> Delta).
(imp_l A Q1 B Q2) >- (Gamma --> Delta) :-
  memb (A imp B) Gamma,
  Q1 >- (Gamma --> (A::Delta)),
  Q2 >- ((B::Gamma) --> Delta).
(forall_l T Q) >- (Gamma --> Delta) :-
  memb (forall A) Gamma,
  Q >- (((A T)::Gamma) --> Delta).
(neg_l A Q) >- (Gamma --> Delta) :-
  memb (neg A) Gamma,
  Q >- (Gamma --> (A::Delta)).
(or_l A Q1 B Q2) >- (Gamma --> Delta) :-
  memb (A or B) Gamma,
  Q1 >- ((A::Gamma) --> Delta),
  Q2 >- ((B::Gamma) --> Delta).
(exists_l Q) >- (Gamma --> Delta) :-
  memb (exists A) Gamma,
  pi Y\ ((Q Y) >- (((A Y)::Gamma) --> Delta)).
```

```
(eq_sym A B Q) >- (Gamma --> Delta) :-
  memb (eq A B) Gamma,
  Q >- (((eq B A)::Gamma) --> Delta).
(eq_tran A B C Q) >- (Gamma --> Delta) :-
  memb (eq A B) Gamma,
  memb (eq B C) Gamma,
  Q >- (((eq A C)::Gamma) --> Delta).
(lt_tran A B C Q) >- (Gamma --> Delta) :-
  memb (lt A B) Gamma,
  memb (lt B C) Gamma,
  Q >- (((lt A C)::Gamma) --> Delta).
(eq_lt_tran A B B C Q) >- (Gamma --> Delta) :-
  memb (eq A B) Gamma,
  memb (lt B C) Gamma,
  Q >- (((lt A C)::Gamma) --> Delta).
(eq_lt_tran A B C A Q) >- (Gamma --> Delta) :-
  memb (eq A B) Gamma,
  memb (lt C A) Gamma,
  Q >- (((lt C B)::Gamma) --> Delta).
```

Figure C.1: lc_checker.mod

Bibliography

[1] Tel 1996. LATA switching systems generic requirements (LSSGR) FR-NWT-000064, 1992 edition. Feature requirements, including: SPCS capabilities and features, SR-504. 1996.

[2] Peter B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28:193–214, 1981.

[3] J. Blom, R. Bol, and L. Kempe. Automatic detection of feature interactions in temporal logic. In K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III*, pages 1–19. IOS Press, 1995.

[4] L. du Bousquet. Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods*. Springer Verlag, 1999.

[5] Amy P. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, Technical Report MS-CIS-89-53, August 1989.

[6] Amy P. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, pages 43–81, 1993.

[7] Amy P. Felty. Temporal logic theorem proving and its application to the feature interaction problem. In Enrico Giunchiglia and Fabio Massacci, editors, *Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, pages 19–28, June 2001. University of Siena, Technical Report DII 14/01.

[8] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automatic conflict detection. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, pages 179–192. IOS Press, 2000.

[9] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology*, 2003. To appear.

[10] Huet. G. The undecidability of unification in third order logic. In *Inf. Control 22*, 1973.

[11] A. Gammelgaard and J.E. Kristensen. Interaction detection, a logical approach. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 178–196. IOS Press, 1994.

[12] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[13] G.J. Holzmann and M.H.Smith. Automating software feature interaction. *Bell Labs Technical Journal 5*, 2000.

[14] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[16] William W. McCune. Otter 3.0 reference manual and guide. http://www-unix.mcs.anl.gov/AR/otter/, 1994.

[17] Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming.* MIT Press, 1988.

[18] Gopalan Nadathur and Dustin. J. Mitchell. System description: Teyjus — a compiler and abstract machine bases implementation of λProlog. In *The 16th International Conference on Automated Deduction*, pages 287–291. Springer-Verlag, July 1999.

[19] M. Plath and M. Ryan. Plug-and-play features. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 150–164. IOS Press, 1998.

[20] Philip Wickline. The terzo implementation of λProlog. http://www.cse.psu.edu/~dale/lProlog/terzo/, 1999.