Incorporating Specialized Theories into a General Purpose Theorem Prover

by

Anna Pogosyants

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Signature of Author .....
Department of Electrical Engineering and Computer Science
December 9 1994

Certified by .....
Stephen J. Garland
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Incorporating Specialized Theories into a General Purpose Theorem Prover

by

Anna Pogosyants

Submitted to the Department of Electrical Engineering and Computer Science
on December 9 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis focuses on automatic theorem proving. In this area there is usually a tradeoff between the "generality" of a prover and the amount of user-guidance it requires to find a proof. This thesis demonstrates that the amount of guidance needed can be reduced by employing reasoning procedures for specialized theories. It presents an enhancement to the Larch Prover (LP) with specialized theories of equality, propositional logic and linear arithmetic. An application of enhanced LP to the verification of a concurrent algorithm is described.

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Acknowledgements

I would like to thank all the people who helped me with the work of this thesis. First of all I thank Stephen Garland, who contributed many useful suggestions and helped tremendously to put this draft together. I want to thank John Guttag for giving constantly good advice and encouragement. I thank Yang Meng Tan for proof reading earlier drafts of this thesis.

I also want to thank my officemates, David Evans and Yang Meng Tan, for providing a pleasant atmosphere and many interesting discussions. I want to thank members of the TDS group for suggesting interesting examples for this thesis.

Finally, I want to thank my family for constant encouragement and support.

Contents

1 Introduction **13**

1.1 Goal and approach 14

1.1.1 General picture 14

1.1.2 Goal 15

1.1.3 Approach 15

1.2 Background 16

1.2.1 Propositional logic 16

1.2.2 Theory of equality with uninterpreted function symbols 16

1.2.3 Linear arithmetic 17

1.2.4 The refutation principle 18

1.2.5 Nelson's method for reasoning about a combination of theories . . . 19

1.2.6 An example of Nelson's method 20

1.2.7 The Larch Prover (LP) 20

1.3 Related work 21

1.3.1 The Stanford Pascal Verifier 21

1.3.2 The Boyer-Moore prover 21

1.3.3 IMPS 22

1.3.4 EHDM 22

1.3.5 PVS 23

1.3.6 Summary 23

2 Description of the work **25**

2.1 Using built-in specialized theories in the Larch Prover 25

2.2 Overview of the design of the specialized theories module 28

2.2.1 Ground_system and satisfy 29

2.2.2 TheoryE 32

2.2.3 TheoryB 32

2.2.4 TheoryR 36

2.3 Important details of the work 39

2.3.1 Communication between specialized theories 39

2.3.2 Integration of specialized theories and the Larch Prover 41

2.3.3 Communication of specialized theories with the user 43

3 Experiment with the enhanced Larch Prover **47**

3.1 Background 48

3.1.1 Input/Output Automata 48

3.1.2 Invariants and simulations 51

3.2 Original sample proof 52

3.2.1 Problem description 52

3.2.2 Axiomatization 54

3.2.3 LP proof 56

3.3 Simplified sample proof 57

3.3.1 Changes in the specifications 58

3.3.2 New proof is 30 % shorter 61

3.3.3 New proof is three times faster 62

4 Conclusion **65**

4.1 Possible enhancements to the existing tool 65

4.1.1 Possible improvements to the linear arithmetic procedure 65

4.1.2 Addition of new theories 66

4.2 Possible directions of future research 69

4.2.1 Communication between specialized theories and rewriting 69

4.2.2 Using specialized theories in explorative proof systems 72

4.2.3 Useful variable-free instances of axioms with variables 75

List of Figures

2-1 A sample proof using specialized theories 27

2-2 Proof by specialized theories failed: the conjecture is wrong 28

2-3 Proof by specialized theories failed: more variable-free instances needed . . 29

2-4 Organization of the specialized theories module 29

2-5 Informal specification of the satisfy procedure 30

2-6 Organization of TheoryB module 33

2-7 Unit propagation 33

2-8 Simplification rules for clauses 35

2-9 Organization of the TheoryR module 37

2-10 Multiple instances of rational linear arithmetic 42

2-11 Simplification of an example assignment 45

3-1 A counting automaton $C(k, c_1, c_2)$ 52

3-2 A report automaton $R(a_1, a_2)$ 53

3-3 LSL specification of the counting automaton C 55

3-4 LSL specification of natural numbers 55

3-5 LSL specification of time bounds 56

3-6 LSL specification for some properties of real numbers 57

3-7 Proof obligation of the third property of simulation 58

3-8 The old LP proof of the third property of simulation 59

3-9 Lemmas about naturals that are used in enhanced LP proof 60

3-10 The specification of reals used in the new proof 60

3-11 The new LP proof of the third property of simulation 61

3-12 Why proofs are getting shorter 62

3-13 New proof is faster 63

4-1 Axiomatization of partial orders 67

4-2 Axiomatization of theory of lists 68

4-3 Application of specialized theories in rewriting 70

4-4 Using specialized theories in conditional rewriting 71

Chapter 1

Introduction

This work is focused on automatic theorem proving. Theorem provers are tools that allow people to use computers for constructing and checking formal proofs.

Two kinds of provers can be distinguished: general purpose provers that are used to reason about general theories (e.g., first order logic) and special purpose provers that can perform reasoning only about restricted (*specialized*) theories (e.g., boolean propositional logic). For some specialized theories there are reasoning procedures (possibly incomplete) that perform reasoning about them. I will call such procedures *specialized* procedures.

This work describes an experiment of incorporating specialized procedures for some specialized theories into a general purpose prover. The incorporation improved the general purpose prover in two ways: it became easier to construct proofs, in particular because the amount of necessary user guidance was greatly reduced, and some existing proofs ran faster.

1.1 Goal and approach

1.1.1 General picture

Ideally, when one thinks of a theorem prover one imagines a black box that automatically produces a proof given any true conjecture. Such provers can be called *general and automatic*, since they attempt to prove any conjecture automatically. The problem of finding a proof for any conjecture automatically is computationally very hard. For some general domains, it is undecidable.

There are some ways, however, to improve the situation. One possibility is to restrict the domain of the prover to something that is efficiently decidable. Then, to check whether a conjecture is a theorem, it suffices to invoke a specialized procedure for this domain. This leads us to *specialized and automatic* provers, since they only reason about restricted domains, but do that automatically. Some examples are:

- Linear arithmetic — Reasoning about linear inequalities can be performed by employing linear programming algorithms.
- Theory of transitivity — Reasoning about transitive relations can be done by computing transitive closure.

There are many other domains that can be reasoned about automatically. However, generality is lost in this reasoning. We can no longer reason about all conjectures, only about restricted classes of them.

An alternative way of improving the search for a proof is to let the user guide the search. Indeed, the user often has some intuition about how to conduct the proof, and such intuition is difficult to automate. For example, the user can prove useful lemmas or select appropriate proof methods. Using this approach we retain generality, but our reasoning is not completely automatic any more. This class of provers is best called *general and guided.*

1.1.2 Goal

Even if we allow for some guidance in *general and guided* provers, we still want to minimize the amount of user guidance that is required. We adopt the point of view that guidance should be used ideally only to highlight the key steps of the proof. It is undesirable for low-level details of a proof to require a lot of guidance. The focus of this research is to reduce the amount of guidance by enhancing *general and guided* provers to handle more low-level details automatically.

1.1.3 Approach

As mentioned before, reasoning about certain domains can be performed fully automatically by specialized procedures. It often happens that problems from these domains occur as low-level subgoals of general conjectures. If a general and guided prover is used to work on a general conjecture with such subproblems, doing the specialized subproblems by the general techniques usually requires a lot of guidance.

This thesis describes an attempt to reduce the guidance needed to solve specialized subproblems by combining general and guided provers with specialized and automatic ones. This combination incorporates procedures used in specialized and automatic provers into a general and guided prover.

I have implemented a *specialized theories module* that performs reasoning about combinations of theories. The module uses the combining method described by Greg Nelson [NO80]. It combines the following specialized theories:

- boolean propositional logic;
- theory of equality with uninterpreted function symbols;
- linear arithmetic.

I incorporated the specialized theories module into the Larch Prover (LP) [GG89], which is a general and guided theorem prover.

I applied the enhanced LP to a proof that verifies properties of a concurrent algorithm [Söy94]. The use of specialized theories reduced the amount of guidance required for the proof by 30% and reduced the time LP spends checking the proof by factor of three. The details of the experiment with enhanced LP are described in the Chapter 3.

1.2 Background

In the rest of this section I give the necessary information about algorithms used in the specialized theories module I have implemented and also a short description of the Larch Prover.

1.2.1 Propositional logic

The specialized procedure for propositional logic used in my specialized prover can be thought of as a satisfiability procedure that performs smart truth table checking. To construct the truth table, the procedure performs case splits by assigning propositional variables to *true* or *false*, and by backtracking if some assignment is found impossible.

I used some heuristics for the satisfiability search from T. Larrabee's work on test pattern generation [Lar90].

1.2.2 Theory of equality with uninterpreted function symbols

The theory of equality with uninterpreted function symbols defines a predicate "=" on terms as being an equivalence relation that is also a congruence. A sample theorem of this theory is:

$$(x = y) \Rightarrow (f(x) = f(y))$$

where f is an uninterpreted function symbol and x, y are variables. The specialized procedure for reasoning about this theory is based on the congruence closure algorithm suggested

by Greg Nelson. The worst case running time of the algorithm is $O(n^2)$, where n is the number of symbols in the formula. The average case time is $O(n * log(n))$ [NO80].

1.2.3 Linear arithmetic

Linear arithmetic is the theory of a set of numbers under addition, multiplication by a constant element of the set, and ordering operations.

One could consider instances of linear arithmetic for different sets of numbers. In this work we consider the following three:

- linear arithmetic on rational numbers;
- linear arithmetic on integer numbers;
- linear arithmetic on natural numbers (nonnegative integers).

A sample rational linear arithmetic theorem is:

$$(x + x < y) \wedge (y < z) \Rightarrow (2 * x < z),$$

where x, y and z are universally quantified variables. This theorem is true for rationals, integers and naturals. If a universally quantified statment of integer (or natural) linear arithmetic is true under the theory of rational linear arithmetic then it is a theorem of integer (or natural) linear arithmetic as well. The converse is not true. For instance

$$(x < y) \Rightarrow (x \leq y - 1)$$

is true for integers but not for rationals.

The specialized procedure for rational linear arithmetic is based on the simplex method of linear programming [NO80]. The worst case running time of the simplex algorithm is exponential, but its average case time is linear [Sch86].

There are no efficient complete specialized procedures for integer or natural linear arithmetic, since the corresponding linear programming problems are NP-complete [Sch86]. In

my work, incomplete specialized procedures for these theories will be used. These procedures are based on the simplex method enhanced with some additional inferences (see Chapter 2 for more details).

1.2.4 The refutation principle

A theory T consists of a set of function symbols and a set of axioms A that constrain the interpretation of the function symbols.

We say that a formula F *follows* from a set of facts G modulo T (written as $A \cup G \vdash F$) if for every interpretation in which all formulas from $A \cup G$ are true, F is also true. We say that a set of formulas is *satisfiable*, if there exists an interpretation in which all formulas from the set are true. A set of formulas S is said to be *satisfiable modulo* T if $A \cup S$ is satisfiable.

Reasoning about a theory T involves determining whether a formula F follows from a set of facts G, modulo the theory T. In other words, we want to know whether

$$A \cup G \vdash F.$$

This problem can be approached by doing *refutation*:

$A \cup G \vdash F$ is true if and only if $A \cup G \cup \{\neg F\}$ is unsatisfiable.

Thus, the problem of determining consequence with respect to a theory is reduced to the satisfiability problem modulo the theory.

If there exists a procedure P for a theory T such that $P(G, F) = \mathit{true}$ if $A \cup G \vdash F$, then P is called a *complete* procedure for T. If a procedure P is such that $A \cup G \vdash F$ if $P(G, F) = \mathit{true}$,then P is called a *sound* procedure for T.

In practice, sometimes complete procedures for a theory are too expensive or do not exist, but we can use efficient and useful incomplete procedure for the theory instead. We always require our procedures to be sound.

The refutation approach implements reasoning about a theory T by constructing a satis-

fiability procedure for T. To guarantee soundness we require the following property to be true for a satisfiability procedure S for T:

If $S(G, \neg F)$ returns *unsatisfiable* this answer must be correct.

This property is essential, because the answer *unsatisfiable* will be interpreted to mean that F follows from G under T, and we want to guarantee soundness. Since we are willing to accept incomplete reasoning, $S(G, \neg F)$ may return *satisfiable* when $A \cup G \cup \{\neg F\}$ is in fact unsatisfiable. Note that the satisfiability procedure that always returns *satisfiable* will guarantee sound and incomplete reasoning. Such a procedure, however, is not very useful, since it will not prove any formula.

1.2.5 Nelson's method for reasoning about a combination of theories

A *combination* of theories T_1, ..., T_n is a theory T such that

- the set F of functions of T is $F_1 \cup ... \cup F_n$, where F_i is the set of functions of T_i;
- the set A of axioms of T is $A_1 \cup ... \cup A_n$, where A_i is the set of axioms of T_i.

In my specialized theories module, reasoning about a combination of theories is performed using a method suggested by Greg Nelson [Nel80], which allows us to reason about combinations of theories if we can reason about each theory individually. Through the use of refutation, the problem of determining consequence with respect to a combination of theories is reduced to the satisfiability problem modulo the combination of theories.

Nelson developed a method of constructing a satisfiability procedure for a combination of theories given a satisfiability procedure for each individual theory. The method only works for variable-free satisfiability problems (variable-free refers only to formulas to be checked, but axioms of the theories may contain variables). The key idea of the method is to combine individual procedures by propagating the equalities entailed by one procedure to all other procedures. Nelson showed that if the combined theories do not share function symbols, and if the procedures for individual theories are sound (the answer *unsatisfiable* is correct), then

the procedure for the combination is sound as well. Nelson also established some technical conditions on theories, under which combining complete procedures (ones that guarantee that the answer *satisfiable* is correct) results in a complete procedure for the combination of theories.

1.2.6 An example of Nelson's method

The following example shows how Nelson's method works. Given the combination of linear arithmetic and the theory of equality with uninterpreted function symbols, suppose we want to determine whether

$$(a \leq b \wedge b \leq c \wedge c \leq a) \vdash f(a) = f(b)$$

is true.

According to Nelson's approach we negate the goal and determine whether

$$a \leq b \wedge b \leq c \wedge c \leq a \wedge f(a) \neq f(b)$$

is unsatisfiable.

Under linear arithmetic the conjunction

$$a \leq b \wedge b \leq c \wedge c \leq a$$

entails an equality $a = b$. When this equality is propagated to the theory of equality, this theory entails an equality $f(a) = f(b)$. The discovery of $f(a) = f(b)$ contradicts $f(a) \neq f(b)$ Therefore the whole conjunction is unsatisfiable, and the original conjecture is proved.

1.2.7 The Larch Prover (LP)

The Larch Prover is a general purpose theorem prover for first-order logic [GG91]. LP is interactive. It does not attempt to do complicated proof steps (like generating unobvious lemmas) automatically. Instead, an LP user must guide the proof process by issuing explicit commands. However, LP performs some useful steps, such as simplifying conjectures,

automatically. The best way to think about LP is as an interactive proof debugger. A very important feature of LP is that it provides useful feedback when a proof fails.

The main automatic inference mechanism of LP is equational term rewriting. LP also provides user-controlled natural deduction rules of inference.

1.3 Related work

In this section I describe other provers that deal with specialized theories and the specialized procedures used in these provers.

1.3.1 The Stanford Pascal Verifier

The Stanford Pascal Verifier [VG79] is a verification system designed to reason about Pascal programs. It generates verification conditions for a program by computing the weakest preconditions of the desired postconditions of the program. The system also provides proof facilities for checking the correctness of the program.

Greg Nelson's algorithms were originally used in the Stanford Pascal Verifier. The important difference between my research and the work done in the Pascal Verifier is that the prover in the Pascal Verifier is specialized, and therefore the work on the Pascal Verifier is focused on building a good specialized prover, rather than on combining specialized and general provers.

1.3.2 The Boyer-Moore prover

The Boyer-Moore prover [BM79, BM88a] is a general purpose theorem prover designed to prove theorems from mathematics using induction. The language of the prover is a version of pure LISP. The main simplification mechanism of the Boyer-Moore prover is (nonequational) term rewriting. The prover attempts to construct proofs automatically using heuristics to invent useful lemmas, to pick the right induction schema, etc.

The Boyer-Moore prover has a built-in procedure for linear arithmetic on natural numbers [BM88b]. The procedure is based on a variable elimination technique (a generalization of Gaussian elimination for the case of linear inequalities), which is complete for rational numbers but is incomplete for naturals and integers.

The main differences between the work done in the Boyer-Moore prover and my research are the following:

- The Boyer-Moore prover does not have a general framework for specialized theories, whereas I enhanced the Larch Prover with such a framework.
- The Boyer-Moore prover does not expect the user to give explicit commands, and therefore it does not care much about giving the user hints of what to do next in the case when prover is stuck. In my work, in cases were specialized theories fail to perform the proof, the satisfying assignment found by the unsuccessful refutation is returned. See Section 2.1 for more details.

1.3.3 IMPS

IMPS [FGT93] is an interactive mathematical proof system. It was designed as a general purpose tool for formulating mathematical concepts and reasoning about them. The IMPS logic is based on type theory and allows for reasoning about higher order functions. The main simplification mechanism used in reasoning is (nonequational) term rewriting. IMPS also has a built-in mechanism for algebraic simplification of polynomials.

IMPS has a built-in specialized procedure for rational linear arithmetic. Like the arithmetic procedure of Boyer-Moore, the procedure in IMPS uses the variable elimination technique.

1.3.4 EHDM

The Enhanced Hierarchical Development Methodology (EHDM) is an interactive system for constructing and analyzing formal specifications and programs.

EHDM has a framework for specialized theories, which are combined by R. Shostack's method [Sho82]. Like Nelson's method, Shostack's method allows reasoning about combinations of theories, provided that reasoning can be performed about individual theories. Unlike Nelson's method, it does not combine the procedures for individual theories, but constructs a procedure for the combination of the theories based on a single uniform canonization procedure. The code for the canonization procedure is efficient; however, it is harder to add new theories to Shostack's system than to Nelson's.

1.3.5 PVS

The Prototype Verification System (PVS) [Sha93] was developed for constructing formal specifications and verifying them. It is based on higher order logic. PVS is interactive and uses (nonequational) term rewriting as an inference mechanism. There are fewer rewriting facilities in PVS than in the Larch Prover: for example, PVS does not maintain a termination ordering and it does not have a critical pairing mechanism.

PVS uses Greg Nelson's method to combine specialized theories. It uses a different procedure for linear arithmetic than the one suggested by Nelson. The procedure is based on the *Sup-Inf* algorithm that was first proposed by W. Bledsoe and later analyzed and improved by R. Shostack [Sho77]. The main idea of the algorithm is reflected in its name: for each variable x, it computes *Sup(x)* and *Inf(x)* with respect to the set of constraints of the problem. If for some x the interval [*Inf(x), Sup(x)*] is void, then the original problem is unsatisfiable. The Sup-Inf algorithm is complete for rational numbers. The algorithm can also be adapted for reasoning about integer and natural linear arithmetic. Although the adapted algorithm is incomplete, it is useful in practice. The Sup-Inf algorithm also can be modified to handle arithmetic with infinity.

1.3.6 Summary

The work described in this thesis incorporates specialized reasoning into a general purpose prover (LP), which makes it different from specialized verification systems, such as the

Stanford Pascal Verifier. It also provides an extensible framework for specialized theories and some help in cases where proofs by specialized theories get stuck, which distinguishes it from the work done in some general provers, such as the Boyer-Moore prover. This thesis pays special attention to extensibility (possibility of adding/deleting theories) of the framework, which makes it different from work done in provers that use less flexible methods for creating the framework of specialized theories (like EHDM).

Possible future research may be related to communication between the rewriting machinery of LP and the specialized theories (see Chapter 4). In this case the fact that LP provides a lot of rewriting facilities makes future research for this work different from the future research for the work done in provers with fewer rewriting facilities (PVS, IMPS).

Chapter 2

Description of the work

In this chapter I describe the important aspects of my work, which consists of the following parts:

- Implementation of the specialized theories module for propositional logic, the theory of equality, and linear arithmetic using Nelson's methods.
- Incorporation of the specialized theories module into the Larch Prover (LP).

I first describe the general usage of specialized theories provided by the module in LP and then I outline the design of the specialized theories module. In the Section 2.3 I focus on some important details of the work.

2.1 Using built-in specialized theories in the Larch Prover

The built-in specialized theories are used in the Larch Prover to conclude proofs of prenex universal or variable-free subgoals. The specialized theories module is called on a conjecture upon user command **zap**. The module also keeps track of all variable-free facts in the system. When the module is called on a conjecture, it determines whether the conjecture follows from the variable-free facts by performing refutation (see Chapter 1, Section 1.2.4).

Specialized theories provide a powerful reasoning mechanism; they give most benefit when used in a "correct" way. The methodology of using specialized procedures comes from the following restrictions:

- Specialized theories can conclude only proofs of subgoals that are within their domain.
- Specialized theories work only with variable-free axioms and variable-free or universally quantified conjectures.

The first part of using specialized theories is to prepare suitable input for them. This can be done by using general techniques supported by LP. The inference techniques of LP can be classified into

- Backward inference – these techniques produce subgoals from the conjecture.
- Forward inference – these techniques produce consequences from the axioms.

LP's backward inference mechanisms can be used to reduce a general conjecture to subgoals, some of which are within the domain of specialized theories. Forward inference can be used to produce necessary variable-free instances of axioms with variables.

After a conjecture is reduced to specialized subgoals, and the necessary variable-free instances of axioms are obtained, the built-in specialized theories can be applied to attempt to complete the proofs of the subgoals.

Figure 2-1 illustrates the use of specialized procedures in LP. The first two lines define the sort **Q** and associate the theory of rational numbers with it (see Section 2.3.2). The next two lines declare an operator f, a constant a, and variables x and y. The **assert** command introduces an axiom that states that f ranges over numbers strictly greater than -3. Then we attempt to prove the conjecture $\exists y(f(a) + y > 0)$. The specialized theories module is not called on the conjecture, since latter includes an existential quantifier. The **resume** command performs a backward inference, determining that proving the subgoal $f(a) + 4 > 0$ is enough to prove the conjecture. The subgoal is also variable-free, so the specialized theories module can be applied to it. The specialized theories, however, cannot

```
declare sort Q
include Rational(Q)
declare operator a:→ Q, f: Q→Q
declare variable x, y: Q
assert f(x) > -3
prove ∃ y (f(a) + y > 0)
resume by specializing y to 4
instantiate x by a in *
zap
```

Figure 2-1: A sample proof using specialized theories

prove the subgoal because no information about $f(a)$ is known to the specialized theories module. The **instantiate** command performs the forward inference by producing the variable-free consequence $f(a) > -3$, which is passed to the specialized theories module. After this instance is computed, the command **zap** invokes the specialized theories module on the subgoal and the proof is finished successfully. Indeed,

$(f(a) > -3) \vdash (f(a) + 4 > 0)$

under the theory of arithmetic.

Figure 2-1 shows a successful application of specialized theories. The proof was finished by an explicit call to the specialized theories module. It could happen, however, that specialized theories are not able to finish the proof of a subgoal. The methodology of interactive theorem proving requires reasonable feedback from the prover in such a proof failure.

In cases where specialized theories do not succeed, **zap** displays a possible satisfying assignment of the negation of the conjecture and the variable-free facts of the system to the user (the conjunction is satisfiable since the refutation did not succeed). By looking at the assignment a user may be able to decide why the proof by specialized theories failed. Here are some examples and analyses of proof failures:

- The subgoal does not follow from the axioms in the system. In this case, by examining the satisfying assignment, the user can either change the subgoal or add more axioms. Figure 2-2 illustrates the situation. The conjecture shown in Figure 2-2 is not always

```
declare sort Q
include Rational(Q)
declare variable x, y: Q
prove x + y > x
zap
A possible satisfying assignment of the variable-free facts in the system
and the negation of the current conjecture is:
Assignment: y ≤ 0
```

Figure 2-2: Proof by specialized theories failed: the conjecture is wrong

true for any rational numbers x, y, but only if $y > 0$. By looking at the assignment the user can find a way to weaken the conjecture, e.g., to:

$(y > 0) \Rightarrow (x + y > x)$

which is true for all rational x, y.

- The subgoal follows from the axioms, but there are not enough variable-free instances of the axioms with variables for the specialized theories module to finish the proof (since the specialized theories only work with variable-free axioms). In Figure 2-3 the only axiom of the example states that the function f is pointwise greater than g. This axiom contains a single variable x. When the user tries to prove the conjecture, the proof fails, and the assignment $f(b) < g(b)$ is returned. The user can observe that this assignment contradicts the axiom and add the instance of the axiom that is needed to prove the conjecture:

  $f(b) \geq g(b)$.

  The instance can be introduced by instantiating the axiom.

2.2 Overview of the design of the specialized theories module

In this section I give an overview of the design of the specialized theories module that I implemented. The general principle of the design is to let each theory be described by

```
declare sort Q
include Rational(Q)
declare variable x: Q
declare operator f, g : Q → Q, a, b:→Q
assert f(x) ≥ g(x)
prove (f(b) + a) ≥ (g(b) + a)
zap
A possible satisfying assignment of the variable-free facts in the system
and the negation of the current conjecture is:
Assignment: f(b) < g(b)
```

Figure 2-3: Proof by specialized theories failed: more variable-free instances needed

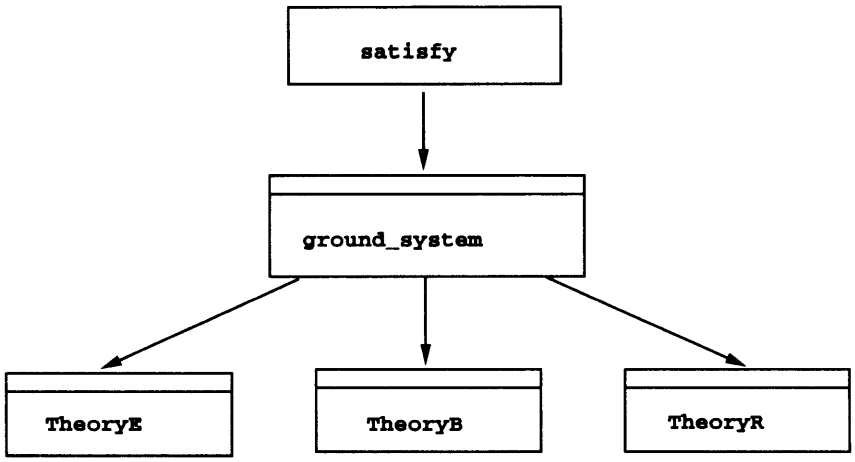

Figure 2-4: Organization of the specialized theories module

a separate submodule, and to organize communication among them around a top-level module. Figure 2-4 shows the organization of the specialized theories module.

2.2.1 Ground_system and satisfy

The top-level data module is named ground_system. Ground_system maintains a conjunction of ground (variable-free) formulas. It also maintains a set of theories for reasoning about the formulas. Theories are implemented as separate modules. There are three modules implementing reasoning about different specialized theories. Module TheoryE implements the theory of equality, TheoryB implements the theory of propositional logic, and TheoryR im-

```
%Requires: t is variable free
%Effects:  If gsys.conjunction ∧ t is satisfiable wrt to gsys.theories
%              then returns a model m of gsys.conjunction ∧ t
%              else signals "unsatisfiable"
satisfy = proc (gsys: ground_system, t: term) returns (model)
                                      signals (unsatisfiable)
```

Figure 2-5: Informal specification of the satisfy procedure

plements linear arithmetic. The theories communicate with the ground_system via *literals*. A literal is either an atomic boolean formula or a negation of one. Theories can accept literals from the ground_system or pass literals to it. A literal is called *relevant* to a theory if the top-level operator of its corresponding atomic formula is interpreted in the theory (e.g., the literal $x < 0$ is *relevant* to TheoryR).

Satisfy is a top-level module that performs satisfiability checking. The informal specification for the satisfy procedure is shown in the Figure 2-5.

The notations **gsys.conjunction** and **gsys.theories** are used to denote the conjunction of formulas and the set of theories maintained by the ground_system, **gsys**. The intended use of satisfy assumes that **gsys** contains all variable-free facts known to the prover, and **t** is the negation of a conjecture. The **Requires** statement of the specification says that **t** is to be variable-free. Satisfy may work incorrectly if a term with variables is passed as an argument. The rest of LP guarantees that **Requires** clause is not violated. Satisfy performs the satisfiability test needed in the refutation.

Satisfy first adds **t** to the conjunction of formulas, maintained by ground_system, and tries to satisfy the conjunction. It does the satisfiability by performing case splits. Ground_system is designed to choose a literal that belongs to some formula in the conjunction. Satisfy picks the literal l from the ground_system and *splits* on it: it first assigns l to *true*, but if later this assignment is discovered to be inconsistent, it reassigns l to *false*. Ground_system checks for inconsistency by employing modules implementing theories (each module implementing

a theory is designed to detect whether the conjunction of literals relevant to the theory is inconsistent). If all possible assignments of l are found to be inconsistent, satisfy backtracks by changing the previous assignment to its opposite (or backtracking further, if the opposite was already tried). After backtracking is done, satisfy gets a literal from the ground_system again. This process continues until either all literals involved in ground_system are assigned (which means that a model is found), or all previously made assignments are undone and some literal l still cannot be assigned (which means that no model exists). Satisfy takes exponential time in the worst case.

I use the *depth-one plunge* heuristic to improve the performance. The depth-one plunge heuristic improves the performance by the following means:

- It decreases the number of non-assigned literals in ground_system.
- It controls the ordering of case splits.

To decrease the number of non-assigned literals, depth-one plunge performs *forced* assignments. An assignment of literal l to value v (*true* or *false*) is said to be *forced* if the opposite assignment of l to $\neg v$ (*false* or *true*) is detected to be inconsistent (by some incomplete, but cheap check). Depth-one plunge checks all literals and performs corresponding forced assignments. While doing quick checks, depth-one plunge counts the number of literals that will remain unassigned if l is assigned to a particular value (*true* or *false*) This information is later used to control the ordering of case splits: ground_system suggests a case split on literal l such that assigning l to *true* results in minimum unassigned literals (minimum is taken over all literals from the conjunction of formulas in ground_system).

Depth-one plunge was originally used as a heuristic for boolean satisfiability. The reason I used it outside TheoryB is to make other theories (TheoryR and TheoryE) participate in the detection of inconsistencies.

In the rest of this section, I describe the main submodules TheoryE, TheoryB, and TheoryR and some low-level modules that they depend on.

2.2.2 TheoryE

TheoryE maintains a conjunction of equality and disequality literals between terms. The conjunction is closed under term congruence and equivalence. The specialized theories module maintains the invariant that TheoryE is consistent, by undoing steps that introduce inconsistency.

TheoryE is incremental and backtrackable: it allows the addition of new literals and can undo these additions. When a new literal is added to TheoryE it does the following:

- If appending the new literal to the conjunction introduces an inconsistency under the theory of equality, TheoryE reports a contradiction.
- If appending the new literal to the conjunction results in a consistent (under the theory of equality) conjunction, then TheoryE outputs a set E of equality literals such that E is equivalent to the set of all literals that are implied by the extended conjunction under the theory of equality, but were not implied by the unextended one.

The implementation of TheoryE relies on the egraph abstraction. Egraph is a congruence closure graph, that is, it is a subterm graph with a binary relation defined on its nodes. The relation is closed under equivalence and term congruence.

2.2.3 TheoryB

TheoryB maintains a conjunction of clauses. It is intended to be the clausal form of formulas from ground_system. The formulas are converted to clauses by a linear time conversion which preserves satisfiability (the produced set of clauses is satisfiable iff the original set of formulas is) and may introduce new propositional variables.

The organization of the module TheoryB is shown in Figure 2-6.

Unit clauses (further referred as units) are maintained separately in TheoryB by module *units*. Other clauses (binary and longer) are maintained by module *clauses*. The specialized theories module maintains the invariant that the set of unit clauses of TheoryB is consistent.

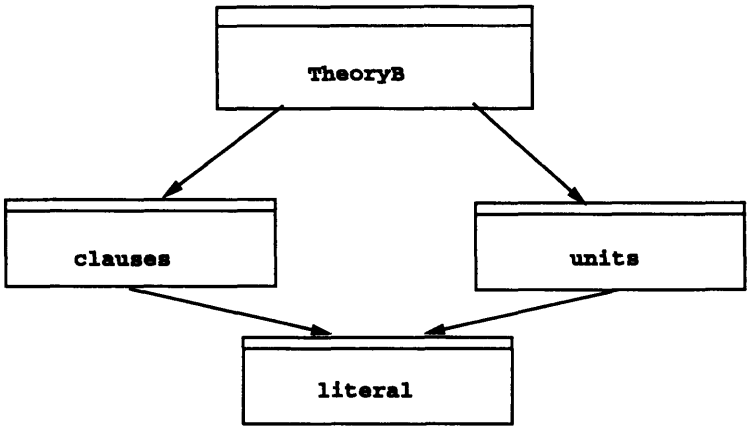

Figure 2-6: Organization of TheoryB module

Transform $p \wedge F(p) \wedge ...$ into $p \wedge F(true) \wedge ...$

Transform $\neg p \wedge F(p) \wedge ...$ into $\neg p \wedge F(false) \wedge ...$

Figure 2-7: Unit propagation

Like TheoryE, TheoryB is incremental and backtrackable. Both clauses and literals of TheoryB can be extended. New clauses can appear in TheoryB only from the conversion of some formulas of ground_system into clausal form. Units can also come from case splitting. Indeed, assigning a literal to *true* or *false* is equivalent to just adding this literal or its negation as a unit clause to the conjunction maintained by TheoryB. These extensions can be undone.

TheoryB uses the *unit propagation* heuristic to maintain the conjunction of clauses. I first describe the heuristic and then show how it is implemented in TheoryB. Figure 2-7 shows the main inferences performed by the heuristic.

If unit p is a member of a conjunction, it must be assigned to *true* in any possible model. Therefore it is possible to substitute *true* for p in the rest of the conjunction. In the case when unit $\neg p$ is a member of the conjunction, p must be assigned to *false*, and p can be replaced by *false* in the rest of the conjunction. Usually when unit propagation is used,

there are some assumptions made about formulas conjunctions, and some further simplifications are possible (e.g., in TheoryB all formulas are clauses). After these simplifications are made, more units can appear. These have to be propagated as well. The process stops where no more units can be obtained. It is possible for an inconsistency to be discovered by unit propagation, if some propositional variable must be assigned to both *true* and *false*. Consider the example:

$a \wedge (\neg a \vee b) \wedge \neg b.$

This formula is inconsistent, and the inconsistency can be detected by unit propagation in the following way:

- Eliminate unit a from the conjunction, and rewrite a to *true* in the rest of the conjunction. The new conjunction is:

  $(false \vee b) \wedge \neg b.$

- Simplify the conjunction by omitting *false* in the first conjunct to obtain:

  $b \wedge \neg b,$

  which is detectably inconsistent, since it means that b must be assigned to both *true* and *false* by unit propagation.

Unit propagation is sound in the sense that if an inconsistency is discovered, then the conjunction is indeed inconsistent. However, unit propagation is not complete, as next example shows:

$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b).$

This formula is obviously inconsistent (none of the four possible interpretations models it), but unit propagation will not detect this fact, since there are no units to work with.

In TheoryB unit propagation is implemented by maintaining the invariant that all units are propagated and all clauses are simplified with respect to the simplification rules shown in Figure 2-8.

The first rule says that the unit *true* can simply be eliminated from a conjunction. The

$$true \wedge F \rightarrow F$$

$$false \wedge F \rightarrow false$$

$$true \vee F \rightarrow true$$

$$false \vee F \rightarrow F$$

Figure 2-8: Simplification rules for clauses

second rule states that if a conjunction has a conjunct *false* the conjunction is equivalent to *false*. The third rule states that if one of the disjuncts of a clause is *true* the whole clause is equivalent to *true*. The last rule says that if one of the disjuncts of a clause is *false*, then this disjunct can be eliminated from the clause.

If some new clauses are added to TheoryB, the following steps are done to preserve the invariant:

- If a unit clause is added, it is propagated.
- If a non-unit clause is added, then the following steps are carried out:
  1. If a literal from the clause is assigned to *true* (or *false*) by some previously done unit propagation in TheoryB, this literal is rewritten to *true* (*false*).
  2. The simplification rules for clauses are applied.
  3. If a simplification of a clause results in a unit clause, the unit clause is propagated.
- If an inconsistency is discovered by the above steps, TheoryB reports a contradiction. If no inconsistency is discovered, TheoryB outputs all new unit clauses discovered by unit propagation.

2.2.4 TheoryR

Module TheoryR implements the theory of linear arithmetic. It maintains a conjunction of linear equations and inequalities. The specialized theories module maintains the invariant that the conjunction is consistent under linear arithmetic.

After analyzing some verification proofs that involved arithmetic reasoning, I decided that we ought to have three kinds of linear arithmetic:

- rational linear arithmetic;
- integer linear arithmetic;
- linear arithmetic on natural numbers with 0.

TheoryR checks the consistency of the conjunction by employing linear programming. The checks for rational equations and inequalities are complete. The checks for naturals and integers are incomplete. This is mainly done for efficiency: integer and natural linear programming problems are *NP-complete.*

TheoryR uses the simplex algorithm for all three kinds of arithmetic. This gives a sound and complete inconsistency check for rational arithmetic literals, and sound, but an incomplete check for naturals and integers (if a set of linear arithmetic literals is not satisfiable in the rationals, it is not satisfiable in the integers and naturals either, but if it is satisfiable in the rationals, this does not guarantee satisfiability in the integers and naturals).

For integer and natural literals some additional steps are performed to gain more deductive power:

- For each natural sorted term n from TheoryR, the literal $n \geq 0$ is added to the conjunction of TheoryR.
- If literal $i < j$ (or $i > j$) is in the conjunction of TheoryR and i, j are either integer or natural sorted terms, then the literal $i \leq j - 1$ (or $i \geq j - 1$) is added to the conjunction.

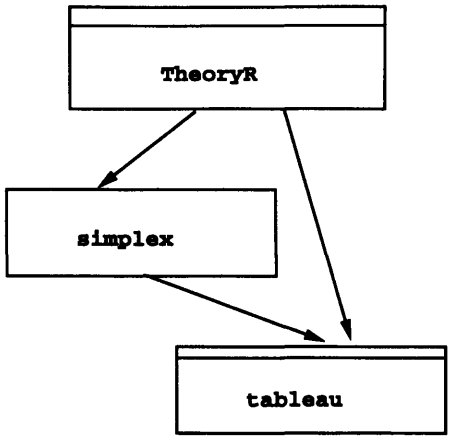

Figure 2-9: Organization of the TheoryR module

Like the other modules implementing theories, TheoryR is incremental and backtrackable. If a literal (linear equality or inequality) is added to the conjunction, TheoryR checks whether the new conjunction is satisfiable. It does the check by extending the conjunction according to the inferences for integers and naturals mentioned above, if those are applicable, and then running the simplex method to determine the satisfiability of the extended conjunction in the rationals. If unsatisfiability is detected, TheoryR reports a contradiction. If the extended conjunction is satisfiable, TheoryR outputs a set of equalities that is equivalent to the set of all equalities between arithmetic terms implied only by the extended conjunction (and not by the old one). The addition of literals to TheoryR can be undone.

The Figure 2-9 shows the organization of the TheoryR module. TheoryR depends on modules *simplex* and *tableau*. These two modules together implement a version of the simplex algorithm that is used in TheoryR. The tableau is a *simplex tableau* [Sch86], with some modifications that will be mentioned later. The module simplex contains a collection of optimization (minimization and maximization) procedures that are based on the simplex computational procedure.

In the rest of this subsection, I will describe how the version of simplex used in TheoryR is different from the classic simplex algorithm.

The simplex method is traditionally used to solve optimization problems of the form:

maximize $c \cdot x$

subject to the constraints:

$$A \cdot x = b$$

$$x \geq 0$$

where c and x are m-dimensional vectors, A is an $n \times m$ matrix, and b is an m-dimensional vector.

The linear constraints define a polyhedron. The linear form $c \cdot x$ therefore achieves its maximum on one of the extreme points (vertices) of the polyhedron. The simplex algorithm consists of two parts:

- Determine one extreme point of the polyhedron (if none exists, the polyhedron is void, and the problem has no solution).
- Given the starting extreme point, determine the extreme point where the maximum of the objective function is achieved by employing the *simplex computational procedure*. If the polyhedron is unbounded the procedure will detect it.

The first phase of the simplex algorithm could also be done by using the simplex computational procedure. Note that the first phase exactly performs the satisfiability check for the set of linear constraints. For more information on the simplex method see [Sch86].

Nelson's way of treating linear arithmetic constraints can be viewed as a variant of solving the first phase of simplex method in linear programming. There are, however, several important differences.

- Nelson's method works incrementally: both equality constraints and inequality constraints can be given one at a time. This requires certain changes to the simplex computational procedure, which assumes that all constraints are given at once, and all unknowns are non-negative.
- Nelson's method enhances the simplex computational procedure to determine all

equalities implied by the current set of constraints. This is done by developing a special format for the simplex tableau in which such equalities can be detected easily.

For more details of Nelson's method see [NO80].

2.3 Important details of the work

In this section I will focus on important properties of my implementation of the specialized theories module. The details that I will describe in this section can be classified into three categories:

- Communication between the specialized theories. I will explain properties of my implementation of Nelson's algorithms.
- Integration of the specialized theories into LP. I will describe the changes that were made in LP in order to incorporate and use specialized theories.
- Communication of the specialized theories with the user. I will describe how specialized theories generate appropriate output for the user.

2.3.1 Communication between specialized theories

One of the design goals for the incorporation of specialized theories into LP was to create a framework that would allow us to put new theories in or take some theories out easily. This extensibility property is essential, because a complete set of useful procedures is not yet identified.

Greg Nelson's algorithms and combining method allow for extensible implementations. However, in earlier usages ([NO80]) extensibility was less important and sometimes sacrificed for efficiency.

In my implementation, I enhanced extensibility by providing a standardized modular interface to specialized theories. In this interface the theories propagate equalities as immutable

objects (literals). An alternative implementation could keep pointers among data structures representing different theories, such that pointers would relate corresponding nodes. In this way, if two nodes in one of the structures would become equivalent, this equality can be propagated by following pointers to other structures, and merging corresponding nodes in them. The disadvantage of such an implementation is that when a new theory is added, some of the existing data structures have to be modified to keep pointers to the new theory, and the new theory has to know about existing theories, in order to maintain pointers to them.

The advantage of my implementation is that the modules implementing different specialized theories are independent of each other. They communicate via the ground_system module. An individual specialized theory can accept a fact from the ground_system and report all entailed facts back to it. This implementation enables us to add or remove specialized theories without changing (or knowing about) the implementation of existing ones.

The described interface to the specialized theories can also be used to obtain more deductive power. This can be done by propagating not only equalities, but also some other facts.

In my implementation I started to propagate disequalities. Disequalities are propagated to TheoryR by the ground_system, and this causes TheoryR to perform more inferences.

For example, assume that the inequality

$\mathtt{x} \leq \mathtt{y}$

is in the conjunction maintained by TheoryR. If TheoryR receives the disequality

$\mathtt{x} \neq \mathtt{y}$,

it can derive

$\mathtt{x} < \mathtt{y}$.

TheoryR does not add disequalities to its conjunction, since maintaining disequalities can be done more efficiently by TheoryE. Therefore, if nothing else is done, in the case when $x \neq y$

appears first and $x \leq y$ second, $x < y$ will not be derived. To fix this, the ground_system preforms special steps to correlate disequalities and inequalities:

- If the ground_system receives an inequality like $x \leq y$, it forms the equality $x = y$ and sends it to TheoryE.
- If TheoryE reports a contradiction which means that $x \neq y$ is known to TheoryE, $x < y$ is added to TheoryR.
- The ground_system undoes the assertion of $x = y$ to TheoryE.

Note that for integer and natural linear arithmetics, propagation of disequalities brings more deductive power. In the example above, if x and y are integers, then once $x < y$ is added to TheoryR, $x \leq y - 1$ will be added also via the additional inference mechanisms for integers described in Section 2.2.4. This literal would not be discovered otherwise. For rational linear arithmetic no new deductive power is obtained (since reasoning about rationals is complete anyway), but discovering $x < y$ at an earlier stage can improve its performance.

2.3.2 Integration of specialized theories and the Larch Prover

Specialized theories by themselves carry some deductive power. An important part of the work described in this thesis was to make appropriate changes to LP in order to use this power. In general, for each specialized theory there should be a reasoning procedure and also ways to define information relevant to the theory.

In my work I have incorporated three specialized theories into LP: boolean propositional logic, the theory of equality with uninterpreted functional symbols, and the theory of linear arithmetic. It turned out that propositional logic and the theory of equality did not need any special support besides the reasoning procedures. Since LP was dealing with first order logic, it had all the concepts necessary for propositional logic and equality already. Therefore, the deductive power for propositional logic and the theory of equality could be added without special support.

```
declare sort Time, Length
include Rational(Time)
include Rational(Length)
declare operator t1, t2 :→ Time
assert t1 = t2 + 1
```

Figure 2-10: Multiple instances of rational linear arithmetic

The situation with linear arithmetic was very different, because LP had no built-in notion of arithmetic before. Therefore I developed a way of using built-in arithmetic facilities in LP.

As it was said in Section 2.2, the specialized theories module in LP provides reasoning about three kinds of arithmetic:

- rational linear arithmetic;
- integer linear arithmetic;
- natural linear arithmetic.

Reasoning about rationals is complete; reasoning about integers and naturals is incomplete for efficiency reasons.

Multiple instances are allowed for each arithmetic theory. In Figure 2-10 the sorts **Time** and **Length** are declared. The **include** commands state that both sorts satisfy the arithmetic theory of the rationals. This means that the arithmetic and ordering operators ($+, -, *, <, >, \leq, \geq$) are automatically defined with their usual semantics for both sorts. Numerals for both sorts are also defined. The next line defines constants, **t1** and **t2**, of sort **Time**. The **assert** command illustrates the usage of the operator "+" and the numeral "1" that were implicitly defined by the **include** command.

The current implementation of arithmetic allows the user to reason about arithmetic properties of **Time** and **Length**.

2.3.3 Communication of specialized theories with the user

As mentioned in Section 2.1, specialized theories can communicate with the user if a proof by specialized theories fails. This is consistent with the general principle of providing hints of "what to do next" when the prover gets stuck. In case of failure the specialized theories output the satisfying assignment of the negation of the conjecture and all variable-free facts of the system.

Since the user will have to read the assignment and make decisions based on it, the important task is to print the assignment in a nice format. Indeed, in the examples shown in Section 2.1, it was easy to make the decisions based on the assignments, because they were previously simplified. The unsimplified assignment from the example in Figure 2-2 is:

```
¬(x + y > x).
```

The unsimplified version of assignment from Figure 2-3 is:

```
¬((f(b) + a) ≥ (g(b) + a)).
```

It is harder to draw conclusions from unsimplified versions than from simplified ones. The specialized theories module, in fact, performs the simplifications that are shown on the examples. In the remainder of this subsection, I describe how the assignments are simplified in the specialized theories module.

There are two major problems with satisfying assignments:

- Satisfying assignments are often very long, and contain a lot of redundant literals.
- Single literals are complicated and hard to read.

The specialized theories module does some work to eliminate redundancies from assignments, and therefore make them shorter. It also takes some steps to simplify single literals using knowledge about the specialized theories.

All simplifications are only possible on literals that are relevant to one of the incorporated specialized theories. In case of the three theories that I worked with there are two types of relevant literals:

- equality literals;
- inequality literals.

The simplification of a set of equality literals is done by computing an internormalized (mutually irreducible) set of rewrite rules, equivalent to the original set of equalities. This was done by employing a fast ground completion algorithm due to W. Snyder ([Sny94]) in TheoryE. The complexity of the algorithm is bounded from above by the cost of the congruence closure algorithm, which in our implementation is $O(n^2)$.

Simplification of inequality literals consists of several parts:

- Some redundant inequalities are eliminated by the TheoryR itself. This is done by always performing redundancy checks when inequalities are asserted.
- "Negative" inequality literals are transformed to equivalent inequalities, e. g., $\neg(a > b)$ is transformed to $a \leq b$.
- After the internormalized rewrite system for the set of equalities is computed it is used to reduce the atomic subterms of arithmetic terms to normal forms.
- The usual algebraic simplification of literals is performed:
  1. Like terms are grouped together, e.g., $a + a < b$ is transformed into $2 * a < b$.
  2. Inequalities are transformed to have 0 in right hand side, e. g., $a < b$ is transformed into $a - b < 0$
  3. If during this process some redundant literals (like $-1 < 0$) are found, they are eliminated.

Figure 2-11 shows a sample simplification of an assignment.

An internormalized rewrite system, corresponding to the two equalities is

```
Set of literals
   {a = b, b = 1, a + b < 3, a + c < 1}
Is printed as
   {a = 1, b = 1, c < 0}
```

Figure 2-11: Simplification of an example assignment

```
a → 1
b → 1.
```

The inequality $a+b<3$ is normalized to $1+1<3$, and then simplified to $-1<0$, which is redundant, and therefore is eliminated. The inequality $a+c<1$ is normalized to $1+c<1$, and then is simplified to $c<0$.

Chapter 3

Experiment with the enhanced Larch Prover

I used the Larch Prover enhanced with specialized theories on a sample proof. The proof verifies some properties of a concurrent algorithm. The main model used in the proof is the *timed input/output automata* model ([LA91]). The original LP proof (which did not use specialized theories) first appeared in [Söy94]. Dealing with time bounds in the proof required a lot of arithmetic reasoning, and the LP proof in [Söy94] requires a lot of user-interaction to perform it. I produced a simpler version of the proof by employing specialized theories. Here is the overview of this chapter:

- In Section 3.1 I give background on the model and proof technique used in the sample proof.
- In Section 3.2 I present the original LP proof.
- In Section 3.3 I show my version of the proof. My version is shorter than the original and is checked faster.

3.1 Background

3.1.1 Input/Output Automata

Input/output Automata (I/O automata) provide a model for describing components of an asynchronous distributed system. An I/O automaton A is a state transition machine in which transitions are associated with *actions.*

Formally, an I/O Automaton A has five components:

- *states*(A), a set of states (possibly infinite)
- *start*(A), a set of start states (a non-empty subset of *states*(A))
- *actions*(A), a set of actions. Three classes of actions can be distinguished: *input* actions (*input(A)*), *output* actions (*output(A)*) and *internal* actions (*internal(A)*). Input and output actions are called *external*. So $external(A) = input(A) \cup output(A)$.
- *trans(A)*, a set of triples such that $trans(A) \subseteq states(A) \times actions(A) \times states(A)$. $Trans(A)$ is called a transition relation. An action a is called *enabled* in state s if there exists a triple $(s, a, s') \in trans(A)$. $Trans(A)$ must be such that any input action is enabled at any state.
- *part*(A), a partition of $internal(A) \cup output(A)$ into disjoint classes (known as *tasks*). $Part(A)$ is an equivalence relation on $internal(A) \cup output(A)$. A task T is called *enabled* if any of its actions is enabled.

An execution of an I/O automaton is a sequence

$s_0, a_0, s_1, a_1, ...$

such that:

- $s_o \in start(A)$
- $\forall i (s_i, a_i, s_{i+1}) \in trans(A)$.

A state s of A is said to be *reachable* if it is the final state of some finite execution of A.

An index i is called an *initial* index for a task $T \in part(A)$ in an execution e if T is enabled for s_i in e, and either $i = 0$ or T is not enabled in s_{i-1}.

The *trace* of an execution fragment e (denoted $trace(e)$) is the list of all external actions from this fragment. A *trace* of an automaton A is a trace of some execution of A.

To facilitate reasoning about timing properties, an I/O automaton A can be augmented with a *boundmap* on tasks produced by $part(A)$ ([LA91]). The boundmap sets time bounds for each task of the automaton. If T is a task, then a boundmap b specifies time bounds $b_{lower}(T)$ and $b_{upper}(T)$. Intuitively, the time bounds specify how much time should pass after T became enabled until either an action of T occurs or T becomes disabled. An automaton A together with a boundmap b forms an MMT automaton (named after Merritt, Modungo and Tuttle, who first defined the model [LA91]). I denote the MMT Automaton corresponding to A by A'.

A timed execution e of an MMT automaton A' is a sequence α

$s_0, (a_0, t_0), s_1, (a_1, t_1), ...$

such that:

- The sequence $s_0, a_0, s_1, a_1, ...$ is an execution of A.
- The times $0 \leq t_1 \leq t2 \leq ...$ are nondecreasing nonnegative numbers.
- If i is an initial index of task $T \in part(A)$ in the untimed execution corresponding to α then
    1. If $b_{upper}(T) < \infty$, then there exists $j > i$ with $t_j \leq t_i + b_{upper}(T)$ such that either $a_j \in T$ or T is disabled at s_j.
    2. There exists no $j > i$ with $t_j < t_i + b_{lower}(T)$ and $a_j \in T$.

An infinite timed execution is called *admissible* if the times in it increase without bound. A *timed trace* of the execution is the subsequence of external actions and their associated

times. The *admissible timed traces* are the timed traces of admissible timed executions of A'

It is sometimes convenient to embed the boundmap b in the automaton A itself. This kind of automaton is called timed automaton ([LA91]) and is denoted by $time(A, b)$. The embedding is performed by including additional components in the states and actions of regular I/O automaton A.

Each state of $time(A, b)$ includes three additional components:

- a real valued variable *now*, which represents the current time;
- a function $first$, which represents the earliest time some action from each task of $part(A)$ can occur;
- a function $last$, which represents the latest time some action from each task of $part(A)$ can occur.

Actions of $time(A, b)$ are pairs (a, t) where a is either an untimed action of A, or a special time passage action ν. Intuitively, t denotes the time at which the action a occurs.

In the start state s_0, $first(T) = b_{lower}(T)$ and $last(T) = b_{upper}(T)$. If a task T is not enabled in state s then $first(T) = 0$ and $last(T) = \infty$ in state s.

A legal timed transition $(s, (a, t), s')$, as well as values of $first$ and $last$, can be defined as follows:

1. If $a \in Actions(A)$ then:

- The value of the variable *now* is the same in s and s'.
- The action a makes a legal transition from the untimed version of s into the untimed version of s' in automaton A;
- If $a \in T$ then $first(T) \leq t$ in s'.
- If T is enabled in s and $a \notin T$ and T is enabled in s', then the state components $first(T)$ and $last(T)$ of s' are the same as those of s.

- If T is enabled in s and either $a \in T$ or T is not enabled in s', then $first(T) = t + b_{lower}(T)$ and $last(T) = t + b_{upper}(T)$ in state s'.

2. If $a = \nu$ then

- $now = t$ in s';
- $t \leq last(T)$ in state s for all T (this restricts the amount of time that can pass so as not to exceed the upper bound on any task);
- $first(T)$ and $last(T)$ in s' are the same as in s for all T.

The execution of a timed automaton $time(A, b)$ is similar to one of the MMT automaton A', but for all i, $(s_i, (a_i, t), s_{i+1})$ must form a legal timed transition. A timed trace of $time(A, b)$ is the subsequence of external actions of some execution of $time(A, b)$.

3.1.2 Invariants and simulations

An *invariant* I_A of an automaton A is any property that holds in all reachable states of the automaton.

Let A and B be timed automata with the same set of external actions. We say that B correctly implements A if any timed trace of B is a timed trace of A.

Usually we are interested in proving that B correctly implements A. This can be done by finding a *simulation mapping* from B to A.

A simulation mapping from B to A is a relation f over $states(B)$ and $states(A)$ such that:

- If $f(s, u)$, then the *now* components of states s and u are equal.
- If $s \in start(B)$, then there exists some $u \in start(A)$ such that $f(s, u)$.
- If $(s, (a, t), s')$ is a legal timed transition of B, and $I_B(s)$, $I_A(u)$, and $f(s, u)$ are true then there exists some $u' \in states(A)$ such that $f(s', u')$, and there exists some timed execution fragment of A from u to u' such that its timed trace is same as that of (a, t).

State components
reported: Boolean, initially *false*
count: Integer, initially $k \geq 0$
Actions
Output *report*
Pre: $count = 0 \wedge \neg reported$
Eff: $reported \leftarrow true$
Internal *decrement*
Pre: $count > 0$
Eff: $count \leftarrow count - 1$
Classes of $part(C)$ **with boundmap**
$\{report\}$: $[c_1, c_2]$
$\{decrement\}$: $[c_1, c_2]$

Figure 3-1: A counting automaton $C(k, c_1, c_2)$

The following theorem is proved in [LA91]:

Theorem: If there exists a simulation mapping f from B to A then every admissible timed trace of A is an admissible timed trace of B.

3.2 Original sample proof

In the sample proof the correctness and timing properties of a simple automaton are verified.

3.2.1 Problem description

In the sample proof two timed automata are considered. One is a *specification* automaton R; another is an *implementation* automaton C.

The implementation automaton C is a counting automaton. It counts downwards from some fixed constant k ($k \geq 0$), and it issues a *report* when 0 is reached. Figure 3-1 shows the description of C.

A state of $C(k, c_1, c_2)$ consists of two components: a boolean variable *reported* and a non-

State components
reported: Boolean, initially *false*
Actions
Output *report*
Pre: $\neg$*reported*
Eff: *reported* $\leftarrow$ *true*
Classes of *part*(R) **with boundmap**
{*report*}: $[a_1, a_2]$

Figure 3-2: A report automaton $R(a_1, a_2)$

negative integer variable *count*. The transition relation is described in a precondition-effects style. The automaton C has one output action *report* and one internal action *decrement*; it also has two tasks, specified by *part*(C). Each task contains precisely one action. C is timed, and the time bounds on the tasks are the following:

- $b_{lower}(\{\text{report}\}) = b_{lower}(\{\text{decrement}\}) = c_1$;
- $b_{upper}(\{\text{report}\}) = b_{upper}(\{\text{decrement}\}) = c_2$;

The specification automaton R just issues the *report* message. Figure 3-2 describes R. The states of $R(a_1, a_2)$ consist of a boolean variable *reported*. $R(a_1, a_2)$ has a single output action *report*. *Part*(R) specifies single class.

The sample proof verifies that automaton $C(k, c_1, c_2)$ correctly implements the specification automaton $R(a_1, a_2)$ when $a_1 = (k+1) \times c_1$ and $a_2 = (k+1) \times c_2$. The proof was done in the following way:

- Timed automata (automata with timing information embedded in states) for both R and C were constructed as described in 3.1.1.
- A mapping $f(s, u)$ between states of C and R was defined as follows: $f(s, u)$ is true if and only if
  1. $u.now = s.now$

2. $u.reported = s.reported$
3. $u.first(report) \leq \begin{cases} s.first(decrement) + s.count \cdot c_1 & \text{if } s.count > 0 \\ s.first(report) & \text{otherwise} \end{cases}$
4. $u.last(report) \geq \begin{cases} s.last(decrement) + s.count \cdot c_2 & \text{if } s.count > 0 \\ s.last(report) & \text{otherwise} \end{cases}$

(Record notation is used to denote state components)

- The mapping $f(s, u)$ was proved to be a simulation relation by establishing the three properties defined in Section 3.1.2.

3.2.2 Axiomatization

The definitions of the specification and implementation automata and the definition of simulation relation were axiomatized using the Larch Shared Language (LSL), which is a first-order specification language [GH93]. The complete LSL specification of the problem is given in [Söy94]. Here I show some fragments of it, which will be needed to illustrate the use of specialized theories.

Recall that among other specialized theories, we had incorporated theory of linear arithmetic. The definitions of the automata R and C as well as the definition of the mapping f involve arithmetic.

The counter of the counting automaton C is a natural number. Figure 3-3 shows the LSL axiomatization of the counting automaton. The second line the specification includes the knowledge about natural numbers. The LSL specification of natural numbers is shown in Figure 3-4.

Most properties of natural numbers are defined in the **includes** part of the specification by referring to other specification modules. **DecimalLiterals** defines properties of natural numerals, **TotalOrder** defines the ordering relations $<, \leq, >$ and $\geq$ on naturals with the usual semantics. **ArithOps** defines operators $+$ and $*$. The **asserts** part introduces some additional axioms including a definition of the $\ominus$ (restricted minus) operator.

Another place where arithmetic is used is the definition of a boundmap. Figure 3-5 contains

```
AutomatonCount (C, k): trait
  includes UntimedAutomaton(C), CommonActionsRC, Natural
  States[C] tuple of count: N, reported: Bool
  introduces
    k: → N
    decrement, report: → Actions[C]
  asserts
    sort Actions[C] generated by decrement, report
    sort Tasks[C] generated by task
    ∀ s, s': States[C], a, a': Actions[C]
      isExternal(report);
      isInternal(decrement);
      common(report) = report;
      task(a) = task(a')        ⇔ a = a';
      start(s)                  ⇔ ¬s.reported ∧ s.count = k;
      enabled(s, report)        ⇔ s.count = 0 ∧ ¬s.reported;
      effect(s, report, s')     ⇔ s'.count = s.count ∧ s'.reported;
      enabled(s, decrement)     ⇔ s.count > 0;
      effect(s, decrement, s') ⇔ s'.count + 1 = s.count
                                    ∧ s'.reported = s.reported;
      inv(s)                    ⇔ s.count > 0 ⇒ ¬s.reported
```

Figure 3-3: LSL specification of the counting automaton C

```
Natural (N): trait
   includes
    DecimalLiterals,
    TotalOrder (N),
    ArithOps(N)
  introduces
    __ ⊖ __: N, N → N
  asserts
    N generated by 0, succ
    ∀ x, y: N
      succ(x) ≠ 0;
      succ(x) = succ(y) ⇔ x = y;
      x < succ(x);
      0 ⊖ x ⇔ 0;
      x ⊖ 0 ⇔ x;
      succ(x) ⊖ succ(y) ⇔ x ⊖ y
```

Figure 3-4: LSL specification of natural numbers

```
Bounds: trait
  includes Real(Time)
  Bounds tuple of bounded: Bool, first, last: Time
  introduces
    __+__: Bounds, Time → Bounds
    __+__: Bounds, Bounds → Bounds
    __*__: lpNat, Bounds    → Bounds
    __ ⊆ __: Bounds, Bounds → Bool
    __ ∈ __: Time, Bounds → Bool
  asserts ∀ b, b1, b2: Bounds, t: Time, n: lpNat
    0 ≤ b.first;
    b.first ≤ b.last;
    b + t = [b.bounded, b.first + t, b.last + t];
    b1 + b2 =
      [b1.bounded ∧ b2.bounded, b1.first + b2.first, b1.last + b2.last];
    n * b = [b.bounded, n * b.first, n * b.last];
    b1 ⊆ b2 ⇔
      b2.first ≤ b1.first
        ∧ ( (b1.bounded ∧ b2.bounded ∧ b1.last ≤ b2.last)
              ∨ ¬b2.bounded );
    t ∈ b ⇔ b.first ≤ t ∧ (t ≤ b.last ∨ ¬b.bounded)
```

Figure 3-5: LSL specification of time bounds

the LSL specification of time bounds.

Time is defined using real numbers. In the second line of Figure 3-5 the specification **Real** is associated with sort **Time**. A partial LSL specification of real numbers is shown in Figure 3-6. The specification defines some properties of real numbers that are necessary for the proof. It states some properties of real addition (+) and defines ordering operations $<, \leq, >, \geq$. It also defines multiplication by a natural number.

3.2.3 LP proof

Specifications of the problem were written in LSL. Then they were automatically translated into the input language of LP by employing the LSL specification checker. The LP proof was constructed interactively. The proof verified that the relation f is indeed a simulation relation. Section 3.1.2 states three properties that we have to establish in order to prove the simulation. We also have to establish strong enough invariants to carry on the proof.

```
Real (R): trait
  includes Natural, TotalOrder(R), AC(+, R)
  introduces
    0: → R
    __ + __, __ - __: R, R → R
    __ * __: N, R → R
  asserts ∀ t, t1, t2: R, n: N
    0 + t = t;
    (t + t1) < (t + t2) ⇔ t1 < t2;
    (t + t1) ≤ (t + t2) ⇔ t1 ≤ t2;
    (t + t1) = (t + t2) ⇔ t1 = t2;
    0 * t = 0;
    1:N * t = t;
    (n+1)*t = (n*t) + t;
  implies ∀ t: R, n: N
    0 ≤ t ⇒ 0 ≤ (n * t)
```

Figure 3-6: LSL specification for some properties of real numbers

Therefore the proof consists of two main pieces: proof of invariants and proof of simulation, which itself consists of three parts, each of which proves one of the three properties of simulation. The proofs of invariants and the first two properties of simulation are not interesting for our purpose, since they do not use specialized reasoning. But the proof of the third property involves a lot of arithmetic reasoning. Figure 3-7 shows the proof obligation of the third property. Figure 3-8 shows the part of the LP proof that proves the third property.

3.3 Simplified sample proof

I took the specifications of the problem and simplified them, so that they use the new arithmetic facilities that I added to the system instead of the general LP facilities like rewrite rules and deduction rules that were used to axiomatize arithmetic properties in the original proof. Then I adjusted the proof, by employing built-in specialized theories instead of manual guidance where it was possible. In the rest of the section I give details of the experiment.

```
declare variables u: States[TR], alpha: StepSeq[TR]
prove
  f(s, u)
    ∧ isStep(s: States[TC], a, s')
    ∧ inv(s:States[TC])
    ∧ inv(u:States[TR])
    ∧ inv(s.basic:States[C])
  ⇒ ∃ alpha (execFrag(alpha)
                ∧ first(alpha) = u
                ∧ f(s', last(alpha))
                ∧ trace(alpha) = trace(a:Actions[TC]))
  by induction on a:Actions[TC]
  ..
```

Figure 3-7: Proof obligation of the third property of simulation

3.3.1 Changes in the specifications

The new axiomatization does not include any specific axioms for natural numbers. This is due to the fact that the necessary semantics is provided by the built-in arithmetic theory. This introduces a difficulty: since no rewrite rules about arithmetic are introduced by the new axiomatization, no arithmetic simplification of conjectures and other axioms can be performed automatically by rewriting. Some rules that were introduced in the old axiomatization were useful for simplification. Therefore, I introduced these rules as lemmas in the new proof. Figure 3-9 shows the lemmas about naturals that were used. Each of them can be proved by a single call to the built-in specialized theories.

The arithmetic properties of time are implemented using some properties of real numbers. The specification for Reals was also simplified. Figure 3-10 shows the new specification. If we compare it to Figure 3-6, we see that the definitions of the ordering operations and the operation + disappeared. The necessary semantics is provided by the built-in arithmetic facilities. However, the operator * on naturals and reals still has to be specified explicitly, since built-in arithmetic does not support such "mixed" arithmetic operations. As in the case with naturals, I needed some additional lemmas. These appeared in the **implies** clause of the specification.

```
resume by cases a1c = decrement, a1c = report
  % Case 1: simulate decrement action
  resume by specializing alpha to null(uc)
  instantiate c:Tasks[C] by task(report) in *impliesHyp
  instantiate c:Tasks[C] by task(decrement) in *impliesHyp
  resume by case s'c.basic.count = 0
    resume by case (uc.bounds[reportTask]).bounded
      resume by ∧ -method
        apply Transitivity to conjecture
        apply Transitivity to conjecture
      apply transitivity to conjecture
    instantiate t:Time by c.first, n by s'c.basic.count in Real
    instantiate t:Time by c.last, n by s'c.basic.count in Real
      resume by case (uc.bounds[reportTask]).bounded
        resume by ∧ -method
          apply Transitivity to conjecture
          apply Transitivity to conjecture
        apply Transitivity to conjecture
  % Case 2: simulate report action
  resume by specializing
    alpha to null(uc) {addTime(report, uc.now),
                  [[true],uc.now,update(uc.bounds,task(report),[false,0,0])]}
          ..
    resume by induction on c:Tasks[R]
  % Case3: simulate passege of time
  resume by specializing alpha to null(uc) {nu(t1c),[uc.basic,t1c,uc.bounds]}
      resume by induction on c:Tasks[R]
        resume by case sc.basic.count = 0
          instantiate c: Tasks[C] by task(report) in *Hyp
          resume by case (uc.bounds[task(report)]).bounded
          instantiate c: Tasks[C] by task(report) in *Hyp
          instantiate c: Tasks[C] by task(decrement) in *Hyp
          instantiate
            y:Time by s'c.basic.count*c:Bounds.last,
            x:Time by (s'c.bounds[task(decrement)]).last in *Lemma
            ..
          resume by case (uc.bounds[task(report)]).bounded
qed
```

Figure 3-8: The old LP proof of the third property of simulation

```
∀ n: N
   0 + n = n;
    n ≤ n;
   ¬(n < n);
   n = 0 ∨ n > 0;
   n > 0 ⇔ n ≠ 0
```

Figure 3-9: Lemmas about naturals that are used in enhanced LP proof

```
Real (R): trait
  includes Natural(N)
  introduces
    __ * __: N, R → R
  asserts ∀ t, t1, t2: R, n: N
    0 * t = 0;
    1:N * t = t;
    (n+1)*t = (n*t) + t;
  implies ∀ t: R
    0 ≤ t ⇒ 0 ≤ (n * t);
    0 + t = t;
    t ≤ t
```

Figure 3-10: The specification of reals used in the new proof

```
resume by cases a1c = decrement, a1c = report
  % Case 1: simulate decrement action
  resume by specializing alpha to null(uc)
    instantiate c:Tasks[C] by task(report) in *impliesHyp
    instantiate c:Tasks[C] by task(decrement) in *impliesHyp
    resume by case s'c.basic.count = 0
      zap
      instantiate t:lpArith by c.first, n by s'c.basic.count in Real
      instantiate t:lpArith by c.last, n by s'c.basic.count in Real
      zap
  % Case 2: simulate report action
  resume by specializing
     alpha to null(uc) {addTime(report, uc.now), [[true], uc.now,
                   update(uc.bounds, task(report), [false, 0, 0])]}
       ..
       resume by induction on c:Tasks[R]
    % Case 3: simulate passage of time
    resume by specializing alpha to null(uc) {nu(1c), [uc.basic, 1c, uc.bounds]}
      resume by induction on c:Tasks[R]
        instantiate c:Tasks[C] by task(report) in *Hyp
        resume by case sc.basic.count = 0
          zap
          instantiate c:Tasks[R] by reportTask in *Hyp
          instantiate n by sc.basic.count in TimedAutomaton
          instantiate c:Tasks[C] by task(decrement) in *hyp
          zap
qed
```

Figure 3-11: The new LP proof of the third property of simulation

3.3.2 New proof is 30 % shorter

As expected, the LP proof that uses specialized theories is shorter than the original LP proof, because some reasoning is done automatically by employing specialized theories. Figure 3-11 shows the new LP proof, which is a subset of the original proof in Figure 3-8 with specialized reasoning replaced by calls to specialized theories (command **zap**). It is about thirty percent shorter than the original one.

To better understand why the new proof is shorter, consider a more detailed comparison of two corresponding fragments from the two proofs. Figure 3-12 shows a fragment from the original proof and the corresponding fragment from the new proof. Five

Fragment of the original proof:

```
    < ... >
  resume by case (uc.bounds[task(report)]).bounded
    resume by ∧-method
    apply Transitivity to conjecture
    apply Transitivity to conjecture
  apply Transitivity to conjecture
    < ... >
```

Corresponding fragment of new (enhanced) proof:

```
    < ... >
  zap (call to specialized theories)
    < ... >
```

Figure 3-12: Why proofs are getting shorter

lines of original proof were replaced by single call to the specialized theories. The first line of the original fragment performs a case split. It first assigns the boolean term $(uc.bounds[task(report)]).bounded$ to $true$ and then to $false$, and tries to prove both cases. This is boolean reasoning. The second line uses a proof by conjunction, by proving each conjunct individually. This is also part of boolean reasoning. The three remaining lines of the first fragment use the transitivity property of the relation $\leq$ and therefore are performing linear arithmetic reasoning. Since all these lines just perform reasoning about specialized theories that are incorporated into Larch Prover, they can be performed by a single call to the specialized theories in the new proof.

3.3.3 New proof is three times faster

The amount of guidance needed to construct a proof is an important measure, but the time to perform the checking of an existing proof is also very important. In this section I present some profiling data showing that the new proof is checked three times faster than the old one. Figure 3-13 shows the measurements taken for the two proofs by the built-in profiling

	Original proof	Enhanced proof	Change in time
Rewriting	53.60	23.66	−29.94
Deductions	26.94	02.05	−24.89
Prover	1 : 53.50	20.82	−1 : 32.68
Specialized theories	00.19	12.05	+11.86
Total	3 : 14.23	58.58	−2 : 15.65

Figure 3-13: New proof is faster

mechanism of the Larch Prover.

Figure 3-13 presents measurements of how much time was spent in different inference components of LP. In total, the new proof takes about 2 minutes less to check.

We can observe that time spent doing rewriting decreased considerably. This happened because some fragments of the proof were replaced by calls to specialized theories. These calls happen atomicly; therefore, if in the old proofs rewriting was activated during the fragments, it is not activated in the new proof. Another reason for the decrease in the rewriting time is that some rules describing arithmetic properties were taken out.

The inference mechanism called *deduction rules* is used to handle implications in LP. In the original proof deduction rules were used to handle transitivity properties of arithmetic ordering relations, and most of the time spent in deductions was used by these transitivity rules. As follows from Figure 3-13, the original proof spends about 25 seconds more doing deductions than the enhanced one. This happens because in the enhanced proof transitivity is handled by specialized theories. The time spent in deductions in the new proof is not zero, because LP hardwired deduction rules are used to simplify formulas, and these are

still used in the new proof.

In Figure 3-13 the prover component can be viewed as the controller of all inference mechanisms and commands of LP. It is responsible for applying deduction rules and rewrite rules, generating proof obligations upon receiving user commands, etc. The time spent in the prover decreased a lot, since there are fewer rewrite rules and deduction rules in the enhanced proof, and there are fewer explicit commands too.

Figure 3-13 shows that the time spent in the specialized theories module increased by about twelve seconds. But this is a very small price to pay for a big decrease of time in other measures.

Chapter 4

Conclusion

The primary goal of enhancing the Larch Prover with specialized theories was to reduce the amount of user guidance needed for constructing proofs and the amount of time needed for checking proofs. As shown in Chapter 3, the current implementation is helpful in reducing both the amount of interaction and the running time. There is, however, room for further improvements. Besides, some interesting problems emerged during the work. In this chapter, I describe both routine enhancements of the current tool and interesting future work that would require some research.

4.1 Possible enhancements to the existing tool

4.1.1 Possible improvements to the linear arithmetic procedure

The current procedure for linear arithmetic is based on the simplex algorithm. The current implementation keeps all the information in a *simplex tableau* that includes a matrix of coefficients of linear constraints. In practice this tableau is very sparse, and the current implementation does not handle sparse tableaus efficiently. As a result, the procedure for arithmetic theory takes more space than it should. It is also slower than it should be,

because the procedure performs many vector (row and column) comparisons. An efficient implementation of sparse matrices and vectors could improve the efficiency of the arithmetic procedure.

Currently, incomplete reasoning about integers and naturals is done by performing some extra inferences on top of the simplex-based arithmetic procedure (see Section 2.3.2 for details). As mentioned in the introduction, there exists another algorithm (sup-inf) for performing arithmetic [Sho77]. It would be interesting to implement the sup-inf algorithm and to compare the performance of simplex-based and sup-inf-based linear arithmetic procedures in practice.

4.1.2 Addition of new theories

An important enhancement to the specialized theories module would be the addition of more specialized theories. As mentioned earlier, a complete set of useful theories has not been identified yet and establishing such a set will require experimenting with the enhanced LP in order to identify useful theories. However, the following theories are likely to be useful:

- The theory of partial orders;
- The theory of transitivity;
- The theory of lists;
- The theory of arrays.

In the rest of this section, I will discuss these theories and sketch the basic algorithms for reasoning about them.

The theory of partial orders is the theory of two ordering relations, a reflexive relation ($\leq$) and the corresponding irreflexive one ($<$). One possible axiomatization of the theory is shown in Figure 4-1.

$$\forall x \quad x \leq x$$

$$\forall x \forall y \forall z \, [x \leq y \wedge y \leq z \Rightarrow x \leq z]$$

$$\forall x \forall y \quad [x \leq y \wedge y \leq x \Leftrightarrow y = x]$$

$$\forall x \forall y \quad [x < y \Leftrightarrow x \leq y \wedge x \neq y]$$

Figure 4-1: Axiomatization of partial orders

The set of facts that belongs to the theory (facts of the type $a < b$, $a \leq b$, or their negations) can be represented as a directed graph, where edges are labeled with either $<$ or $\leq$, and nodes are terms. The specialized procedure for the theory is based on the strongly connected components algorithm for a directed graph. The procedure should first put all the known constraints in the graph, and then compute strongly connected components of the graph. If there exists a component that has an edge labeled with $<$, then the set of constraints is unsatisfiable since it violates the irreflexivity of $<$. If all components have only edges labeled with $\leq$, then the set of constraints is satisfiable. The complexity of the algorithm is $O(V + E)$, where V is the number of nodes in the graph and E is the number of edges [CLR90]. Therefore the complexity of the procedure is $O(n)$, where n is the number of symbols in the set of constraints.

The theory of transitivity is the theory with one free boolean-ranged binary function T, and one axiom:

$$\forall x \forall y \forall z \, [(T(x, y) \wedge T(y, z)) \Rightarrow T(x, z)]$$

A set of facts of type $T(a, b)$ or $\neg T(a, b)$ can be represented as a graph, where nodes are terms, and two nodes, representing terms t_1, t_2 are connected by an edge if $T(t_1, t_2)$ is in the set. The specialized procedure is based on the transitive closure algorithm. The procedure first computes the transitive closure of the graph. Then, for each pair of terms t_1, t_2 such that $\neg T(t_1, t_2)$ is in the input set of facts, it checks whether there exists a path from t_1 to

$$\forall x \forall y \quad [car(cons(x,y)) = x]$$

$$\forall x \forall y \quad [cdr(cons(x,y)) = y]$$

$$\forall x \exists y \exists z \; [\neg atom(x) \Rightarrow x = cons(y,z))]$$

$$\forall x \forall y \quad \neg atom(cons(x,y))$$

$$atom(nil)$$

Figure 4-2: Axiomatization of theory of lists

t_2 in the closed graph. If so, the input set of constraints is unsatisfiable. Otherwise it is satisfiable. The complexity of the procedure is $O(n^3)$, where n is the number of nodes in the graph (which is proportional to the number of symbols in the input set of facts) [CLR90].

For both the theory of partial orders and the theory of transitivity, the possibility of having multiple instances of the theory is desirable. This poses a technical problem of representing multiple instances efficiently.

The theory of lists is the theory with free functions *nil*, *atom*, *cons*, *cdr* and *car*. The axioms of the theory are shown in Figure 4-2.

Nelson described a specialized procedure for this theory ([Nel80]). The procedure is based on the congruence closure algorithm and also on computing some important instances of the axioms from Figure 4-2. The worst case running time of the procedure is $O(n^2)$, and the average case is $O(n * log(n))$, where n is the number of symbols in the input set of facts.

The theory of arrays is the theory of two functions, *store* and *select*, with two axioms:

$\forall a \forall x \forall i \; [select(store(a,i,x),i) = x]$

$\forall a \forall x \forall j \forall i \; [i \neq j \Rightarrow (select(store(a,i,x)j) = select(a,j))]$

A specialized procedure for the theory of arrays was suggested by Nelson in [Nel80]. Like the procedure for lists, the procedure for arrays uses the congruence closure algorithm and some instances of the axioms. However, the algorithm has to perform case splits, and its running time is exponential.

4.2 Possible directions of future research

4.2.1 Communication between specialized theories and rewriting

Currently in LP specialized theories are applied only upon explicit call from the user. If there are no rewrite rules about built-in specialized theories in the system, rewriting will not simplify facts relevant to them. We have seen examples of such a situation in Section 3.3.1, where some lemmas were necessary for simplifications to be performed. In this section I will discuss the issue of automatic invocation of specialized theories during rewriting, which could reduce the number of lemmas needed.

One way to use specialized theories in rewriting is to reduce some boolean subformulas to *true* or *false*. Assume that we have subformula F in a term, then:

- If the specialized theories can prove F, then F can be reduced to *true*.
- If the specialized theories can prove $\neg F$, then F can be reduced to *false*.

Such a use of the specialized theories will give more deductive power to rewriting. Figure 4-3 shows an example of such an application of specialized theories.

In the example there are two rewrite rules, $a < b \rightarrow false$ and $a > b \rightarrow false$, where a and b are constants. The term *if* $f(a) = f(b)$ *then* c *else* d is irreducible by these rules. Note that if the lemma $a < b \vee a > b \vee a = b \rightarrow true$ is introduced, then rewriting can reduce the term to c (assuming that some rules for booleans are hardwired in the system).

By using the specialized theories we can eliminate the lemma. Instead, $f(a) = f(b)$ can be proved by the specialized procedures. Indeed:

Rewrite rules:
$a < b \rightarrow false$
$a > b \rightarrow false$

Term in Normal form without specialized theories:
if $f(a) = f(b)$ *then* c *else* d

Normal form with specialized theories:
c

Figure 4-3: Application of specialized theories in rewriting

$(\neg(a < b) \wedge \neg(a > b)) \vdash f(a) = f(b)$,

under the theory of arithmetic and the theory of equality.

Hence, the term *if* $f(a) = f(b)$ *then* c *else* d can be reduced to c without using the lemma if the specialized theories are used in rewriting.

A difficulty in following this approach is avoiding an exponential blowup in the specialized theories computation. One way to avoid it is to carefully choose subformulas to be attacked by the specialized theories. Another way is to use an incomplete version of the specialized theories which is efficient in time.

One possible class of subformulas to attack by specialized procedures are conditions in conditional rewrite rules. Conditional rewrite rules are used in LP to handle implications. They have the form:

```
c :: l → r,
```

which is equivalent to $c \Rightarrow (l = r)$. The formula c is called a *condition*. A conditional rewrite rule is applied to a term t if:

- The left side l of the rule matches t with substitution σ, and

Rewrite rules:
$x = y :: F(x, y) \rightarrow d$
$\quad a \leq b \rightarrow true$
$\quad b \leq a \rightarrow true$

Term in normal form without specialized theories:
$F(a, b)$

Normal form with specialized theories:
d

Figure 4-4: Using specialized theories in conditional rewriting

- $\sigma(c)$ can be reduced to *true* (discharged) by the rewrite rules of the system.

Conditions are often simple formulas (sometimes conjunctions of literals), and therefore attempting to discharge them with specialized theories may not be overly expensive. Figure 4-4 illustrates the use of specialized theories for discharging conditions in conditional rewrite rules.

There are three rewrite rules in Figure 4-4. The first one is a conditional rewrite rule. The left side $F(x, y)$ of the first rule matches the term $F(a, b)$ with substitution $\sigma = \{x \mapsto a, y \mapsto b\}$. To reduce the term, we must first discharge the condition $\sigma(x = y)$, which is equal to $a = b$. If specialized theories are not involved, we cannot prove $a = b$, and therefore $F(a, b)$ is in normal form. On the other hand, if we invoke specialized theories, we can prove $a = b$, since

$(a \leq b \wedge b \leq a) \vdash a = b,$

under the theory of arithmetic. In this case, the rule $x = y :: F(x, y) \rightarrow d$ can be applied to reduce the term $F(a, b)$ to d.

All examples shown so far illustrate that using specialized theories can bring more deductive power to rewriting. Specialized theories may make rewriting more efficient as well. For

example, if a subformula is a conjunction of equalities, the theory of equality can reduce it very quickly, probably faster than general rewriting facilities. However, it is hard to give convincing examples and justification why it is so.

It is important to create an easy-to-analyze framework for using specialized theories in rewriting. In particular it should provide ways of measuring deductive power and efficiency gained by using specialized theories.

One of the theoretical issues that would be interesting to examine is related to completeness. For example, if we remove the rewrite rules for booleans from the system, and use specialized theories in rewriting instead, what can we say about the completeness of our rewrite system?

4.2.2 Using specialized theories in explorative proof systems

When one is constructing a proof of a theorem by hand, it often happens that the proof branches into many independent subcases. While trying to prove a particular case, the constructor of the proof may find himself stuck. What some people normally do in such a situation is to temporarily stop working on the case and move to another (independent) subcase of the proof. It would be nice to support this kind of interface in a theorem prover.

One project that attempts to enhance LP with such an interface is described in [Voi92].

What is essentially needed for an explorative interface is the possibility of having access to different parts of the proof. The current implementation of LP would require copying the proof state. In a big proof the state is very large, and keeping multiple copies can result in memory overflow. Specialized theories make states even bigger.

In ordinary proof management (one that does not provide multiple subgoals), using specialized theories does not require a lot of space. This is because specialized theories are implemented using a backtrackable data structures that can efficiently "undo" changes made to them. Since the proof tree is traversed in a depth-first manner, the "undo" mechanism can be used to restore subgoals. Using "undo" rather than copying is efficient, since subgoals share a lot of state.

It is important to develop a mechanism like "undo" that would allow us to eliminate copying in proof systems with multiple subgoals.

One approach to the problem is to use *persistent* data structures, which were first suggested in 1986 by Driscoll, Sarnak, Tarjan and Sleator in [DSST86]. According to them, any ordinary data structure is *ephemeral* in the sense that once it is modified, the version of the structure prior to the modification is inaccessible. *Persistent* data structures allow access to such "past" states, by efficiently maintaining the history of different versions of the data structure.

In [DSST86] two kinds of persistent data structures are distinguished:

- *Partially* persistent – data structures that allow access to previous versions, but do not allow modifications of previous versions. Only the *current* (newest) version can be modified, producing a new version that will become current. The history therefore is linear, and previous versions just form a list. The current version is the last version in the list.
- *Fully* persistent – data structures that allow both access and modifications of previous versions. There is no notion of current version. If version v is modified, a new version v' is produced, and v' becomes a *child* of v. Therefore, versions make a tree.

In [DSST86] it is shown how to make a linked data structure with bounded in-degree fully persistent in an amortized $O(1)$ time and space cost per operation.

In 1989 Paul Dietz [Die89] showed how to make arrays fully persistent at an amortized cost of $O(log(log(n)))$ per operation and in $O(n)$ total space, where n is the number of operations to be performed.

In 1990 Dietz and Raman showed how to make a disjoint set data structure partially persistent [DR90]. They show that a sequence of m disjoint set operations, n of which are make-set operations, can be performed in $O(m + m\alpha(m, n))$ time and space on their partially persistent data structure ($\alpha(m, n)$ is the inverse of Ackermann's function). This essentially means that they can make disjoint sets partially persistent at the cost of $O(1)$ time and space per

operation.

However, making complicated data structures efficiently fully persistent is still an open research problem.

One important data structure of the specialized theories module is a congruence closure graph. A congruence closure graph is a subterm graph, with a binary relation R defined on its nodes, such that R is closed under equivalence and term congruence. It would be interesting to construct a fully persistent data structure for a congruence closure graph. To start with, it would be nice to extend the method of Dietz and Raman [DR90] to produce not only a partially persistent but a fully persistent data structure for disjoint sets.

There are other data structures used in both LP and the specialized theories module for which the design of efficient persistent analogs are needed, e.g., the partial order data structure that LP uses to maintain a termination ordering on terms, or the simplex tableau used by linear arithmetic theory.

There are some issues that are specific to theorem proving that are important to keep in mind while designing persistent data structures. One is the need to delete useless versions. E.g., the congruence closure graph is used in the satisfiability loop, which can be executed exponentially many times. However, not all versions produced in the loop are useful. It would be nice to delete useless versions in order to prevent an exponential increase in space. There are two approaches to this problem: one is to support an efficient *delete_version* operation for persistent data structures; another is to construct data structures in such a way that unnecessary versions will be garbage collected.

Another issue specific to interactive theorem proving is the propagation of changes made in one version to all children of the version in the version tree. The following example illustrates the importance of the issue.

Assume that a user has several subgoals to prove, and while working on the first one, he discovers a lemma that obviously is needed for all other subgoals as well. In the current LP, the user will have to prove this lemma separately in every case, since once a case is finished all lemmas proved during the case disappear from the system. It would be nice to

be able to move up in the proof tree, prove the necessary lemma in the parent context, and propagate it automatically to all child contexts.

There are several problems with the propagation of changes. First, it is not clear what semantics it should have. In the example above, what should happen if the user first proved the lemma in a parent context, and then deleted it in one of the child contexts? Another question is how to implement the propagation of changes efficiently.

4.2.3 Useful variable-free instances of axioms with variables

As was said before, the specialized theories module can work only with variable-free axioms. This is because only the variable-free satisfiability problem can be solved by Nelsons method. We have seen some examples (Chapter 2, Figure 2-3) where a proof by specialized theories fails because there are not enough variable-free instantiations of axioms containing variables in the system. Currently, a user has to make instantiations by hand. It would be nice, however, to find some way of producing useful instantiations automatically.

In his thesis [Nel80], Greg Nelson suggested a way of producing useful variable-free instances of an axiom by matching it against a set of equivalence classes produced by already known variable-free equations represented as a congruence closure graph. This method produces useful instances, but unfortunately it turns out to be NP-complete. So, it is too expensive to match all axioms of the system against the congruence closure graph. A cheaper way of discovering useful instances has yet to be developed.

Bibliography

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.

[BM88a] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[BM88b] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, pages 83–123. Clarendon Press, 1988.

[CLR90] Thomas H. Cornmen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Die89] Paul F. Dietz. Fully persistent arrays. Technical Report CS-TR-290, University of Rochester, June 1989.

[DR90] Paul F. Dietz and Radjeev Raman. Persistence, Amortization and Randomization. Technical Report CS-TR-353, University of Rochester, November 1990.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch Prover. In *Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, 1989. Springer-Verlag Lecture Notes in Computer Science 355.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Report 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[LA91] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. Technical Memo MIT/LCS/TM-412.e, Lab for Computer Science, Massachusetts Institute Technology, Cambridge, MA, November 1991.

[Lar90] Tracy Larrabee. Efficient generation of test patterns using boolean satisfiability. Technical Report STAN-CS-90–1302, Stanford University, February 1990.

[Nel80] Greg Nelson. Techniques for program verification. CLS 81–10, XEROX Palo Alto Research Center, June 1980.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of Association for Computing Mashinery*, 27(2):356–364, April 1980.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley, New York, 1986.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In *Fourth Conference on Computer-Aided Verification*, pages 280–921, Elounda, Greece, June 1993. Springer-Verlag.

[Sho77] Robert E. Shostack. On the SUP-INF method for proving Presburger formulas. *Journal of Association for Computing Mashinery*, 24(4):529–543, October 1977.

[Sho82] Robert E. Shostack. Deciding combinations of theories. In *Conference on Automatic Deduction*, pages 209–222, New York, June 1982. Springer-Verlag Lecture Notes in Computer Science 138.

[Sny94] W. Snyder. A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. *Journal of Symbolic Computation*, 1994. To appear.

[Söy94] Ekrem Sezer Söylemez. Automatic verification of timing properties of MMT automata. Master's thesis, Massachusetts Institute of Technology, 1994.

[VG79] Stanford Verification Group. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, March 1979.

[Voi92] Frederic Voisin. A new front-end for the Larch Prover. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch*. Springer-Verlag, July 1992.