GAZING:
A Technique for Controlling the Use of Rewrite Rules

Dave Plummer

Submitted for the degree of
PhD.

University of Edinburgh

1987

Table of Contents

List of Figures

Abstract

This thesis is concerned with the possibility of designing computer programs that can carry out logical reasoning. The problem is tackled by first defining a *logical system* which is a set of rules which may be used to make inferences, and then writing a computer program which can execute these rules.

I begin by defining two logical systems, $\mathcal{ND}$ and $\mathcal{GSC}$. These systems form the basis of the automatic theorem provers that are discussed in the remainder of the thesis. Two such theorem provers were developed at the University of Texas by a research team led by Prof. W. W. Bledsoe. These theorem provers, PROVER and IMPLY, are based on different formulations of natural deduction. The relationship between the logics of the two programs and the systems of $\mathcal{GSC}$ and $\mathcal{ND}$ is discussed.

I have developed a theorem prover, called RUT, which is based on the better ideas in both PROVER and IMPLY. RUT modifies and extends both PROVER and IMPLY.

Because of the difficulty of the problem of automatic theorem proving it is necessary to build the program so that it can determine when a particular course of action is likely to lead to a proof. This notion is completely extra-logical in the sense that there is nothing in the logical system which would enable a prover, automatic or human, to make such a decision. While the University of Texas provers have some means of choosing between different courses of action, these methods are quite weak, and often give the wrong advice. I describe *gazing*, a technique which I have developed to enable RUT to decide when the application of an inference rule is likely to lead to a proof of a given conjecture.

The use of gazing is shown to be efficient and effective. Gazing is effective since the guidance provided by gazing is useful more often than that provided by the techniques used by the University of Texas provers. Gazing is efficient in terms of the amount of search that the system must carry out to allow gazing to provide guidance. The main result is that the use of gazing will cause a theorem prover to carry out less search than might be carried out if the proof proceeded without the guidance provided by use of the technique. Thus gazing is a useful technique for an automatic theorem prover to apply.

Finally I present details of work related to that reported in this thesis, and outstanding problems which remain to be solved concerning gazing.

Some proofs that have been performed by RUT are presented in an appendix. For the purposes of comparison, the proofs are presented as they are performed both with, and without, using gazing.

Declaration

I declare that this thesis has been composed by myself, and that the work described in it is my own.

Dave Plummer

6th April 1987

Funding for this research was provided by *The Science and Engineering Research Council of Great Britain* from grant number GR/C/20826 and a studentship to the author, and by grant number DCR-8313499 from *The National Science Foundation of the United States*.

Acknowledgements

I would like to thank the many people who have helped and encouraged me in the research reported here, in particular:

- My supervisor, *Alan Bundy*, for his insightful comments and encouragement at every stage in the progress of the work. I am pleased to have been one of the many people influenced by his enthusiasm for the enterprise of building automated mathematicians.

- My assistant supervisors: *Jane Hesketh* and *Dave Schmidt*.

- The members of the *PRESS* and *DREAM projects* in the Department of Artificial Intelligence at the University of Edinburgh.

- *Woody Bledsoe* for his interest in this research, and for providing me with the opportunity to continue this research at the University of Texas.

- The members of the *ATP project* at the University of Texas.

- *Peter Mott*, *John Self* and *Kit Dodson* at the University of Lancaster, who between them are chiefly responsible for my interest in logic, AI and mathematics.

- The many people who read and commented on early drafts of this thesis: *Ernie Cohen, Jane Hesketh, Natarajan Shankar, Don Simon* and *Tie-Cheng Wang*.

- *Bernadette*, to whom this work is dedicated, *Mum* and *Dad*, *Lynda* and *Gill*, and my many friends in Edinburgh and Austin, all of whom encouraged and helped, often without realizing it.

The TOP and EMACS editors, SCRIBE and various implementations of the PROLOG programming language were used in the course of the research.

Chapter 1
Natural Deduction and Mechanical Theorem Proving

1.1. Introduction

This thesis is concerned with the possibility of designing computer programs that can carry out logical reasoning. This enterprise is a subfield of Artificial Intelligence called *automatic theorem proving*. The problem is tackled by first defining a *logical system* which is a set of rules which may be used to make inferences, and then writing a computer program which can execute these rules.

In this chapter I define two logical systems from the family of *natural deduction*. These systems will form the basis of the automatic theorem provers that are discussed in the remainder of the thesis. The two particular systems that are discussed are called $\mathcal{ND}$ and $\mathcal{GSC}$.

Chapter 2 contains descriptions of two theorem provers developed at the University of Texas by a research team led by Prof. W. W. Bledsoe. These theorem provers, PROVER and IMPLY, are based on different formulations of natural deduction. The relationship between the logics of the two programs and the systems of $\mathcal{GSC}$ and $\mathcal{ND}$ is discussed in this chapter.

Chapter 3 contains a description of RUT, a theorem prover which I have developed by modifying and extending the University of Texas provers. The logic of RUT is also a member of the natural deduction family.

The University of Texas provers have a number of features which are related to the efficient execution of the inference rules of the logical system. Because of the difficulty of the problem of automatic theorem proving it is necessary to build the program so that it can determine when a particular course of action is likely to lead to a proof. This notion is completely extra-logical in the sense that there is nothing in the logic of the system which would enable a prover, automatic or human, to make such a decision. The decision must be based on features of the problem which are not available to the logic. In chapter 4 I describe *gazing*, a

technique which I have developed to enable RUT to decide when the application of an inference rule is likely to lead to a proof of a given conjecture.

Chapter 5 contains a justification for using gazing in terms of the amount of search that the system must carry out to allow gazing to provide guidance. The main result of this chapter is that the use of gazing will cause a theorem prover to carry out less search than might be carried out if the proof proceeded without the guidance provided by use of the technique. Thus gazing is a useful technique for an automatic theorem prover to apply.

Work related to that reported in this thesis, and outstanding problems which remain to be solved concerning gazing are presented in chapter 6.

Proofs that have been performed by RUT are presented in appendix B. For the purposes of comparison, the proofs are presented as they are performed both using gazing, and without the use of gazing.

1.1.1. Overview of This Chapter

This chapter prepares the ground for what follows, providing some necessary logical preliminaries. No attempt is made to cover the material outlined here in depth as many excellent books are readily available. These are referenced in the appropriate places in the text.

Section 1.2 introduces two systems of proof: *natural deduction* ($\mathcal{ND}$), and *Gentzen sequent calculus* ($\mathcal{GSC}$). These systems will form the basis of the automatic theorem provers that are discussed in the remainder of the thesis.

In section 1.3 I will describe some techniques that have been developed in the study of automatic theorem proving, but which are not part of the underlying logic.

In section 1.4 I discuss two systems of proof: *the connection method* and *resolution*, both of which are commonly used in automatic theorem proving. The relative merits of these systems as opposed to $\mathcal{ND}$ and $\mathcal{GSC}$ are discussed in section 1.5.

1.2. Logical Preliminaries

In this section I present the logical preliminaries required in the remainder of this thesis. For a detailed discussion of the material of this section the reader is referred to [Chang & Lee 73, Tennant 78, Kleene 67].

The theorem provers which I shall consider all use the first-order predicate calculus to express the conjectures that are to be proved. This is a formal language which allows the expression of propositions unambiguously.

Definition 1: *The alphabet of predicate calculus.*

Expressions of the predicate calculus are constructed from the following alphabet:

1. Quantifiers: $\exists$, $\forall$.
2. Connectives: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$.
3. Infinitely many predicates: P_0, P_1, ...
4. Infinitely many functions: f_0, f_1, ...
5. Infinitely many variables: x_0, x_1, ...
6. Punctuation: ( ) ,

While predicates, variables and functions, must formally be members of the sets specified above, I will often use symbols from mathematics and elsewhere in these roles. These symbols can be understood as representing some member of the appropriate set. The general convention that I will follow is that any word beginning with an upper case letter stands for a predicate. A word beginning with a lower case letter near the end of the alphabet stands for a variable, and a word beginning with a lower case letter near the beginning of the alphabet stands for a function. I will use lower case Greek letters to stand for arbitrary expressions of the calculus, and upper case Greek letters to stand for sets of expressions.

Definition 2: *Terms of the calculus.*

- A variable is a *term*,
- If ϕ is a function, and τ_1, ... τ_n $(n \geq 0)$ are terms, then $\phi(\tau_1, \ldots, \tau_n)$ is a *term*. ϕ is said to be an *n-ary function*.
- Nothing else is a term.

Definition 3: A *constant* is a 0-ary function.

Definition 4: *Atomic formulae*

- If π is a predicate, and τ_1, ... τ_n $(n \geq 0)$ are terms, then $\pi(\tau_1, \ldots, \tau_n)$ is an *atomic formula*. π is said to be an *n-ary predicate*.

- $\bot$ and $\top$ are atomic formula (*false* and *true* respectively).
- Nothing else is an atomic formula.

Definition 5: *Well-Formed Formulae (wffs)*

- An atomic formula is a well-formed formula,
- If α and β are well-formed formulae, then so are:
  1. $(\alpha \wedge \beta)$, (meaning: α *and* β)
  2. $(\alpha \vee \beta)$, (meaning: α *or* β)
  3. $(\alpha \rightarrow \beta)$, (meaning: α *implies* β)
  4. $(\alpha \leftrightarrow \beta)$, (meaning: α *if and only if* β), and,
  5. $\neg \alpha$, (meaning: *not* α).
- If ξ is a variable, and α a well-formed formula, then:
  1. $(\exists \xi.\alpha)$, (meaning: *there is a* ξ *which makes* α *true*)
  2. $(\forall \xi.\alpha)$, (meaning: *all* ξ *make* α *true*).

  are also well-formed formulae.
- Nothing else is a well-formed formula.

The following definitions will be used in the remainder of this thesis:

Definition 6: The variable ξ is said to be *bound* by the quantifier that precedes it.

Definition 7: Any variable that appears in a wff and is not bound by any quantifier is said to be *free* in that wff.

Definition 8: A *sentence* is a wff that contains no free variables.

Definition 9: The *universal closure* of a wff ϕ is the wff $\forall \xi_1.\forall \xi_2 \ldots \phi$ where each free variable of ϕ appears as ξ_i for some i.

The universal closure of any wff is a sentence by definition.

Definition 10: Any atomic formula is a *literal*, and the negation of any atomic formula is also a *literal*.

In the remainder of this thesis only well-formed formulae will be considered. Unless explicitly stated, the word *formula* shall be read to mean *well-formed formula*.

The definitions above have described the way in which the class of well-formed formulae can be constructed from the alphabet of the predicate calculus. In subsections 1.2.1 and 1.2.2 I will describe the proof systems $\mathcal{ND}$ and $\mathcal{GSC}$ respectively.

1.2.1. Natural Deduction (ND)

A proof system is a set of rules which indicate how to construct proofs of formulae within the system. The rules of the system that I shall call $\mathcal{ND}$ are presented in figures 1-1 through 1-16. For a detailed discussion of these rules and their logical basis, see for example [Tennant 78].

Each rule has a number of hypothesis formulae, a conclusion formula, and an inference bar. The conclusion appears below the inference bar and the hypotheses above. The rule asserts that if the hypothesis formulae are known then the rule of inference may be applied and the conclusion formula deduced. The inference bar may be labelled by the name of the inference rule. For any connective or quantifier χ, the rule $\chi{-}I$ causes χ to be Introduced, and $\chi{-}E$ causes χ to be Eliminated.

Large Greek capital letters appear in some of the rules. These represent subproofs with the formula which appears above the subproof as one of the hypotheses and the formula which appears below the subproof as conclusion. The subproof may depend on other formulae which are not shown in the inference rule.

Any formula may be assumed as a hypothesis at any point in the proof. This means that the proof will depend on the assumed hypothesis, unless it can subsequently be shown that the proof is independent of this assumption. This occurs by the use of inference rules which allow hypotheses to be *discharged*. That a hypothesis has been discharged is indicated by writing an inference bar above the formula. The resulting proof depends only on the undischarged hypotheses of the proof. The inference bars which appear as a result of discharging an assumption are labelled with small Roman numerals. All hypotheses discharged by the same application of an inference rule and the inference bar of the rule are labelled by the same numeral.

> **Definition 11:** α^{ξ}_{τ} denotes the formula which is the same as α except that all occurrences of τ have been replaced by ξ.

$$\frac{\alpha \wedge \beta}{\alpha} \qquad \frac{\alpha \wedge \beta}{\beta}$$

Figure 1-1: $\wedge$-E

$$\frac{\alpha \qquad \beta}{\alpha \wedge \beta}$$

Figure 1-2: $\wedge$-I

$$\frac{\alpha}{\alpha \vee \beta} \qquad \frac{\beta}{\alpha \vee \beta}$$

Figure 1-3: $\vee$-I

$$\frac{\alpha \vee \beta \qquad \begin{matrix}\overline{\alpha}^{(i)} \\ \Pi \\ \chi\end{matrix} \qquad \begin{matrix}\overline{\beta}^{(i)} \\ \Sigma \\ \chi\end{matrix}}{\chi}\,(i)$$

Figure 1-4: $\vee$-E

$$\frac{\alpha \qquad \alpha \rightarrow \beta}{\beta}$$

Figure 1-5: $\rightarrow$-E

$$\frac{\begin{matrix}\overline{\alpha}^{(i)} \\ \Pi \\ \chi\end{matrix}}{\alpha \rightarrow \chi}\,(i)$$

Figure 1-6: $\rightarrow$-I

$$\frac{\alpha \rightarrow \beta \qquad \beta \rightarrow \alpha}{\alpha \leftrightarrow \beta}$$

Figure 1-7: $\leftrightarrow$-I

$$\frac{\alpha \leftrightarrow \beta}{\alpha \rightarrow \beta} \qquad \frac{\alpha \leftrightarrow \beta}{\beta \rightarrow \alpha}$$

Figure 1-8: $\leftrightarrow$-E

$$\frac{\neg\ \alpha \qquad \alpha}{\bot}$$

Figure 1-9: $\neg$-E

$$\begin{array}{c} \overline{\alpha}^{(i)} \\ \Pi \\ \bot \\ \hline \neg\ \alpha \end{array}^{(i)}$$

Figure 1-10: $\neg$-I

$$\frac{\forall\ \xi.\alpha}{\alpha^{\tau}_{\xi}}$$

Figure 1-11: $\forall$-E

$$\frac{\alpha}{\forall\ \xi.\alpha^{\xi}_{\tau}}$$

Condition: τ is a constant which does not appear in any assumption used in deriving $\forall\ \xi.\alpha$.

Figure 1-12: $\forall$-I

$$\frac{\exists\ \xi.\alpha \qquad \begin{array}{c} \overline{\alpha^{\tau}_{\xi}}^{(i)} \\ \Pi \\ \chi \end{array}}{\chi}^{(i)}$$

Condition: τ does not appear in α, χ or any undischarged hypotheses of Π

Figure 1-13: $\exists$-E

$$\frac{\alpha}{\exists\ \xi.\alpha^{\xi}_{\tau}}$$

Figure 1-14: $\exists$-I

$$\frac{\bot}{\alpha}$$

Figure 1-15: Absurdity Rule

$$\frac{}{\alpha \vee \neg \alpha}$$

Figure 1-16: Law of Excluded Middle

Figures 1-15 and 1-16 show two rules which do not introduce or eliminate a connective. For this reason they deserve special comment. The *absurdity* rule 1-15 indicates that if we have as a hypothesis the atom $\bot$, then we may deduce any consequence that we like. This may be read as saying that if we are able to prove $\bot$ then the world is so crazy that we may assume that anything is true.

The *law of excluded middle* given in figure 1-16 is not an inference rule, since it has no hypotheses. The rule consists of a formula which follows from no hypotheses, that is, the formula is always true. Such a formula is called an *axiom*. The meaning of the law of excluded middle is that every formula is either true or false (any middle possibility is excluded).

The rules of natural deduction presented in figures 1-1 to 1-16 describe the valid inferences of the proof system. A proof in the system is constructed from these inferences by using the conclusion of one inference as the hypothesis of another. The unique formula which is not used as the hypothesis of some other inference is called the *conclusion of the proof*. The undischarged hypotheses of the proof are the hypotheses.

As an example of the use of the $\mathcal{ND}$ system consider the goal of proving C from hypotheses $A \vee B$, $A \rightarrow C$ and $B \rightarrow C$. The hypothesis formulae may appear as undischarged hypotheses of the proof. I begin by noting that one of the hypotheses is a disjunction, and that the rule $\vee{-}E$ (figure 1-4) is likely to be of use. Instantiating $\vee{-}E$ with the formula $A \vee B$ gives the partial proof of figure 1-17.

$$\frac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \Pi \\ \delta\end{array} \qquad \begin{array}{c}\overline{B}\\ \Sigma \\ \delta\end{array}}{\delta}$$

Figure 1-17: First Partial Proof of the Conjecture

The subproofs, Π and Σ, which use A and B as hypotheses respectively, must both prove the same conclusion, and this will be the conclusion of the $\vee{-}E$ rule. Using $\rightarrow{-}E$ with A and $A \rightarrow C$ we can deduce C. Similarly with B and $B \rightarrow C$ we can also deduce C by the same rule. These two applications of $\rightarrow{-}E$ complete the proof of figure 1-18.

$$\dfrac{A \vee B \qquad \dfrac{\overline{A} \qquad A \rightarrow C}{C} \qquad \dfrac{\overline{B} \qquad B \rightarrow C}{C}}{C}$$

Figure 1-18: Completed Proof of the Conjecture

Notice that the conclusion of the proof is C as desired, and the only undischarged hypotheses are the hypotheses of the conjecture.

1.2.2. Gentzen Sequent Calculus (GSC)

The Gentzen sequent calculus ($\mathcal{GSC}$) has basically the same rules of inference as $\mathcal{ND}$. However rather than being phrased in terms of formulae as the rules of $\mathcal{ND}$ are, the rules of $\mathcal{GSC}$ are phrased in terms of sequents.

Definition 12: A *sequent* is written $\Gamma \vdash \Theta$, where Γ and Θ are sets of formulae.

Definition 13: The symbol, $\vdash$, is called the *turnstile*.

$\Gamma \vdash \Theta$ means that there is a proof of some formula in Θ from some subset of Γ. Clearly there is a simple mapping between the proofs of $\mathcal{ND}$ and the sequent representation: If there is an $\mathcal{ND}$ proof of β with undischarged hypotheses $\alpha_1, \ldots \alpha_n$, then we write $\{\alpha_1, \ldots, \alpha_n\} \vdash \{\beta\}$. For a more detailed development of $\mathcal{GSC}$ the reader is referred to [Kleene 67].

The rules of $\mathcal{GSC}$ are given in figures 1-19 through 1-34. Unlike the rules of $\mathcal{ND}$, the rules of $\mathcal{GSC}$ always *introduce* a connective or quantifier. There is one rule for introduction into a formula of the hypothesis, and another for introduction into the conclusion. If a rule introduces a connective or quantifier χ into the hypothesis it is referred to as $\chi{-}\vdash$, and into the conclusion as $\vdash{-}\chi$.

$$\frac{\Delta \vdash \Sigma \cup \{\alpha\} \qquad \Gamma \vdash \Theta \cup \{\beta\}}{\Delta \cup \Gamma \vdash \{\alpha \wedge \beta\} \cup \Sigma \cup \Theta}$$

Figure 1-19: $\vdash$-$\wedge$

$$\frac{\{\alpha, \beta\} \cup \Gamma \vdash \Theta}{\{\alpha \wedge \beta\} \cup \Gamma \vdash \Theta}$$

Figure 1-20: $\wedge$-$\vdash$

$$\frac{\Delta \cup \{\alpha\} \vdash \Gamma \qquad \Theta \cup \{\beta\} \vdash \Gamma}{\Theta \cup \Delta \cup \{\alpha \vee \beta\} \vdash \Gamma}$$

Figure 1-21: $\vee$-$\vdash$

$$\frac{\Delta \vdash \Gamma \cup \{\alpha,\beta\}}{\Delta \vdash \Gamma \cup \{\alpha \vee \beta\}}$$

Figure 1-22: $\vdash$-$\vee$

$$\frac{\Delta \cup \{\alpha\} \vdash \Gamma \cup \{\beta\}}{\Delta \vdash \Gamma \cup \{\alpha \rightarrow \beta\}}$$

Figure 1-23: $\vdash$-$\rightarrow$

$$\frac{\Gamma \vdash \Theta \cup \{\alpha\} \qquad \{\beta\} \cup \Gamma \vdash \Theta}{\{\alpha \rightarrow \beta\} \cup \Gamma \vdash \Theta}$$

Figure 1-24: $\rightarrow$-$\vdash$

$$\frac{\Delta \cup \{\alpha\} \vdash \{\beta\} \cup \Gamma \qquad \Theta \cup \{\beta\} \vdash \{\alpha\} \cup \Pi}{\Delta \cup \Theta \vdash \Gamma \cup \Pi \cup \{\alpha \leftrightarrow \beta\}}$$

Figure 1-25: $\vdash$-$\leftrightarrow$

$$\frac{\{\alpha,\beta\} \cup \Gamma \vdash \Theta \qquad \Delta \vdash \Sigma \cup \{\alpha,\beta\}}{\Gamma \cup \Delta \cup \{\alpha \leftrightarrow \beta\} \vdash \Theta \cup \Sigma}$$

Figure 1-26: $\leftrightarrow$-$\vdash$

$$\frac{\Delta \cup \{\alpha\} \vdash \Theta}{\Delta \vdash \Theta \cup \{\neg \alpha\}}$$

Figure 1-27: $\vdash\text{-}\neg$

$$\frac{\Delta \vdash \{\alpha\} \cup \Theta}{\Delta \cup \{\neg \alpha\} \vdash \Theta}$$

Figure 1-28: $\neg\text{-}\vdash$

$$\frac{\Gamma \vdash \Theta \cup \{\alpha\}}{\Gamma \vdash \Theta \cup \{\forall \xi.\alpha^{\xi}_{\tau}\}}$$

Condition: τ does not appear free in $\Gamma \vdash \Theta \cup \{\forall \xi.\alpha^{\xi}_{\tau}\}$

Figure 1-29: $\vdash\text{-}\forall$

$$\frac{\{\alpha, \forall \xi.\alpha\} \cup \Gamma \vdash \Theta}{\{\forall \xi.\alpha\} \cup \Gamma \vdash \Theta}$$

Figure 1-30: $\forall\text{-}\vdash$

$$\frac{\Gamma \vdash \{\alpha\} \cup \Theta}{\Gamma \vdash \{\exists \xi.\alpha\} \cup \Theta}$$

Figure 1-31: $\vdash\text{-}\exists$

$$\frac{\Gamma \cup \{\alpha\} \vdash \Theta}{\Gamma \cup \{\exists \xi.\alpha^{\xi}_{\tau}\} \vdash \Theta}$$

Condition: τ does not appear free in $\Gamma \cup \{\exists \xi.\alpha^{\xi}_{\tau}\} \vdash \Theta$

Figure 1-32: $\exists\text{-}\vdash$

$$\frac{}{\Gamma \cup \{\bot\} \vdash \Sigma}$$

Figure 1-33: Absurdity rule for $\mathcal{GSC}$

$$\frac{}{\Gamma \vdash \Sigma \cup \{\alpha, \neg \alpha\}}$$

Figure 1-34: Law of Excluded Middle for $\mathcal{GSC}$

$$\frac{}{\Gamma \cup \{\alpha\} \vdash \Delta \cup \{\alpha\}}$$

Figure 1-35: Matching Rule for $\mathcal{GSC}$

Figure 1-35 shows that when the same formula appears in both the hypothesis and conclusion sets then the sequent is true.

The rules of $\mathcal{GSC}$ are more expressive than those of $\mathcal{ND}$ since all of the undischarged hypotheses of the proof are represented in the set of formulae that appears to the left of the turnstile symbol. The process of discharging assumptions is reduced to that of eliminating the formula from the hypothesis set. The sequent representation also can represent the idea that the conclusion of the proof is some member of the conclusion set of the sequent, while $\mathcal{ND}$ proofs have a unique conclusion. In $\mathcal{ND}$ proofs the only formulae that appear in the proof as hypotheses are those that are used in the proof. The sequent representation allows that hypotheses that are never used may be members of the hypothesis set. Unused hypotheses and unproved conclusions are unaffected by any application of an inference rule.

We shall see in this thesis that the more expressive representation of the goal to be proved that is available in $\mathcal{GSC}$ is more useful to an automatic prover. This is because the automatic prover cannot be sure which of the hypotheses of the proof are necessary in the proof, or which of the possible conclusions may be proved. The ability to "carry" hypotheses which are not useful without incurring the penalty of having to attempt to use them, is a useful facility.

In this section I have introduced the logical preliminaries required for the remainder of this thesis. I have described the first-order predicate calculus, and two natural deduction systems of proof which I call $\mathcal{ND}$ and $\mathcal{GSC}$.

In the remainder of this thesis I will use the more powerful sequent notation to describe goals and theorems. It should be understood that this notation can be translated readily into the notation of $\mathcal{ND}$ if desirable.

1.3. Automating Natural Deduction

There are many implementations of mechanical theorem provers which use a natural deduction proof system, see for example: [Bledsoe & Tyson 75a, Bledsoe & Tyson 78, Bledsoe 83, Nevins 74, Nevins 75a, Pastre 77, Reiter 76, Reiter 73, Cvetkovic & Pevac 83]. Most of these represent the goals of the system as sequents and so bear more resemblance to a $\mathcal{GSC}$-based system than to $\mathcal{ND}$. However there are some features of natural deduction systems that do not lend themselves to automation. One of the advantages of the sequent representation of the proof, is that the proof can be seen as a set of subproofs, and that subproofs are "glued" together as inferences are made, this is in contrast to viewing the proof as a series of steps from the hypotheses to the conclusion.

> **Definition 14:** A sequent that the theorem prover is to prove is called the *goal*, or sometimes *conjecture*, and is written: $\Gamma \vdash^{?} \Theta$ where Γ and Θ are sets of formulae.

The superscript $?$ indicates that it is not known whether the sequent is true or false.

When the hypothesis or conclusion of a sequent or goal is a singleton set I will simply write the formula in place of the set. Thus the sequent $\{\alpha\} \vdash \{\beta\}$ may be written $\alpha \vdash \beta$.

1.3.1. Forward and Backward Inference

One of the problems with the rules of $\mathcal{ND}$ and $\mathcal{GSC}$ is that they are too powerful. For example consider $\vee$–$\vdash$ (figure 1-21) which allows the deduction of $\alpha \vee \beta$, from the hypothesis α. The problem with this rule is that there is no restriction on the formula that can be used as the hypothesis. Giving this rule to a theorem prover would be disastrous: for every formula that the theorem prover was given as a hypothesis, this rule would be applicable. Moreover, the rule would be applicable in an infinite number of ways, as the prover could invent a different β for each application of the rule. The prover might produce the following proof when given the hypothesis α.

$$\dfrac{\dfrac{\dfrac{\alpha \vdash \sigma \qquad \alpha \vdash \sigma}{\alpha \vee \alpha \vdash \sigma} \qquad \alpha \vdash \sigma}{\{\alpha \vee (\alpha \vee \alpha)\} \vdash \sigma} \qquad \alpha \vdash \sigma}{\{\alpha \vee (\alpha \vee (\alpha \vee \alpha))\} \vdash \sigma}$$

$$\vdots$$

Notice that nothing is gained by the application of the rule in this way since $\alpha \vee \alpha$ is logically equivalent to α. Clearly giving a theorem prover such a rule would be a bad idea. However *not* giving the theorem prover this rule would also be a bad idea, since the prover would then be unable to prove some theorems. Consider the goal: $\alpha \vdash^{?} \alpha \vee \beta$. It is necessary for the prover to use $\vee$–$\vdash$ in this case to prove the goal.

One answer to the problem is to distinguish between two different types of inference: forward and backward. *Forward inference* is inference from one hypothesis to another, not necessarily making reference to the desired conclusion. The application of $\vee$–$\vdash$ above is an example of this. *Backward inference* is inference from one conclusion to another. For example, the step from the above goal to $\alpha \vdash \alpha$ is an example of $\vee$–$\vdash$ applied backward. In this step, the disjunctive conclusion has been replaced by one of its disjuncts. The interpretation of this step is: "to show $\alpha \vee \beta$ it is sufficient to show α". Since the completed proof does not indicate the order or direction of deduction, this backward use of the rule is indistinguishable from the inference from α to $\alpha \vee \beta$. The choice of application of this rule is determined by the conclusion that we hope to reach. For this reason backward inference is sometimes called *goal-directed*. Forward inference is correspondingly called *data-driven*, since the deductions that are made depend only on the known hypotheses, or data.

Most theorem provers are able to use the introduction rules of *ND* exclusively in backward inference mode, and the elimination rule exclusively forwards. Similarly the rules of *GSC* are usually used backward. It is important to realize that while the rules are constrained to be used in only one direction, the system has not lost any power. All theorems of *ND* and *GSC* are provable with the directed rules. The constraint of using the rules in a prespecified direction is merely *control* of the rules of the system. This control prevents many redundant inferences from being carried out.

One of the effects of the split between forward and backward inference is that the proof appears to proceed in a somewhat haphazard manner, as the inferences are made first forward and then backward. The normal view of proof that is presented is as a gradual working forward from the hypotheses until the desired conclusion is reached. The effect of the mixture of forward and backward inference is that the proof appears as the construction of many partial proofs, which are eventually to be "glued" together somewhere in the middle. This view of the theorem proving process is surely much nearer the truth. Written proofs, like so much of mathematics, are rational reconstructions of the actual activity of mathematics, and as such give little insight to the reader about how to actually do proofs.

In this subsection I have described forward and backward inference, and shown how the automation of natural deduction requires that some of the inference rules be used in a data-driven mode, and others in a goal-directed way. This leads to theorem provers which appear to carry out proofs in a more haphazard way that we would have expected.

1.3.2. Completeness

We would like to design a theorem prover which could decide whether any conjecture is a theorem or a non-theorem. Such a program would be a *decision procedure*. A result by Church [Church 36] based on pioneering work by Godel [Godel 31] shows that no decision procedures exist for first-order logic. A weaker notion is that of a complete prover.

> **Definition 15:** An automatic theorem prover is said to be *complete* if every theorem may, in principle, be proved by the program.

Complete provers exist for first-order logic.

Any prover which is guaranteed to terminate the search for a proof is not complete. This is because, if the prover were complete it would find a proof of all theorems, and if it terminated after a finite number of steps without finding a proof, the conjecture would be shown to be a non-theorem. Thus, the prover would be a decision procedure for theorems, which we know to be impossible.

Completeness is a double-edged property. If we have a complete prover we can be sure that if the conjecture is a theorem then we will be able to find a proof. This is clearly a desirable state of affairs. However completeness inevitably leaves open the possibility of infinite proof attempts on some non-theorems. Some authors prefer to have an incomplete prover which will fail quickly on non-theorems, and succeed on many theorems, in preference to a complete prover which may search infinite paths. The theorem provers which I shall discuss in this thesis are not complete.

1.3.3. Skolemization and Unification

As a result of the mixture of forward and backward inference in the theorem prover, the system does not have a complete proof until the whole theorem is proved. The system is, as observed above, manipulating a number of partial proofs, in order to "glue" them together into a complete proof of the goal. One of the problems that this approach causes is that the restrictions on the quantifier rules, described in 1-32 and 1-29 above, cannot be checked. The restrictions ensure that the objects that are introduced when quantifiers are eliminated or introduced are arbitrary. But checking that the restrictions are met requires that the whole proof above the application of the rule is available for inspection, and as we have observed, this is not the case in general. One of the ways commonly adopted to avoid this problem is the technique of *Skolemization*.

Skolemization

Skolemization replaces the quantifier rules by eliminating the quantifiers from the formula before the proof begins. The idea is that the original conjecture, containing quantifiers, is skolemized before the proof proceeds. All of the quantifiers are eliminated, and the bound variables of the formula are replaced by new terms, which record in their structure the dependencies between the quantifiers in the original formula. The result is a new formula, which is provable under the same conditions as the original formula.

In order to describe the process of skolemization in detail, it is necessary to introduce the notion of *polarity*.

Definition 16: The *polarity* of a formula is given by the following rules:

1. A formula in the hypothesis (conclusion) of a sequent has polarity - (+),
2. If a formula has polarity + (-) and is of the form:
   - $(\alpha \wedge \beta)$, then α and β have polarity + (-),
   - $(\alpha \vee \beta)$, then α and β have polarity + (-),
   - $(\alpha \rightarrow \beta)$, then α has polarity - (+) and β has polarity + (-),

- $(\neg \alpha)$, then α has polarity - (+),
- $(\alpha \leftrightarrow \beta)$, then the polarities of α and β are undefined but opposite to each other.
- $(\forall x.\alpha)$, then α has polarity + (-), and $\forall$ is called a positive (negative) quantifier.
- $(\exists x.\alpha)$, then α has polarity + (-), and $\exists$ is called a negative (positive) quantifier.

Using these rules we can assign each subformula of a given formula a unique polarity, providing that the formula contains no subformula involving the $\leftrightarrow$ connective. The reason that the polarity of biconditionals is undefined can be easily seen. Consider that $\alpha \leftrightarrow \beta$ is equivalent to $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$. By the polarity rules the latter formula has both α and β appearing with both polarity + and -. The polarity of α and β in the original formula can therefore not be determined.

Definition 17: The *scope* of a quantifier χ in the formula $\chi\tau.\alpha$, is the formula α.

The process of skolemizing a formula has three steps.

1. Using the fact that $\alpha \leftrightarrow \beta$ is equivalent to $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$, replace all occurrences of biconditional connectives with implication arrows.
2. Assign each subformula a polarity (this is now possible since the $\leftrightarrow$ symbols have been eliminated).
3. Delete each quantifier and variable of quantification. If the variable is bound by a positive quantifier then replace it throughout the formula by a new term $sf(\xi_1, \ldots, \xi_n)$, where sf is a new function symbol (a *s*kolem *f*unction), and the ξ_i are those variables of the whole formula which are bound by negative quantifiers whose scope includes the quantifier just eliminated.

The technique of skolemization is well-known in the theorem proving literature. For a more detailed discussion see [Bundy 83a, Chang & Lee 73].

Notice that the above rules cause some of the variables in the formula to be replaced by skolem terms, and others to be left as variables. The distinction between the two types of variable is made on the polarity of their quantifier in the formula. The variables that are left in the formula represent objects that we know nothing about, but we might be able to determine their identity in the course of the proof. If we do we speak of *binding* the variable to a term. This is done in the unification process, which is described in the next subsection. The skolem terms which are introduced by the skolemization process represent specific objects, whose exact identity depends on the arguments to the term. If we are constructing a

skolem term, that depends on no variables, then we need a function of arity 0, which is just a new constant. We will represent these new constants by the symbol sc subscripted to distinguish distinct constants. Consider the sequents (A) and (B)*.

$$\vdash \forall x.\exists y.\ y < x \qquad (A)$$

$$\vdash \exists y.\forall x.\ y < x \qquad (B)$$

The formula (A) asserts that for any number x, there is another number y that is smaller than x. This is true in real arithmetic. Notice that y depends on x, that is the choice of y must be made after the choice of x. (B), on the other hand, asserts that there is a number y that is smaller than every x, which is false. The skolemized forms of (A) and (B) are the formulae (C) and (D) respectively.

$$\vdash y < sc_0 \qquad (C)$$

$$\vdash y < sf(y) \qquad (D)$$

Notice how (C) clearly shows that we are asked to find a y which is less than a given x by replacing x by a constant: sc_0, and leaving y as a variable. In the formula (D) the variable x has been replaced by a skolem function of y, indicating that the choice of this quantity has to depend on the choice of y.

The point of eliminating the quantifiers by skolemizing the formula is that the quantifier elimination rules, and the problems that the conditions on their use pose can then be discarded. These rules disappear from the logical system, in favour of the skolemization rule. It is known that this substitution leaves the consistency of the logical system unchanged. That is, if τ is any theorem of $\mathcal{ND}$ then the skolemized form of τ is a theorem of $\mathcal{ND}$ without the quantifier rules.

Unification

One further problem with manipulating partial proofs, is the eventual "gluing" together of the partial proofs. The idea is that once the conclusion of the sequent has been manipulated into a particular form, and the hypothesis has been made into the same form, then the proof is complete. This manifests itself in the $\mathcal{ND}$ system as a partial proof of the conclusion from an intermediate set of hypotheses, and another partial proof, of this intermediate set of hypotheses from the original hypotheses of the theorem. This situation would be quite hard to detect. In $\mathcal{GSC}$ the completion of the proof is signalled by the appearance of the same

---

*Formally speaking these formulae are not well-formed. Two common abuses of notation have been perpetrated: first the predicate $<$, is written between its arguments. This is very common practice when dealing with binary predicates and functions. Consider, for example, $=$, $\in$, $\cup$, and $\cap$. Second, we have used the symbol $<$ to stand for some P_i from the formal language. We will allow further such abuses of notation to pass without comment.

formula in the hypothesis set and in the conclusion set. This is much easier to detect, and is usually performed by a process called *unification*.

Unification matches one formula with another. It is not necessary that the formulae be identical, merely that they can be made identical by the replacement of some variables in the formulae, with some terms, so that the result of making this substitution throughout each formula is the same. Consider the example below:

$P(x) \wedge Q(a)$

$P(b) \wedge Q(y)$

These two formulae will unify (match) since if x were replaced by b, and y by a throughout the two formulae, then the result would be the same formula in each case.

$P(x) \wedge (Q(a) \vee R(x,z))$

$P(b) \wedge (Q(y) \vee R(b,w))$

The formulae above also unify. Again it is necessary to replace x by b and y by a, also it is necessary to replace z by w or vice versa. The replacement can be made either way round, since both w and z are variables. In fact another way of making the formulae the same would be to replace both w and z by any term we like (the same term in both cases), but this would be redundant. We seek to make the formulae identical, but do not wish to make more assumptions about the appropriate substitution for variables than is necessary. This is because, for example, w might appear in other formulae elsewhere in the proof, and when we attempt to match the other formula we might require w to be matched to a. If we had left it as a variable, then this would be possible. If we had unnecessarily replaced w by b, then the match would no longer be possible.

The formulae below cannot be unified since to match the literals in predicate P, x must be bound to b, but to match the R literals x must be bound to a.

$P(x) \wedge (Q(a) \vee R(x,z))$

$P(b) \wedge (Q(y) \vee R(a,w))$

Unification is a process which matches two formulae. The formulae unify if the variables in the formulae can be substituted for terms such that the two formulae become identical. This substitution is called *binding* the variables. A substitution is therefore a set of pairs written α/β, where β is a variable and α is the term which is substituted for it. Since each variable may be substituted by only one value, we seek to finding a matching substitution, called a *unifier*, which binds only those variables that must be bound to make the proof go through.

Such a unifier is called a *most general unifier*. The unification routine guarantees that there will be such a (most general) unifier for two formulae that are unifiable.

> **Definition 18:** $\alpha\theta$ denotes the application of substitution θ to formula α. $\alpha\theta$ is the formula α except that the variables in α are simultaneously replaced by the term to which they are bound in the substitution.

So, if θ is the substitution, $\{\tau_1/\xi_1, \tau_2/\xi_2, \ldots, \tau_n/\xi_n\}$, then $\alpha\theta$ is $\alpha^{\tau_1 \tau_2 \ldots \tau_n}_{\xi_1 \xi_2 \ldots \xi_n}$

Example: If θ is the substitution: $\{a/x, b/y, w/z\}$ and α the formula: $P(x,y,w,z)$, then $\alpha\theta$ is the formula: $P(a,b,w,w)$.

The details of the unification routine are presented below.

> **Definition 19:** A *disagreement pair* for two formulae is a pair of subexpressions of the formulae which occur in the same position in the formulae and are different.

Example: $<b,x>$ and $<x,y>$ are the disagreement pairs of formulae $P(x,a,y)$ and $P(b,a,x)$.

To unify two formulae α and β given a substitution θ.

1. Find a disagreement pair $<\tau_1, \tau_2>$ of α and β. If no such pair exists then the formulae unify with unifier θ.
2. If τ_1 is a variable which does not occur in τ_2 then unify $\alpha\{\tau_2/\tau_1\}$ with $\beta\{\tau_2/\tau_1\}$ given the substitution $\theta \cup \{\tau_2/\tau_1\}$.
3. If τ_2 is a variable which does not occur in τ_1 then unify $\alpha\{\tau_1/\tau_2\}$ with $\beta\{\tau_1/\tau_2\}$ given the substitution $\theta \cup \{\tau_1/\tau_2\}$.
4. Otherwise fail.

The restriction that a variable and term do not unify if the variable occurs within the term is called the *occurs-check*, and has the same effect as the conditions on the quantifier rules. This, coupled with the fact that skolem functions contain the variables on which they depend, ensures that these conditions cannot be violated. The technique of unification is standard, and is discussed in the theorem-proving literature. Refer to [Bundy 83a, Robinson 65, Chang & Lee 73] for details.

1.3.4. Rewriting

When proving theorems, it is often convenient to abbreviate complex formulae by simpler ones. This is achieved by stating a definition.

> **Definition 20:** A *definition* is an equivalence in which one side is an atomic formula. The atomic formula must contain a new predicate or function symbol which is the *defined term* , while the other side of the equivalence may contain only terms already introduced.

For example, (1) allows the theorem prover to abbreviate the quantified implication by a simple literal involving the predicate $\subseteq$. The meaning of the complex formula, is that x is a subset of y, and since the definition is an equivalence the simpler formula has the same meaning.

$$x \subseteq y \leftrightarrow \forall z.\, (z \in x \rightarrow z \in y) \tag{1}$$

Conjectures involving these defined terms can be stated and proved. Sometimes the proof of the conjecture will depend on the definition of the term, but other conjectures can be proved without reference to these. For example, the conjecture (E) can be proved without reference to some property of the predicate $\subseteq$ whereas (F) cannot. Notice that it may be possible to reference some fact about the symbol $\subseteq$, not necessarily the definition, to prove (F).

$$\{b \subseteq a,\, a \subseteq b\} \;\vdash^{?}\; a \subseteq b \tag{E}$$

$$\{a \subseteq b,\, b \subseteq c\} \;\vdash^{?}\; a \subseteq c \tag{F}$$

Equally useful is the ability to use previously proved theorems as additional hypotheses for a proof. For example, suppose that $\Pi \vdash \alpha$ has been proved, and that the prover has been given the conjecture $\Pi \vdash^{?} \alpha \wedge \beta$. Clearly all the prover should have to do is show, $\Pi \vdash^{?} \beta$, and the proof would be complete.

> **Definition 21:** Facts that have already been proved and that are useful in the proof of other conjectures are called *lemmas*.

The usefulness of a given definition or lemma in the proof of a particular conjecture is not always easy to spot. One approach to proving theorems would be to conjoin all of the definitions and lemmas to the hypotheses of the conjecture, and to prove this new conjecture. To do this explicitly would allow the theorem prover to make many inferences, most of which would be redundant.

The replacement of formulae by their definitions is justified by a special case of $\rightarrow\!-\!\vdash$. We consider the database to be a set of implications of the form $\alpha \rightarrow \beta$. These implications are

all hypotheses of the theorem, and so $\rightarrow\!\!-\!\!\vdash$ may be used backward at any time. When a formula which matches α occurs in Γ, the left subproof suggested by the rule: $\Gamma \vdash \Theta \cup \{\alpha\}$ is trivially solved (since α appears in both hypothesis and conclusion). In the right subgoal, we may introduce β as a new hypothesis.

$$\frac{\{\alpha, \alpha \rightarrow \beta\} \vdash \Pi}{\{\alpha, \beta, \alpha \rightarrow \beta\} \vdash \Pi}$$

We will regard the inference of figure 1-36 as an abbreviation of a proof of the form above. The inference bar will be labelled $\Rightarrow$, indicating the application of the *rewriting* rule of inference, and the implication that is used will be left implicit.

$$\frac{\{\alpha\} \vdash \Pi}{\{\beta, \alpha\} \vdash \Pi}$$

Figure 1-36: The Abbreviated Representation of a Rewriting Step

Formulae that are used to rewrite other formulae are called *rewrite rules*. There are three types of rewrite rules corresponding to implications, equivalences and equalities. An equality is a literal with predicate $=$ of arity 2. This distinguished predicate has special properties, which are discussed in chapter 2. In particular, if two terms are equal, then any property of one is true of the other, and thus whenever one occurs it could be replaced by the other without altering the truth of the formula. This is the basis for using equalities as rewriting steps. Rewrite rules derived from these three types of formulae are written:

1. $\alpha \Rightarrow \beta$
2. $\alpha \Leftrightarrow \beta$
3. $\tau == \sigma$

respectively. Definitions must be equivalences or equalities and so give rise to rewrite rules of types 2 and 3. Any other formula can be used to rewrite formulae and may give rise to any of the three types of rule.

Here are examples of rewrite rules which are taken from the theory of sets.

$$x \subseteq y \Leftrightarrow \forall z.(z \in x \rightarrow z \in y) \qquad (i)$$

$$x \subseteq y \wedge y \subseteq z \Rightarrow x \subseteq z \qquad (ii)$$

$$x \in \emptyset \Rightarrow \bot \qquad (iii)$$

$$x \cap (y \cap z) == (x \cap y) \cap z \qquad (iv)$$

It is usual to orient rewrite rules which arise from equivalences or equalities in one direction, since enabling the prover to use the rule in both directions could lead to the prover cycling, first rewriting P to Q and then back to P.

Definition 22: I is called the *input side* of a rewrite rule $I \Rightarrow O$.

Definition 23: O is called the *output side* of a rewrite rule $I \Rightarrow O$.

If a rewrite rule introduces a formula which contains quantifiers, as (i) does, then the introduced formula must be skolemized before replacing the original formula. This avoids the possible introduction of quantifiers into the proof, despite their elimination by skolemization before the proof started. The skolemization must occur as if the definition had been present as an additional hypothesis in the statement of the conjecture. As an implementation detail, the rewrite rule is usually stored skolemized both as a hypothesis and conclusion, and then the appropriate form could be used when performing the rewriting step.

1.4. Alternative Logical Systems

In this section I briefly present two logical systems commonly used in automatic theorem proving: *resolution* [Robinson 65] and *the connection method* [Bibel 81a, Bibel 82a]. These systems are more commonly used in mechanical theorem proving than $\mathcal{ND}$ or $\mathcal{GSC}$ based systems as they are more efficient for a machine to use. This is simultaneously a strength and weakness, as I shall describe in the next section.

1.4.1. Resolution

The most common logical system used in mechanical theorem proving is *resolution* [Robinson 65]. This system has a single inference rule, which is shown, in $\mathcal{ND}$ notation in figure 1-37.

$$\frac{\alpha \vee \beta_1 \vee \beta_2 \vee \ \ldots \ \vee \beta_n \qquad \delta \vee \neg \chi_1 \vee \neg \chi_2 \vee \ \ldots \ \vee \neg \chi_m}{(\alpha \vee \delta)\theta}$$

Where θ is the most general unifier of all the β_i and χ_i.

Figure 1-37: The Resolution Rule

Resolution based systems are *refutation* systems. That is, when given a conjecture to prove, the conjecture is negated and then $\perp$ is deduced. The resolution rule is sufficient to refute the negation of all theorems provided that the theorem is first transformed by a number of standard rewriting steps. Rewriting a formula into the required form involves eliminating some of the connectives, and reorganizing the remaining connectives into a standard configuration called *clausal-form*. Initially the formula is negated.

The elimination of connectives can be performed in two steps:

1. First the $\leftrightarrow$ connective is eliminated by means of the rewrite rule $\alpha \leftrightarrow \beta \Leftrightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

2. At this point the formula is skolemized to eliminate quantifiers.

3. The $\rightarrow$ connective is then eliminated by means of the rewrite rule $\alpha \rightarrow \beta \Leftrightarrow \neg \alpha \vee \beta$

The remaining connectives: $\wedge$, $\vee$, and $\neg$ must be reorganized using the following rules.

1. The negation symbols must be distributed so that they dominate only atomic formulae. This may be achieved by repeated use of the rewrite rules:

   - $\neg \neg \alpha \Leftrightarrow \alpha$
   - $\neg (\alpha \wedge \beta) \Leftrightarrow \neg \alpha \vee \neg \beta$
   - $\neg (\alpha \vee \beta) \Leftrightarrow \neg \alpha \wedge \neg \beta$

2. The conjunction symbols must be distributed so that no disjunction dominates them. This is achieved by repeated use of the following rewrite rules:

   - $\alpha \vee (\beta \wedge \delta) \Leftrightarrow (\alpha \vee \beta) \wedge (\alpha \vee \delta)$
   - $(\alpha \wedge \beta) \vee \delta \Leftrightarrow (\alpha \vee \delta) \wedge (\beta \vee \delta)$

Finally, the formula is split into *clauses* each clause containing only the disjunction of some literals.

To provide a proof, the system searches through the clauses until it finds a pair of clauses which can serve as the hypotheses of the resolution rule, that is any two clauses which contain atoms which unify and the atom appears negated in exactly one of the clauses. The derived clause is added to the set of clauses, and the process repeated until the clause which contains no literals (the *empty clause*) is produced. The empty clause is significant as it represents the atomic formula $\perp$ which is always false. Any clause is a disjunction of some number of literals, and so the literal $\perp$ can be added to any clause without altering its truth value. The clause which just contains $\perp$ is false, and therefore the empty clause must also be false.

The resolution rule has been augmented in many ways. These augmentations may be categorized into two classes: *strategies* and *extensions*.

- Strategies limit the clauses that may be considered for resolution, for example: *Linear Input resolution* is a resolution strategy in which the clause which has just been derived is used as one of the parents of the next resolution, and the other parent must be a clause from the statement of the conjecture (rather than a derived clause).

- Extensions, allow many inference steps to be collapsed into a single step, or additional inferences to be made. A simple extension allows many atoms to be matched at a single step, thus carrying out many applications of the simple inference rule at once. Another example is *paramodulation*, which allows the

resolution rule to treat equality literals specially, and allow the substitution of identities.

1.4.2. The Connection Method

In this section I describe the connection, or matrix, method for performing proofs. This method is due to Bibel [Bibel 81a, Bibel 82b, Bibel 81b, Bibel 80, Bibel 82a], and independently to Andrews [Andrews 81, Andrews 80]. These references should be consulted for a detailed description and treatment of the material presented here.

In the connection method the formula is viewed as a matrix, with the literals of the formula as the entries of the matrix. The position of each literal in the matrix is determined by the connective structure of the formula. Conjunction is represented by the rows of the matrix (two conjuncts appear in different rows of the matrix), and disjunction is represented by the columns of the matrix. The identities which allow the expression of other connectives in terms of these are used to determine the position of the remaining literals. It is important to note that the formula is not put into normal form in order to produce the matrix*, the positions of the literals in the matrix are merely determined by imagining that the formula was in normal form.

$$Man(socrates) \wedge \forall x.(\, Man(x) \rightarrow Mortal(x)) \rightarrow Mortal(socrates) \qquad (2)$$

The formula (2) is shown in matrix form in figure 1-38.

$Man(x)$ $\quad$ $\neg\, Man(socrates)$ $\quad$ $Mortal(Socrates)$

$\neg\, Mortal(x)$

Figure 1-38: Matrix form of (2)

As for resolution we negate the conjecture and attempt to show that no assignment of truth or falsehood to the literals of the formula is consistent. For a conjunction we must show that for each conjunct the formula is consistent, and the assignment must contain some contribution from each disjunct. Thus we have to show that each path through the matrix of the formula contains a contradiction, where a path is a selection of literals, one from each column of the matrix, and a contradiction is a pair of literals of opposite polarity whose atoms unify. Such a pair is called a *connection*. The two connections in the matrix 1-38 are shown in 1-39.

---

*Indeed, the matrix need not be created at all. The literals of the formula may be labelled with their position in the matrix, without actually creating the structure.

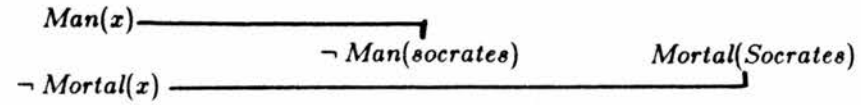

Figure 1-39: Matrix form of (2) showing connections

Since any path through the matrix must contain one of these connections, the original formula is a theorem.

The connection method has a number of advantages over resolution and natural deduction systems. In particular, the connection method requires no normal forming of the conjecture to be proved, and subgoals to be proved are not explicitly derived. The search for the paths can be performed by reference only to the original formula. This leads to a considerable saving of space in the implementation.

One of the perceived disadvantages of the connection method is that proofs are not produced in a manner that is obvious to humans. In [Bibel 81b], Bibel points out that there is a simple correspondence between the connection method and natural deduction proofs. Thus, completed proofs may be described in terms of natural deduction inference rules, despite the fact that they are not carried out according to those rules.

1.4.3. Summary

In this section I have described two logical systems for performing proofs mechanically. The resolution method, which is the most common proof method, requires extensive normal forming before the proof may be attempted. The connection method requires no such normal forming and is very efficient in the representation of formulae and search for proofs.

1.5. Why Natural deduction?

The question of why natural deduction should be used as the logic for a theorem prover is an important one, and will be addressed in this section. For an excellent discussion of non-resolution theorem proving techniques see [Bledsoe 77a].

The reasoning of the prover is easy for the human user to follow. This makes natural deduction the obvious choice for the implementation of interactive theorem proving systems. In such systems the prover may accept advice from the user regarding how the proof should progress. The logic of such systems must be completely comprehensible to users since they have to be able to interpret the output of the prover before giving advice. If the prover's reasoning bears a strong resemblance to the user's then this interaction will be facilitated.

The prover is able to do what the human can. If the theorem prover reasons in a very similar manner to the humans than a designer of the system can easily improve the system. Consider the situation where the human would take a particular step that the prover resolutely refuses to make. This may arise for two reasons: either the prover does not have the capability to make the step, i.e., the inference rule is missing, or the prover is using some other inference rule instead. In either case the comparison between human and machine reasoning facilitates the examination of the program's abilities, and suggests appropriate fixes.

Heuristics are very easily specified within a natural deduction framework. As an extension of the remarks in the previous paragraph, the heuristics that the human adopts in the search for a proof can also be more easily programmed into a natural deduction engine. This is because humans are likely to formulate these heuristics in terms of the inferences that they would make. Since these inferences bear a strong resemblance to those made by the prover, the translation of the heuristic into the vocabulary of the prover is facilitated.

Natural deduction also has some disadvantages as the logic for an automatic theorem prover. The worst of these is its inefficiency compared to the machine-oriented proof systems of resolution and the connection method. This inefficiency arises from the fact that natural deduction is *connective*-based rather than *connection*-based. A particular rule of natural deduction is appropriate for any formula with the required major connective. No reference is made to the desired goal, namely to introduce unifiable formulae into hypothesis and conclusion. In the resolution and connection methods, these unifiable pairs of formulae are sought first, and then the appropriate decomposition of the formula is found. This avoids the redundant decomposition of formulae which occurs in the sequent based systems. For a more detailed discussion of this question see [Wallen 86].

The connection method is more efficient than resolution since the connection method only considers the original formula whereas the resolution rule derives new clauses which become candidates for further resolution steps. Unguided resolution quickly becomes swamped with clauses and the resulting combinatorics quickly defeat a mechanical prover. Sequent based systems also require the explicit representation of subgoals, and are also prone to this problem.

In this section I have outlined three reasons for considering natural deduction as a good logic for building mechanical theorem provers. These hinge on the observation that for human users to successfully interact with a prover, the prover should reason in a way that is similar to that of the human. The interaction between man and machine may be at many levels: the expression of hints to the prover as to how to perform a particular proof, the

comparison of machine proofs and human proofs to identify different proof paths, or the expression of heuristics for carrying out certain types of inference.

Some drawbacks of sequent and natural deduction based systems have also been pointed out. In particular they, like resolution, are inefficient in that they require the explicit representation of subgoals. Unlike resolution, there is additionally a choice of inference rules within a sequent based system, and the overhead of deciding which of these rules to use does not occur in connection method or resolution systems.

1.6. Summary

In this chapter I have introduced the first-order predicate calculus, and the natural deduction proof systems $\mathcal{ND}$ and $\mathcal{GSC}$. These systems will form the logical basis for the automatic theorem proving systems which will be described in the following chapter.

I also described some techniques that have been devised in order to facilitate the automation of natural deduction theorem provers. In particular the techniques of skolemization and unification, which enable provers to avoid the restrictions on quantifier rules, and rewriting were described. These techniques are all in general use and are quite powerful. The dual notions of forward- and backward-inference were introduced. Most rules of $\mathcal{ND}$ are used by automated provers exclusively in one direction in order to control the possible choices that could be made. This has the effect of ensuring that when an inference rule is applied to a goal the resulting subgoals are simpler, in some sense, than the original.

Finally I presented two alternative inference rules for automatically proving theorems, resolution and the connection method. These are more efficient for automatic provers, but are thought less natural for a human to follow. Some reasons why a natural proof system is desirable were also presented in section 1.5.

In chapters 2 and 3 I will describe some theorem proving programs which implement proof systems from the natural deduction family.

Chapter 2
The UT Provers

2.1. Overview

In this chapter I will describe two programs developed at the University of Texas, which implement natural deduction theorem provers. I will indicate the features of the systems and the philosophy underlying their design. Where these provers use techniques which are not part of natural deduction I will describe the techniques and give reasons for their use. The aim of this chapter is to enable an assessment of the extent to which the programs are implementations of natural deduction as described in chapter 1.

In section 2.2 I describe the background to the programs and the philosophy behind their design. Sections 2.3 and 2.4 describe the two programs. Each of these sections is divided into two parts. The first part of each section describes the rules which have a direct correspondence to the rules of the logical systems described in chapter 1. In the second part of each of these sections I describe the extra features of the systems that have been added to provide control of the search for proofs.

In section 2.5 I compare the provers with the logical systems described in chapter 1.

2.2. The UT Provers

While the titles of [Bledsoe & Tyson 75a, Bledsoe 83] appear to refer to the same program, "the" UT prover is in fact (at least) two programs. In this section I describe these two natural deduction theorem provers. Details of certain aspects of the programs may be found in [Bledsoe & Tyson 75b, Bledsoe *et al* 79, Bledsoe 77b, Bledsoe & Tyson 78]. The prover described in [Bledsoe & Tyson 78] is almost identical to that described in [Bledsoe & Tyson 75a].

These provers are by no means the only natural deduction provers: [Brown 78, Nevins 74, Pastre 77, Reiter 76, Cvetkovic & Pevac 83] contain descriptions of some others.

Definition 24: I shall refer to the prover described in [Bledsoe & Tyson 75a] as PROVER.

Definition 25: I shall call the prover described in [Bledsoe 83] as IMPLY.

The research project which produced PROVER and IMPLY has been in progress for some time, and is still continuing. Many interesting and significant techniques have been discovered during the course of this work. The constraints of automation and the devising of new techniques for reasoning mean that the programs have rules which are derived from those of $\mathcal{ND}$, rather than the rules of $\mathcal{ND}$ itself. My intent in this chapter is to provide a description of the provers and an assessment of the relationship between the provers and the natural deduction system of chapter 1.

The guiding principle of the UT provers is that the proof should proceed in a way that is natural to the human mathematician. In order to achieve this, the provers implement a logic which is alleged to perform proofs in a more natural way than a system based on resolution, [Robinson 65], or the connection method [Bibel 82a] which were described in chapter 1 (section 1.4.3). PROVER is an interactive system, meaning that the human user may interact with the prover to suggest possible courses of action or to perform inferences on the provers behalf. Clearly for such a system to be workable, the proof attempt carried out by PROVER must be intelligible to the user, and this is the motivation for the choice of $\mathcal{ND}$ as the underlying logic.

Both programs have a number of rules which may be used when given a goal to prove. Which particular rule is used depends on the form of the goal. The effect of each rule is either to complete a branch of the proof or to create subgoals which are in some way simpler than the original goal. The proofs of the subgoals combined appropriately with the inference rule that has been used constitute a proof of the main goal. These rules are examined in turn by the prover until one is found which applies to the given goal. The knowledge that the prover has of how to prove theorems is therefore in two forms: the inference rules that the prover has, and the order in which the rules are examined.

The rules may be divided into 3 classes:

1. *Logical.* These rules correspond in some way to the rules of $\mathcal{ND}$, for example, $\rightarrow\!-E$.

2. *Database.* These rules allow the prover to deal with defined terms, and previously proved facts. An example of such a rule is expanding the definition of a predicate by use of its defining formula.

3. *Special Purpose.* These rules allow more complicated inferences to take place. For example, one of these rules can detect contradictions in assertions about the values of numerical variables.

The logical rules correspond most directly to the rules of $\mathcal{ND}$. However the prover is

restricted to either forward or backward reasoning for each rule. Some augmentation of the rules has also been allowed, resulting in a more powerful prover than would otherwise have been possible. This augmentation can be justified by presenting a proof of each of the prover rules, in terms of the basic *ND* rules. In this sense the prover rules are *derived* from those of *ND*.

The database rules are those which access the databases of lemmas and definitions described in section 1.3.4. The prover has many rules which correspond to an application of the rewriting inference rule. This is because the prover distinguishes different classes of rewrite rules - in particular predicate definitions, lemmas, and function definitions are all used as rewrite rules by different rules of the prover. The classification of rewrite rules in this way enables the prover to limit the use of some rewrite rules while allowing others to be used more frequently.

In [Stickel 85] the notion of *theory* is defined as *any satisfiable set of formulae that we want to incorporate into the inference process*. The theory that PROVER is working in is formed by the database of rewrite rules. If this database were empty then the theorem prover would only be able to prove theorems in first-order logic. It is the contents of the database, and the intended interpretation of the concepts that are defined which allow us to interpret the proofs that the provers carry out as meaningful.

The special purpose inference rules implement methods for dealing with commonly occurring features of the theories that the provers were designed to perform proofs in. While the database rules simply replace one formula by another, the special purpose rules allow the provers to carry out more complex inferences, which would involve many applications of the definitions and axioms of the theory.

Both PROVER and IMPLY are implemented in the programming language LISP. The inference rules are LISP functions which take the sequent to be proved as an argument, and return the value of the substitution that is necessary to make the proof go through. The value **T** is returned if the proof is propositionally true: no substitution of values for variables being necessary. **NIL** is returned if a proof cannot be found. The rules are Condition/Action/Result triples. These say that if the conditions are true, then perform the action (typically evaluate some function), and then return the specified result. I have given each rule a name, and when referring to a rule this name will appear in **bold** type. Some of these rules were named by Bledsoe in [Bledsoe & Tyson 75a, Bledsoe 83]; I have provided names here for the remainder.

The next two sections of this chapter deal separately with the rules of each of the provers.

Each section is itself divided into two, one dealing with the logical rules and the other with the database and special-purpose rules.

2.3. PROVER

In this section I shall describe the rules of PROVER, the program described in [Bledsoe & Tyson 75a]. This section is divided into two parts: in 2.3.1 I will describe the rules of logic that PROVER is able to apply, moving on to the database and special-purpose rules in 2.3.2. PROVER is divided into two subroutines, called *Imply** and *Hoa*. I will briefly describe the rules of each subroutine.

Definition 26: A call to the routine Hoa, with hypothesis H and conclusion C will be written $H \vdash_h^? C$.

Definition 27: A call to the routine Imply, with hypothesis H and conclusion C will be written $H \vdash_i^? C$.

Definition 28: Most of the rules of both PROVER and IMPLY have a condition on the form of one of the formulae of the conjecture. I will represent such conditions by $\phi \equiv \epsilon$ indicating that the formula ϕ is of the same form as ϵ.

Definition 29: When two subgoals that are given to the prover succeed, each has as its result the substitution necessary to make the proof go through. The notation $\theta \circ \sigma$, means the composition of the two substitutions θ and σ. The problem of composing two substitutions is discussed in section 2.4.1 on page 60.

Definition 30: Calls to other routines used by the prover will be represented by the notation $R(a_1, \ldots, a_n)$, where R is the name of the routine, and the a_i are the arguments. These supporting routines are described in the text.

Definition 31: The notation $V := E$, means that the program variable V is set to the value of the expression E.

Hoa stands for Hypothesis Or And. Hoa's function is to search through the conjuncts of the hypothesis formula until it finds one which can be used by one of the rules of the system. The rule is applied to this conjunct, and the recursive call - if any - is made with the original conjunction. Thus Hoa searches through the hypothesis formula for a useful conjunct, and then returns a new goal to Imply. Imply simply attempts the proof of the conclusion from the current hypothesis. All but one of Imply's rules consider the form of the conclusion and suggest new subgoals on the basis of this information.

The search for a proof begins by skolemizing the formula to be proved, and then passing the skolemized goal to Imply. Each rule has three parts: condition, action and result. The

---

*The potential for confusion is great here. I will distinguish the subroutine of PROVER from the later version of the program, by always writing the latter in capital letters. Thus, Imply is a subroutine of PROVER, while IMPLY is a complete program.

prover considers each rule in turn, examining the conditions and actions of the rule. Actions are simply carried out, they generally construct new formulae for the prover to consider, and carry out subproofs. Conditions are tests that have to be true before the prover may use the rule. If any condition is not true of the current conjecture, then the rule is not used and the next in the sequence is examined. When all of the actions have been completed, and all conditions have been found to be true, the value specified in the result part of the rule is returned.

The rules of Imply are given in figures 2-1 and 2-2. The conjecture to be proved is assumed to have the hypothesis H, and conclusion C.

The rules of the logic of all of the provers discussed within this thesis are presented in tabular form. These tables present the inference rules in an algorithmic manner, and so the order, as well as the layout of the rules is significant. Consider, for example, rule 4 (**And Split**) from figure 2-1. This rule should be read as:

1. (First line) If the conclusion of the goal is a conjunction of formulae A and B, then set θ to be the value returned by the proof of A from the current hypotheses, H, using the rules of Imply.

2. (Second line) If θ has the value **NIL** then return the value **NIL** for this goal.

3. (Third Line) If, on the other hand, θ has a non-**NIL** value, then set λ to be the value returned by the proof of B from the current hypotheses, H, using the rules of Imply.

4. (Fourth Line) If the value of λ is **NIL** then return the value **NIL** for this goal.

5. (Fifth line) Otherwise, λ has a non-**NIL** value, so return the composition of the values θ and λ as the result of this proof.

H $\vdash_i^?$ C

RuleName	Condition	Action	Result
1. (Truth)	$H \equiv \bot$		**T**
	$C \equiv \top$		**T**
2. (Typelist)	Typelist (see page 48)		
3. (Cases)	$H \equiv (A \vee B)$	$NewC := (A \rightarrow C) \wedge (B \rightarrow C)$	
			$\top \vdash_i^? NewC$
4. (And split)	$C \equiv (A \wedge B)$	$\theta := H \vdash_i^? A$	
	$\theta =$ **NIL**		**NIL**
	$\theta \neq$ **NIL**	$\lambda := H \vdash_i^? B$	
	$\lambda =$ **NIL**		**NIL**
	$\lambda \neq$ **NIL**		$\lambda \circ \theta$
5. (Reduce)		$NewH := Reduce(H)$	
		$NewC := Reduce(C)$	
	$NewH \neq H$ or $NewC \neq C$		$NewH \vdash_i^? NewC$
6. (Or fork)	$C \equiv (A \vee B)$		$H \vdash_h^? C$
7. (Promote)	$C \equiv (A \rightarrow B)$	$InterimH := Forward\text{-}Chain(H,A)$	
		$NewC := Andout(B)$	
		$NewH := Orout(InterimH \wedge A)$	
			$NewH \vdash_i^? NewC$
8. (Prove Equiv)	$C \equiv (A \leftrightarrow B)$	$NewC := (A \rightarrow B) \wedge (B \rightarrow A)$	
			$H \vdash_i^? NewC$
9. (= C)	$C \equiv (A = B)$	$\theta := Unify(A,B)$	
	$\theta \neq$ **NIL**		θ
10. (Flip C)	$C \equiv \neg A$		$H \wedge A \vdash_i^? \bot$
11. (Inequality)	$C \equiv (A \leq B)$	$Type := Set\text{-}Type(\neg (A \leq B))$	
	Type has contradiction		**T**
	otherwise,		$H \vdash_h^? C$

Figure 2-1: The rules of Imply - a subroutine of PROVER - Part 1

H $\vdash_i^?$ C

RuleName	Condition	Action	Result
12. (Call Hoa)		$\theta := H \vdash_h^? C$	
	$\theta \neq$ **NIL**		θ
	$\theta =$ **NIL**	Set Flag, $\lambda := H \vdash_h^? C$	
	$\lambda \neq$ **NIL**		λ
13. (Define C)		$NewC := Define(C)$	
	$NewC \neq C$		$H \vdash_i^? NewC$
14. (Eq InEq)	$C \equiv (a = b)$	$NewC := a \leq b \wedge b \leq a$	
			$H \vdash_i^? NewC$
	$C \equiv (a \neq b)$	$NewC := a < b \vee b < a$	
			$H \vdash_i^? NewC$
15. (Imply Fail)			**NIL**

Figure 2-2: The rules of Imply - a subroutine of PROVER - Part 2

Hoa is called from Imply by rules 6, 11 and 12. The tables 2-3 and 2-4 give the rules of this routine. The call to Hoa is made with the hypothesis called B and the conclusion C. The call to Imply which resulted in Hoa being called had hypothesis, H.

$\mathbf{B} \mid\!\!-_h^? \mathbf{C}$

RuleName	Condition	Action	Result
1. (Time)	Time Limit Exceeded		**NIL**
2a. (Match)		$\theta := Unify(B,C)$	
	$\theta \neq$ **NIL**		θ
2b. (Peek)	Flag is set	$NewH := Peek(H,C)$	
	$NewH \neq H$		$NewH \vdash_i^? C$
3. (Pairs)	Flag is set $H \equiv P(\tau_1, \ldots, \tau_n)$, and, $C \equiv P(\sigma_1, \ldots, \sigma_n)$		
		$NewC := Pairs(P)$	
	$NewC \neq C$		$\top \vdash_i^? NewC$
4. (Or Split)	$C \equiv A \vee D$	$NewC := AndOut(C)$	
	$NewC \neq C$		$H \vdash_i^? NewC$
	$NewC = C$	$\theta := B \wedge \neg D \vdash_h^? A$	
	$\theta \neq$ **NIL**		θ
	$\theta =$ **NIL**		$B \wedge \neg A \vdash_h^? D$
5. (Call Imply)	$C \equiv A \rightarrow D$, or, $C \equiv A \wedge D$		$B \vdash_i^? C$
6. (And Fork)	$B \equiv A \wedge D$	$\theta := A \vdash_h^? C$	
	$\theta \neq$ **NIL**		θ
	$\theta =$ **NIL**		$D \vdash_h^? C$
7. (Back Chain)	$B \equiv (A \rightarrow D)$	$\theta := Ands(D,C)$	
	$\theta \neq$ **NIL**	$\lambda := H \vdash_i^? A\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
7E. (Back Chain =)		$B \equiv (A \rightarrow x = y)$	
		$\theta := x = y \vdash_h^? C$	
	$\theta =$ **NIL**		**NIL**
	$\theta \neq$ **NIL**	$\lambda := H \vdash_i^? A\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
8. (Equiv H)	$B \equiv (A \leftrightarrow D)$	$NewH := (A \rightarrow D) \wedge (D \rightarrow A)$	
			$NewH \vdash_h^? C$

Figure 2-3: The Rules of Hoa - Part 1

B $\vdash_h^?$ **C**

RuleName	Condition	Action	Result
9. (Sub =)	$B \equiv x = y$	$\nu := Minus\text{-}On(x,y)$	
	$\nu = 0$		**NIL**
	ν is a number		**T**
	ν is not a number	$x' := Choose(x,y)$	
		$y' := Other(x,y)$	$H^{y'}_{x'} \vdash_i^? C^{y'}_{x'}$
10. (Hoa Cases)	$B \equiv (A \vee D)$		$B \vdash_i^? C$
11. (Flip H)	$B \equiv \neg A$	$NewC := A \vee C$	$H \vdash_i^? NewC$
12. (Hoa Fail)			**NIL**

Figure 2-4: The Rules of Hoa - Part 2

2.3.1. The Logical Rules of PROVER

In this section I discuss the logical rules of PROVER. Each rule suggests 0, 1 or 2 new subgoals of the original goal which are to be proved. If there are no new subgoals, then the proof of the goal is complete. If on the other hand there are subgoals to be proved then completing the proofs of these subgoals will complete the proof of the main goal. When there are two suggested subgoals one of two situations arise, either both subgoals must be proved, or a proof of one of the subgoals will be sufficient.

PROVER completes a proof by applying one of the rules: **Match** (figure 2-3, Hoa rule 2), **Truth** (figure 2-1, Imply rule 1), **Sub** = (figure 2-4, Hoa rule 9), = **C** (figure 2-1, Imply rule 9) or **Inequality** (figure 2-1, Imply rule 11).

Match is used if the conclusion and hypothesis of the goal will unify. The result that is returned is the most general unifier of the two formulae. That is, the most general substitution which when applied to both formulae makes them identical (see section 1.3.3 of chapter 1). The effect of this rule is to join together two partial proofs (either or both of which might be empty). Suppose that some inference rules have been applied to the original hypotheses to get *NewH*, and that some inference rules have been applied to the original conclusion to get *NewC*, and that *NewH* and *NewC* unify. Then we have the proofs at the top of figure 2-5 on page 40. The inferences that allow us to change the original conclusion to *NewC* are necessarily backward inferences, so the proof is from *NewC* to the original conclusion. Since *NewC* and *NewH* unify, we can compose the two partial proofs by identifying the formulae in the proofs, and applying the unifying substitution to both proofs throughout. The result is shown at the bottom of figure 2-5 on page 40.

Original Hypotheses
$\prod$
NewH

NewC
$\sum$
Original Conclusion

Original Hypotheses
$\prod$
NewH (= NewC)
$\sum$
Original Conclusion

Figure 2-5: Applying **Match** to two Partial Proofs

The **Truth** rule implements the absurdity rule of *ND*, and its dual which does not appear as a rule of *ND*. Since PROVER has the ability to rewrite some formulae to truth values using rewrite rules, the system must be able to deal with the truth formulae. *ND* contains the absurdity rule (1-15 on page 11) which allows any conclusion to be deduced from the formula $\bot$. If PROVER has managed to reduce the hypothesis of the goal to be the atom $\bot$, then the goal must be true by this rule. This situation is detected by case 1 of the **Truth** rule. Similarly, if the conclusion has been reduced to the atom $\top$ then the conclusion is indeed true, and this fact does not depend on the hypotheses. This situation is detected by case 2 of **Truth**. Whenever **Truth** is used, the value that is returned by the rule is **T**, indicating that no binding of variables is necessary to complete the proof.

Sub $=$ (figure 2-4, Hoa rule 9), and $=$ **C** (figure 2-1, Imply rule 9) can also terminate a proof. These rules implement the equality axioms which are troublesome for automatic provers, since they are rather powerful. The rule $=$ **C** gives PROVER enough power to detect when the conjecture is proved. This rule represents axiom 1 of the equality axioms given below:

1. $\dfrac{}{\xi = \xi}$

2. $\dfrac{\xi = \psi}{\psi = \xi}$

3. $\dfrac{\xi = \psi \qquad \Pi(\xi)}{\Pi(\psi)}$

Axiom 1 says that if the same object appears on either side of the equality then the literal is true. PROVER implements this axiom by allowing the prover to attempt to unify the two objects if the conclusion of a goal is an equality. If they do unify, then the the conjecture is true. If they do not unify, then it doesn't mean that they are different objects; it may mean that there are two different terms representing the same object (for example, 2+2 and 3+1

are two different terms representing the number 4). This situation may be detected by **Sub =**. This rule allows the prover to simplify complex equality expressions, by performing partial evaluation, and making substitutions of equal objects.

Sub = allows the simplification of equality hypotheses by symbolically subtracting one of the sides from the other. This subtraction is performed by a routine called *Minus-On*. Symbolic subtraction allows the expressions that are to be evaluated to contain non-numeric constants or variables, and so subtracting two quantities using Minus-On may have one of three results:

- If the result is 0, then nothing can be deduced, except that the hypothesis is true. Hoa returns **NIL** in this case, since none of the subsequent rules can be used when the hypothesis is a literal.
- If the result is any other number, then we may deduce that the hypothesis is false, since for the difference to be non-zero, the quantities must be different. Since the hypothesis is false, the conjecture is proved - this is a special case of the **Truth** rule.
- If the subtraction evaluates to a new expression that is not a number then the prover is able to use axiom 3.

Axiom 3 says that if two objects are equal, and one of them has a certain property, then so does the other. Thus, it is possible to simplify the conjecture by replacing one of the objects by the other throughout. Once this is done, some unifications that were not possible may become so and so the conjecture may be provable. It is permissible to replace either of the expressions by the other (by equality axiom 2), but PROVER has a routine which decides which to replace on the basis of simplicity. The functions *Choose* and *Other* take the two equal terms, and return, respectively, the simplest of the terms and the other term, and then PROVER makes the replacement of the first by the second. The replacement described above takes place in the original hypothesis and conclusion. That is, the complete conjunction of which the equality is one conjunct is used as the hypothesis for the recursive call to Imply.

Inequality can complete a proof. A discussion of this rule is deferred until the following section (2.3.2).

The remaining rules each require that one or more subgoals be proved to complete the proof of the main goal. The rules **Or Split** (figure 2-3, Hoa rule 4) and **And Fork** (figure 2-3, Hoa rule 6) suggest two different subgoals but require that only one of these be proved to complete the proof. This situation is called an *or-choice*, since PROVER has to prove either one subgoal or the other. *And-choices*, on the other hand, require PROVER to complete the proofs of both suggested subgoals before the proof of the main goal is complete. **Cases** (figure 2-1, Imply rule 3), **And Split** (figure 2-1, Imply rule 4), **Back Chain** (figure 2-3, Hoa

rule 7) and **Back Chain** $=$ (figure 2-3, Hoa rule 7E) all force the prover to make an and-choice.

Cases is applied on the condition that the hypothesis is a disjunction, $\alpha \vee \beta$. A recursive call to Imply is made with the hypothesis $\top$ and the conclusion: $(\alpha \rightarrow \mathit{Conclusion}) \wedge (\beta \rightarrow \mathit{Conclusion})$. In the notation of chapter 1 the rule is as in figure 2-6, where Π is supplied by the recursive call to Imply.

$$\dfrac{\alpha \vee \beta \qquad \dfrac{\overset{\{\}}{\Pi}}{(\alpha \rightarrow \chi) \wedge (\beta \rightarrow \chi)}}{\chi}$$

Figure 2-6: Cases

It is clear from the description of the prover that this subgoal will cause the **And-Split** rule to be applied when Imply is called to provide Π. This is because the conclusion has been forced to be a conjunction by the application of the **Cases** rule and the hypothesis is the atom $\top$. Thus, whenever **Cases** is invoked an application of **And-Split** follows immediately. The combined application of **Cases** and **And-Split** is shown in figure 2-7.

$$\dfrac{\alpha \vee \beta \qquad \dfrac{\overset{\{\}}{\Pi_1}}{\alpha \rightarrow \chi} \qquad \dfrac{\overset{\{\}}{\Pi_2}}{\beta \rightarrow \chi}}{\chi}$$

Figure 2-7: The combination of **Cases** and **And-Split**

At this stage the choice of the next rule for producing the Π_is is also forced. Again the hypothesis is empty, and the conclusions of both goals are implications. The only rule that applies to conclusions which are implications is **Promote** (figure 2-1, Imply rule 7) this causes the antecedent of each conclusion to be added to the appropriate hypothesis set. The choice of the next rule to be applied is not forced after **Promote** has been applied, since the form of the new hypothesis and conclusion will determine the rule to be used.

The total effect of the application of **Cases**, **And-Split** and **Promote** is shown schematically in figure 2-8. The similarity between this figure and the inference rule of $\vee\!-\!E$ in figure 1-4 of the previous chapter is obvious. Clearly, since the selection of the inference rules is forced from the use of **Cases** it would be much more efficient to perform this proof directly, and introduce the final subgoals directly. This approach is taken in IMPLY.

$$\dfrac{\alpha \vee \beta \qquad \dfrac{\dfrac{\dfrac{\alpha}{\vdots}}{\chi}}{\alpha \rightarrow \chi} \qquad \dfrac{\dfrac{\dfrac{\beta}{\vdots}}{\chi}}{\beta \rightarrow \chi}}{\alpha \rightarrow \chi \wedge \beta \rightarrow \chi}}{\chi}$$

Figure 2-8: Combined applications of **Cases**, **And-Split** and **Promote**

The rules **Or Split** and **And Fork** both cause two subproofs to be suggested. These subgoals represent an or-choice for PROVER since the success of either of these goals is sufficient to prove the main goal. **Or Split** applies if the conclusion is a disjunction. First the rule specifies that the conclusion should be rewritten by Andout (see the discussion of **Promote**). If this is possible then a call is made to Imply with the original hypothesis and the new conclusion. Otherwise an attempt is made to prove the left disjunct of the conclusion, from the conjunction of the hypothesis with the negation of the right disjunct, if this fails then an attempt is made to prove the right disjunct of the conclusion from the conjunction of the hypothesis with the negation of the left disjunct. Both of these attempts are made by invoking Hoa.

$$\dfrac{\dfrac{\eta \wedge \neg \beta}{\vdots}}{\dfrac{\alpha}{\alpha \vee \beta}} \qquad OR \qquad \dfrac{\dfrac{\eta \wedge \neg \alpha}{\vdots}}{\dfrac{\beta}{\alpha \vee \beta}}$$

Figure 2-9: Or-Split

And Fork is invoked if the hypothesis is a conjunction. This rule is central to Hoa since it is **And Fork** which allows Hoa to consider each of the conjuncts of a conjunction separately. Two subgoals are suggested by this rule, the first just uses the left conjunct of the hypothesis, and if this fails then the proof of the same conclusion is attempted from the right conjunct.

$$\dfrac{\dfrac{\alpha \wedge \delta}{\alpha}}{\dfrac{\vdots}{\chi}} \qquad OR \qquad \dfrac{\dfrac{\alpha \wedge \delta}{\delta}}{\dfrac{\vdots}{\chi}}$$

Figure 2-10: And-fork

Or Fork (figure 2-1, Imply rule 6) is called if the conclusion is a disjunction, the result is a call to the routine Hoa.

The application of **And Split** (figure 2-1, Imply rule 4) causes a recursive call to Imply to be made. The call attempts to show that the hypothesis implies the left conjunct of the conclusion. If this proof is successfully constructed then a call to show that the hypothesis

implies the right conjunct of the conclusion is made. The result of the call is the appropriate composition of the two substitutions. This is exactly the $\wedge - I$ rule of $\mathcal{ND}$ only this is being applied backward to eliminate the conjunction from the conclusion of the goal. Notice that the application of this rule represents an and-choice for PROVER.

Although two subgoals suggested by a use of **And Split** are solved independently, they may have variables in common. These shared variables make the solution to the goals dependent, since each variable may be bound to only one term. It may be the case that solving one of the goals binds a shared variable to some term which prevents the proof of the other subgoal being completed. This is called *trapping*, and is discussed in relation to completeness in section 2.5. For example, consider the goal (G),

$$P(a) \wedge P(b) \wedge Q(b) \vdash^{?} P(x) \wedge Q(x) \qquad (G)$$

After splitting, two subgoals arise with a common variable x: namely to prove $P(x)$ and $Q(x)$. There are two choices for binding x in the $P(x)$ goal, either a/x or b/x, but choosing a/x makes it impossible to prove $Q(a)$.

Imply has no means of dealing with this problem, but in IMPLY a method for analysing the failure is used. This is described on page 60.

Promote (figure 2-1, Imply rule 7) applies if the conclusion is an implication. Here the antecedent is conjoined with the hypothesis and this is the new hypothesis for a recursive call to Imply. The new conclusion is the consequent of the old conclusion. This is exactly the $\rightarrow - I$ rule of natural deduction, except that the rule is used for backward inference. The subproof that is represented by Π in figure 1-6 (page 9) is to be produced by the recursive call to Imply.

Promote is one of the rules of $\mathcal{ND}$ that has been augmented considerably in the implementation of PROVER. There are two features of the implementation that must be noted. The first is that PROVER rewrites the hypothesis and conclusion if possible when **Promote** is applied. The rewriting is carried out by the routines *Andout* and *Orout*, the idea of these routines is to attempt to bring conjunctions (respectively disjunctions) toward the "top" of the formulae.

$$\alpha \vee (\beta \wedge \chi) \Rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \chi)$$

$$(\alpha \wedge \beta) \vee \chi \Rightarrow (\alpha \vee \chi) \wedge (\beta \vee \chi)$$

$$\neg(\alpha \vee \beta) \Rightarrow \neg\alpha \wedge \neg\beta$$

Figure 2-11: The Rules of Andout

The rewrite rules that are used by Andout are given in figure 2-11, and those used by Orout in figure 2-12. These rewrite rules are applied repeatedly until the formula cannot be rewritten further. *Andout* is applied to the conclusion and *Orout* to the hypothesis of the new goal.

$$\alpha \wedge (\beta \vee \chi) \Rightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \chi)$$

$$(\alpha \vee \beta) \wedge \chi \Rightarrow (\alpha \wedge \chi) \vee (\beta \wedge \chi)$$

$$\neg(\alpha \wedge \beta) \Rightarrow \neg\alpha \vee \neg\beta$$

Figure 2-12: The Rules of Orout

The implementation of PROVER records more than just that these rules may be used to rewrite formulae, but also when it is *useful* to rewrite a formula to its equivalent form. Thus these rewrite rules are only applied to the formulae that PROVER is manipulating at the entry to **Promote**.

In addition to rewriting both the conclusion and hypothesis, PROVER has the option of forward chaining when a new conjunct is added to the hypothesis. A call to the routine *Forward-chain* is made when the **Promote** rule is called. This routine references a flag which is set by the user. If the flag allows, forward chaining takes place. The flag may specify extra conditions on the new and existing hypotheses which allow forward chaining to take place only when these conditions are met, or prevent forward chaining completely. If forward chaining is to take place, the new hypothesis is matched with the antecedent of any implications which are also hypotheses. If a match can be made, then the consequent of the implication may also be added as a new hypothesis. Forward chaining corresponds to the use of the rule of $\rightarrow\!-E$, which is also implemented in PROVER as the **Back Chain** rule. Forward chaining in this way is a very powerful technique, and the flag that is used in PROVER enables its use to be controlled. The need for control is evidenced by the fact that the inference that is carried out in a forward direction. Under certain circumstances the prover would be able to deduce many hypotheses which may not be relevant to the goal to be proved.

The technique of forward chaining is further extended in *peek-forward-chaining*. Here when a conjunct of the hypothesis is not an implication the system looks at the definition of the predicates in the hypothesis. If peeking (see section 2.3.2) shows that the definition of some predicate contains the predicate of the new hypothesis, the definition is temporarily expanded and then forward-chaining is attempted with this hypothesis.

Rules **Back Chain** (figure 2-3, Hoa rule 7) and **Back Chain** $=$ (figure 2-3, Hoa rule 7E) apply when the hypothesis is an implication. First an attempt is made to prove that the

consequent of the hypothesis implies the conclusion, if this can't be done then the rule fails. This subproof is attempted by a simple prover, called *Ands*. Ands will detect if some conjunct of the hypothesis that is passed to it will match the conclusion, returning the substitution necessary to make the match. That is, Ands can detect the truth of goals of the form:

$$H_1 \wedge \ldots \wedge H_i\theta \wedge \ldots \wedge H_n \vdash C\theta$$

Where θ is a unifier for H_i and C.

If the Ands subproof succeeds, then the antecedent of the implication becomes the conclusion of a new subgoal to be proved by Imply. This goal is to be proved using the original hypothesis. If the subproof of the conclusion fails, then there are two alternatives, first is to apply **Back Chain** $=$. This is only possible if the consequent of the implication is an equality literal. Remembering that the proof of this literal has not been proved by Ands, the proof attempt is made by Hoa. Hoa includes a routine which can manipulate equalities: **Sub** $=$, and it is this rule that is expected to provide the proof. Otherwise the back chaining attempt is unsuccessful.

$$\dfrac{\dfrac{\overset{\eta}{\Pi}}{\alpha} \qquad \alpha \rightarrow \beta}{\dfrac{\overset{\beta}{\Sigma}}{\chi}}$$

Figure 2-13: Back-Chain

Notice the similarity between forward- and backward-chaining. Both of these rules apply if the hypothesis is an implication, and implement the *ND* rule of $\rightarrow\!-E$. The difference is that forward-chaining is making inferences in a forward direction: from the hypotheses to new hypotheses, while the use of back-chaining depends of the conclusion and consequent of the implication being related. This is a backward inference, since a new conclusion is suggested by the prover once the proof of the existing conclusion has been found.

Hoa Cases (figure 2-4, Hoa rule 10) is invoked if the hypothesis is a disjunction and the result is a call to Imply with the hypothesis and conclusion unchanged. Thus the conjuncts of the hypothesis that Hoa might have ignored by use of the **And Fork** are not available in the proof of this subgoal.

Flip C (figure 2-1, Imply rule 10) corresponds exactly to the rule of $\neg\!-I$, or *reductio ad absurdum*. This says that, to show that a formula is not true, assume that it is, and then deduce a contradiction. The *ND* rule is given in figure 1-10 (page 10).

If the hypothesis of the current goal is a negation, then this conjunct can be deleted from the hypothesis and a new conclusion proved from the remaining hypotheses. This new conclusion is formed by disjoining the unnegated hypothesis with the old conclusion. Schematically, the rule is as in figure 2-14. This rule is **Flip H** (figure 2-4, Hoa rule 11), and does not have an equivalent in the *ND* system.

$$\dfrac{\dfrac{\eta \wedge \neg \alpha}{\Pi}}{\dfrac{\alpha \vee \chi}{\chi}}$$

Figure 2-14: Flip - H

Flip-H may be justified by the *ND* proof of figure 2-15.

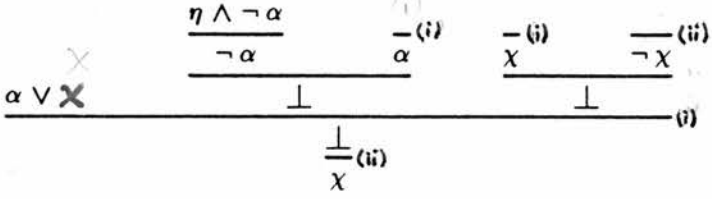

Figure 2-15: The justification of **Flip-H**

The two rules which deal with equivalences, **Prove Equiv** (figure 2-1, Imply rule 8) and **Equiv H** (figure 2-3, Hoa rule 8), correspond exactly to the rules of *ND*. However since the formula that is to be proved is skolemized before the proof is attempted, and the skolemization step requires that all biconditional connectives are eliminated, these rules can never be used. The presence of these rules in PROVER, since they do not effect the behaviour of the prover in any way, is something of a mystery.

2.3.2. The Non-Logical Rules of PROVER

As remarked earlier, PROVER includes a number of inference rules which cannot be found in a natural deduction system. I will divide these into two types, database and special-purpose. The first type enable the prover to handle defined terms, and to use previously proved facts. The special-purpose rules allow the prover to use very efficient procedures to make inferences about certain formulae. For example, the **Typelist** rule of Imply, allows the prover to detect a contradiction which would involve a large amount of inference if it were to be detected by more "conventional" means.

Failure Rules

There are a number of ways in which the proof can be artificially terminated with failure. The rules **Imply Fail** (figure 2-2, Imply rule 15) and **Hoa Fail** (figure 2-4, Hoa rule 12) are the most obvious, these are the last rules of each of the subroutines, and as such tells the prover what to do if all of the other rules have not produced a proof. **Time** (figure 2-3, Hoa

rule 1) is another example of a failure rule. The programmer is allowed to specify a time limit on the search for the proof, if this is exceeded then PROVER fails the proof, returning to the user for advice. Obviously none of these rule have analogues in *ND*, since they are concerned solely with the control and implementation of the search for a proof.

The Typelist

PROVER maintains a representation of the "type" of each of the variables in the conjecture. A type is usually a record of what the object is; for example a set or a number, however in PROVER this notion is taken further. If the object is a number, then the type of the object records the bounds within which the number may lie. For example; if the hypotheses $x \leq 4$ and $2 \leq x$ were known then the type of x would be the interval [2,4]. If later the hypothesis $x < 1$ were deduced, then it would be impossible to represent the type of x, and so PROVER would report a contradiction. This is exactly the same as reducing the hypothesis to $\bot$. The **Typelist** rule checks the types of all of the objects for consistency, and if any object is found with an inconsistent type then the proof is completed. It is important to notice that the inference that is performed by the **Typelist** rule, could be carried out by a prover in other ways. The **Typelist** rule is powerful for the following reasons:

- **The updating and consistency checking of the types of the objects can be performed very efficiently,**
- **By recording the types of the objects in the typelist, many explicit hypotheses can be eliminated from the conjecture. This results in the number of choices that are open to PROVER being decreased, and,**
- **The ordering axioms of the real numbers are not represented explicitly in the prover.**

The PROVER rules **Typelist** (figure 2-1, Imply rule 2), **Inequality** (figure 2-1, Imply rule 11), and **Eq Ineq** (figure 2-2, Imply rule 14) relate to this elegant technique. **Typelist** allows the prover to perform case analysis on the types of objects. For example, suppose that the hypotheses of the theorem include the formula $(j \leq 2) \vee (4 \leq j)$. In this case PROVER constructs two typelists one which contains the type recording $(j \leq 2)$ and the other recording $(4 \leq j)$. The theorem must be proved for both typelists, exactly as the theorem must be proved for both disjuncts of a hypothesis. This becomes clear once it is recognized that the typelist is merely an alternative representation of formulae. This type of case analysis is performed by the **Typelist** rule. This rule is described in detail in [Bledsoe & Tyson 75b].

Inequality is used when the conclusion is an inequality. The inequality is negated and the representation of this type is added to the existing typelist, by using the routine *Set-Type*. If a contradiction is deduced then the theorem is proved and the result **T** is returned. However if no contradiction can be deduced Hoa is called with the updated typelist. This is justified

since if there is a contradiction in assuming that the conclusion is false while the hypotheses are true then the theorem must be true.

Eq Ineq manipulates equality literals which appear in the conclusion into a form where the information that they contain can be used by the typelist. An equality literal $a = b$ is replaced by the formula $(a \leq b) \wedge (b \leq a)$. A literal of the form $a \neq b$ is replaced by $(a < b) \vee (b < a)$. In both of these cases the typelist routines can use the new formula to update the typelist, while the equality form could not be so used. This rule is a special case of **Define C**, since equality of numbers could be considered to be defined in terms of these inequalities. The use of a special rule to do these particular expansions means that when these symbols occur in the hypothesis, they may not be rewritten using these definitions.

Rewriting Rules

There are a number of PROVER rules which rewrite the goal to be proved using rewrite rules from the databases. The separation of the rewrite rules into different databases allows the prover to distinguish between the different types of rules, and to use rules which are appropriate given the context. For example, expanding a definition is performed by using a rewrite rule, but this inference is one that PROVER is reluctant to make, since it can lead to explosive search. The PROVER rule which applies rewrite rules from the database of definitions thus applies late in the rule ordering. The simplification of formulae using logical truths however, is a step that PROVER is more willing to take, since it does not usually have disastrous consequences, so the rule which applies rewrite rules of this class appears earlier.

Reduce (figure 2-1, Imply rule 5) allows the implementation of two types of inference. The first type is the simple rewriting of formulae to their equivalents by the identities of logic. Such rewritings are very important, since they allow the prover to simplify its hypotheses and conclusion. The set of rules available to PROVER are given in 2-17 and 2-16. Notice that, for example, the use of the last rule in figure 2-16 could save the prover much work, since if a duplicated conjunction occurred in the conclusion of a sequent and it was not rewritten then the prover would have to prove this conjunct twice. The proof would be exactly the same, and the work involved would be completely redundant. The first rule of 2-16 is the law of excluded middle, which we saw as an axiom of $\mathcal{ND}$ (1-16).

$\alpha \vee \neg \alpha \Rightarrow \top$

$\alpha \wedge \neg \alpha \Rightarrow \bot$

$\alpha \wedge \top \Rightarrow \alpha$

$\alpha \vee \top \Rightarrow \top$

$\alpha \wedge \bot \Rightarrow \bot$

$\alpha \vee \bot \Rightarrow \alpha$

$\alpha \wedge \alpha \Rightarrow \alpha$

Figure 2-16: Some Logical Reduce Rules of PROVER

A second type of rewriting that **Reduce** may carry out is that of function definitions. For example, the first reduce rule of figure 2-17 allows the prover to "understand" the set-theoretic symbol $\cap$. Notice that both types of rewriting carried out by **Reduce** are applications of $\rightarrow\!-E$ where the implication is an implicit hypotheses of the theorem stored as a reduce rule (this was discussed in chapter 1 section 1.3.4). The rewrite rules should be considered as extra hypotheses of the theorem.

$x \in y \cap z \Rightarrow x \in y \wedge x \in z$

$x \in y \cup z \Rightarrow x \in y \vee x \in z$

$x \cap x \Rightarrow x$

$x \cap \emptyset \Rightarrow \emptyset$

$x \cup \emptyset \Rightarrow x$

$\emptyset \subseteq x \Rightarrow \top$

$x \in \{y\} \Rightarrow x = y$

Figure 2-17: Some Database Reduce Rules of PROVER

Dave Schmidt [Schmidt 83] has investigated the possibility of using the definitions from set theory as new inference rules in a natural deduction prover. This indicates an alternative approach to the use of reduce rules and rewriting.

Define C is also a rewriting step but the set of rewrite rules that are available here is just the set of predicate definitions in the database. PROVER uses the definition of the predicate occurring in the conclusion to make a new conjecture. By only allowing a predicate to be replaced by its defining formula late in the proof attempt, this action is avoided for as long as possible. This is a good idea, since the definitions of terms are used to hide unwanted detail from a prover's consideration. The revelation of this detail should be delayed for as

long as possible, since access to the more complex formulae that are invariably introduced leads to more search for the prover. The name of the routine which performs the rewriting is *Define*. If no definition can be expanded this routine returns the original formula. As observed above, **Eq Ineq** is a special case of **Define C** which behaves as if equality was defined in terms of inequalities.

Peeking and Pairs

Peeking is the name given by Bledsoe to a feature of the prover which provides "sensible" expansion of predicate definitions. The rule allows that the definition of a predicate in the hypothesis of the goal may be expanded if it appears that this will allow the proof to progress. Here the assumption is that the program is "stuck" and that expanding the definition of the hypothesis is one of the few remaining courses that the prover may take. No similar guidance is available to PROVER when considering the expansion of definitions in the conclusion. Such definitions are used only when every other rule has failed to produce a proof. Thus expanding definitions is avoided if at all possible. If a "sensible" expansion of a predicate in the hypothesis can be found by **Peeking** this is made in preference to expanding a definition from the conclusion. The latter course of action is taken by **Define C** only as a final resort.

Peeking is invoked by **Match** only after an initial call to Hoa has failed to provide a proof, the conjecture is passed back to Hoa but this time with a flag set which allows the system to use peeking where it was not on the first pass. Expanding definitions can lead to many choices, since many predicates are defined, and often the definitions are very complex formulae. Guidance is provided by the program having the ability to "peek" at the definition of the hypothesis and decide on the basis of this information whether or not to expand the definition in the proof. To do this the program maintains a set of *peek property lists*. These are ordered pairs: the first element being the predicate which could be expanded, and the second element the set of all of the predicates occurring in the definition of that predicate. If one or more of these predicates occur in the conclusion of the theorem then the program will expand the definition accordingly, if not then no action is taken.

Example: If the prover has been asked to prove the theorem $a \subseteq b \vdash x \in a$ the peek facility will look at the peek property lists and find the pair $(\subseteq, [\in])$. On the basis of this the prover should decide to expand the definition to get $z \in a \rightarrow z \in b \vdash x \in a$.

The peek property list is an example of an *abstraction* of a formula. Abstraction is a very powerful technique which involves temporarily ignoring some of the detail of a structure. In the case of the peek property list the detail that is ignored is the connectives and terms of the formula. The gazing technique, described in chapter 4 uses abstractions extensively. [Plaisted

80, Plaisted 86] also reports some work on the use of abstraction in theorem proving which is discussed in chapter 6.

The **Pairs** (figure 2-3, Hoa rule 3) rule is very similar. This rule is also unavailable to an initial call to Hoa, and may only be used after the flag has been set. Here the program has a database which is consulted when the predicates of the hypothesis and conclusion of the theorem match but the terms cannot be unified. The database again consists of a set of ordered triples which have as their first element the predicate which is matched, the second element is the conjecture to be proved, and the third element is a theorem that may be proved in order to establish that the matching will go through. The second and third elements share variables, which ensures that when the goal is matched with the second element of the triple, the bindings of the variables is transferred to the new goal to be proved. Notice that more than one pairs property list may be available for each predicate, and so there is a choice to be made as to which will be used. Each pair property list records a formula which is *sufficient* but not *necessary* to prove the main conjecture.* Thus failure for one pair property list does not mean that the conjecture is not true. The routine *Pairs* performs the rewriting here, if no rewriting is possible then the routine returns **NIL**.

Example: If the prover has been asked to show $Countable(a) \vdash_h^? Countable(b)$ this cannot be done directly, since the arguments of *Countable* are both constants. The prover consults the pairs property list for *Countable* and finds (3) which asserts that to prove that a and b are both countable we must prove that there exists a bijection between them. The three parts of the pairs property list record: the predicate of the hypothesis and conclusion literals, the goal to be proved and the new goal respectively. The original goal is necessary to ensure the correct instantiation of the new goal.

$$(Countable,(Countable(x) \vdash^? Countable(y)), \quad \exists z.\ Bijection(z) \wedge Range(z,x) \wedge Domain(z,y)) \tag{3}$$

Notice that the inference rules of **Peeking** and **Pairs** depend on more than the major connective of the formula that is being manipulated. These rules refer to the form of the entire goal. The reason for this difference is that these rules attempt to implement heuristics which not only determine whether it is possible to carry out a particular inference, but also whether the inference is sensible. The criterion for "sensible" here is whether the application of the inference rule will lead to a proof. This judgement is necessarily heuristic, but the idea of the condition on the rule is to determine if a proof is likely.

In the case of **Pairs**, the lemmas that may be stored in the pair property list are all of the form that a particular property is preserved under a certain transformation. Such lemmas are very common in mathematics, and they are very useful in many proofs. The idea of the

---

*One could imagine a system which performed an analysis of lemmas as they are proved to determine whether they are of the correct form for the Pairs database. However, since PROVER does not have this capability, the database of Pairs Property Lists must be precoded by the implementors of the theorem prover.

condition that the predicates of the hypothesis and conclusion match is to detect that the desired goal is that some object(s) have a property, and that the hypothesis asserts that some other* object has the same property.

In the case of **Peeking**, the conditions check that the expansion of the definition will introduce the predicate which appears in the conclusion of the goal. Here the idea is that if the conclusion is about some objects having a property, and that by expanding the definition of a hypothesis we will obtain a new hypothesis about some other** objects having the same property, then this is likely to be a useful move to make.

Both the peeking and pairs rules are used after Hoa has failed to prove the conjecture. They are available to the prover only on Hoa's second attempt to prove the theorem, when the flag has been set. Thus the use of either peeking or pairs is considered to be a last attempt to prove the theorem after the more obvious proof methods have been used. The only rules which appear after this second call to Hoa are **Define C** and **Eq Ineq**, both of which effect the unguided expansion of predicate definitions in the conclusion of the goal.

The rules **Call Hoa** (figure 2-2, Imply rule 12) and **Call Imply** (figure 2-3, Hoa rule 5) serve to exchange control of the proof from one of the routines to the other. **Call Hoa** is invoked when almost all of the rules of Imply have been attempted. The rule says that the proof should be attempted by Hoa. If the first attempt at a proof by Hoa fails, then a flag is set which indicates that Hoa may use the **Peek** and **Pairs** rules and the proof attempted again. **Call Imply** is called by Hoa when the conclusion of the goal is either a conjunction or an implication. The effect of the rule is to ask Imply to provide a proof of the conjecture.

2.3.3. Summarizing PROVER

In this section I have described PROVER, the UT natural deduction theorem prover described in [Bledsoe & Tyson 75a]. The prover rules can be divided into three main classes: logical, database and special-purpose. The logical rules enable the prover to perform deductions within the $\mathcal{ND}$ system, although some of the rules have been modified to combine some rules together. Each of the $\mathcal{ND}$ rules have been given a *direction*, and this causes PROVER to use each rule exclusively in either forward or backward reasoning modes.

The database rules allow the prover to handle defined concepts and perform simplifications.

---

* The objects cannot be the same, otherwise the match rule would have applied.

** Notice that this time the literal that will be introduced may be the same as the conclusion; there is no way of telling since the rule considers only the predicates and not the terms of the definition and conclusion.

Mathematicians often use definitions to express concisely very complex formulae, the definitions are used to hide the detail of the statements that would otherwise be very complex. However it is often necessary to refer to the definitional formulae in order to complete proofs involving such defined terms. The replacement of defined terms by their defining formulae is carried out by the rewriting rule of inference, which is justified by the logical rule $\rightarrow\!-E$. That is, any proof involving an application of the rewriting rule of inference can be expressed involving only logical rules. The use of the databases for storing additional hypotheses allows PROVER to "concentrate" on the conjecture, but to retrieve extra hypotheses from the databases as and when they are required.

The special-purpose rules are very efficient ways of carrying out inferences about a particular class of formulae. The **Typelist** package of PROVER is such a rule which stores information about the objects that appear in the conjecture in a special form that the program can manipulate efficiently. The effect of this representation is that the prover can detect inconsistencies (contradictions) very quickly, and that the conjecture is simplified by the elimination of some hypotheses in favour of the efficient representation of the same information.

The **Peek** rule is very different from the other rules in that it attempts to assess whether a particular course of action is likely to be successful before carrying it out. Such a rule is necessarily heuristic, but its use enables the PROVER to prune much potential search from consideration.

In the next section I describe IMPLY, a later implementation of a natural deduction system which is very similar in spirit to PROVER, but has some interesting differences. A comparison of the programs is presented in chapter 3, and a comparison of each of the programs with *ND* is presented in section 2.5 of this chapter.

2.4. IMPLY

IMPLY is reported in [Bledsoe 83]. The spirit of IMPLY is the same as that of PROVER, but there are some differences in the design of the programs.

- The hypotheses are treated as a set in IMPLY rather than as a formula as in PROVER. This makes the logic of IMPLY closer to *GSC* than that of PROVER.
- IMPLY is not interactive*.
- IMPLY is not split into two routines as PROVER is. This makes IMPLY much easier to understand.

*Bledsoe has recently implemented an interactive version of IMPLY which is not considered here. This program has not been the subject of any report.

- Some of the database and special-purpose rules have not been implemented in IMPLY.

The main difference between PROVER and IMPLY is in the treatment of the hypotheses of the goal. In PROVER the treatment is very like that of *ND*, where the hypothesis is a single formula. In IMPLY the hypotheses are represented as a set of formula as in *GSC*.

Partly as a consequence of the simplified structure of IMPLY, the number of rules that the prover has is much reduced (IMPLY has 20, PROVER 28). Figures 2-18 and 2-19 give the rules of the IMPLY program. Like section 2.3 this section is divided into two parts: in 2.4.1 I will describe the logical rules of IMPLY and then the non-logical rules will be presented in 2.4.2.

Definition 32: Following the notation of the previous section, I will write $\Gamma \vdash_I^? \alpha$ to denote the goal of proving α from Γ using IMPLY.

2.4.1. The Logical Rules of IMPLY

Since both PROVER and IMPLY are based on natural proof systems their rules have much in common, but they differ in some important respects. The complete set of rules is presented in tables 2-18 and 2-19.

$H \vdash_I^? C$

RuleName	Condition	Action	Result
1. (Truth)	$\bot$ is a conjunct of H		**T**
	$C \equiv \top$		**T**
2. (Ancestry)		$\theta := Ancestor(C)$	
	$\theta \neq$ **NIL**		θ
	$H\text{-}G\text{-}F(C) \neq$ **NIL**		**NIL**
3. (And Split)	$C \equiv (A \wedge B)$	$\theta := H \vdash_I^? A$	
	$\theta \neq$ **NIL**	$\lambda := H \vdash_I^? B\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
4. (Cases)	$H \equiv (A \vee B)$	$\theta := A \vdash_I^? C$	
	$\theta \neq$ **NIL**	$\lambda := B \vdash_I^? C\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
5. (Reduce)		$NewH := Reduce(H)$	
		$NewC := Reduce(C)$	
	$NewH \neq H$, or $NewC \neq C$		$NewH \vdash_I^? NewC$
6. (Or Fork)	$C \equiv A \vee B$	$NewC := AndOut(C)$	
	$NewC \neq C$		$H \vdash_I^? NewC$
	$NewC = C$	$\theta := H \wedge \neg B \vdash_I^? A$	
	$\theta \neq$ **NIL**		θ
	$\theta =$ **NIL**		$H \wedge \neg A \vdash_I^? B$
7. (Promote)	$C \equiv (A \rightarrow B)$		$H \wedge A \vdash_I^? B$
8. (Prove Equiv)	$C \equiv (A \leftrightarrow B)$	$NewC := (A \rightarrow B) \wedge (B \rightarrow A)$	
			$H \vdash_I^? NewC$
9. (= C)	$C \equiv x = y$	$\theta := Unify(x,y)$	
	$\theta \neq$ **NIL**		θ
10. (Flip C)	$C \equiv \neg A$		$H \wedge A \vdash_I^? \bot$
11. (Inequality)	see page 62		
12. (Match)	$NewH$ a conjunct of H		
		$\theta := Unify(NewH,C)$	
	$\theta \neq$ **NIL**		θ

Figure 2-1: The Rules of IMPLY - Part 1

$H \vdash_I^? C$

RuleName	Condition	Action	Result
13. (Back Chain)	$(A \rightarrow B)$ a conjunct of H,		
		$\theta := Unify(B,C)$	
	$\theta \neq$ **NIL**	$\lambda := H \vdash_I^? A\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
13O. (Back Chain O)			
	$(A \rightarrow B)$ a conjunct of H,		
	B' is a disjunct of B,		
		$\theta := Unify(B',C)$	
	$\theta \neq$ **NIL**	$B'' := Delete(B',B)$,	
		$\lambda := H \vdash_I^? A\theta \wedge \neg B''\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
13S. (Simp Imp)	$(A \rightarrow (B \rightarrow D))$ a conjunct of H,		
	$NewH := Replace((A \rightarrow (B \rightarrow D)),((A \wedge B) \rightarrow D))$		
			$NewH \vdash_I^? C$
14. (Sub =)	$x = y$ is a conjunct of H,		
		$\nu := Minus\text{-}On(x,y)$	
	$\nu = 0$		**NIL**
	ν is a number		**T**
	ν is not a number	$x' := Choose(x,y)$	
		$y' := Other(x,y)$	
		$\theta := H_{x'}^{y'} \vdash_I^? C_{x'}^{y'}$	
	$\theta \neq$ **NIL**		θ
14H. (Back Chain =)			
	$(A \rightarrow x = y)$ is a conjunct of H,		
		$x' := Choose(x,y)$	
		$y' := Other(x,y)$	
		$\theta := H_{x'}^{y'} \vdash_I^? C_{x'}^{y'}$	
	$\theta \neq$ **NIL**	$\lambda := H \vdash_I^? A\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
15. (Flip H)	$\neg A$ is a conjunct of H,		
		$NewH := Delete(\neg A,H)$	
			$NewH \vdash_I^? C \vee A$
16. (Define C)	$NewC := Define(C) \neq C$		$H \vdash_I^? NewC$
17. (IMPLY Fail)			**NIL**

Figure 2-2: The Rules of IMPLY - Part 2

IMPLY and PROVER differ in their handling of hypotheses and of the database and special-purpose rules. The logical rules dealing with the conclusion (**And Split**, **Reduce**, **Or Fork**, **Promote**, **Prove Equiv**, **= C**, **Flip C**, **Inequality** (figure 2-18, rules 3, 5, 6, 7, 8,

9, 10 and 11), and **Define C** (figure 2-19, rule 16) all apply under identical conditions to that of PROVER, and each produce the same subgoal as the corresponding rule in PROVER. However since there is only one proof routine in IMPLY this routine is always called to perform the proof of the new goal. Thus the rule **Flip C** which used to make a recursive call to Hoa, now makes a call to IMPLY. IMPLY **Fail** (figure 2-19, rule 17) is exactly the same as the failure rules of PROVER. Case 2 of the **Truth** rule (figure 2-18, rule 1) is the same as case 2 of the **Truth** rule of PROVER.

The treatment of hypotheses in IMPLY is much closer to that of $\mathcal{GSC}$ than $\mathcal{ND}$. In $\mathcal{GSC}$ both the hypothesis and conclusion are sets of formulae with each member of the set being implicitly conjoined with the others. This has the effect of allowing all of the conjuncts to be treated equally. In $\mathcal{ND}$ the hypothesis is a single (nested) conjunction and this means that hypotheses at a lower level of nesting cannot be reached as easily as those nearer the surface. The PROVER rule **And Fork** has the function of liberating the conjuncts which are deeply nested, since it allows PROVER to weaken the hypothesis of the goal, and attempt to prove the conclusion from each of the conjuncts of the hypothesis separately. In IMPLY the hypothesis is treated as in $\mathcal{GSC}$, and there is a routine for selecting conjuncts from the set of hypotheses.

Case 1 of **Truth** and each of the rules **Match**, (figure 2-18, rule 12) **Back Chain**, **Simp Imp**, **Back Chain O**, **Sub =**, **Back Chain =**, and **Flip H** (figure 2-19, rules 13, 130, 13S, 14, 14H and 15), select conjuncts from the hypothesis set, and then apply the rule to some conjunct that has the appropriate form. The effect of this is that the hypothesis never has to be explicitly split. By allowing the prover to select a conjunct of the appropriate form, the splitting of the hypothesis has been eliminated.

$$(A \wedge B) \wedge (C \wedge D) \vdash^{?} C \qquad (H)$$

For example, to prove the goal (H) in PROVER the steps shown in figure 2-20 are necessary.

$(A \wedge B) \wedge (C \wedge D) \vdash^{?} C$

$(A \wedge B) \vdash^{?} C$

$A \vdash^{?} C$ **fails**

$B \vdash^{?} C$ **fails**

$(C \wedge D) \vdash^{?} C$

$C \vdash^{?} C$ **succeeds: T**

Figure 2-20: Proof of (H) using PROVER

IMPLY proves the same goal immediately since the conjecture is represented as in (I).

$$\{A, B, C, D\} \vdash^{?} C \qquad (I)$$

Notice that the sequents that fail in the PROVER proof, are not explicitly attempted by IMPLY, but they are attempted implicitly in the selection of the conjunct to match. When IMPLY is applying **Match** it has to select each of the conjuncts of the hypothesis in turn to attempt the match until it finds one which succeeds. In IMPLY though, this work is carried out without constructing explicit subgoals.

IMPLY's **Cases** rule is an exception to this handling of hypotheses. This rule requires that the hypothesis is a disjunction before it will be used. In PROVER the subgoal that results from an application of **Cases** to a sequent $A \vee B \vdash^{?} C$ is $\top \vdash^{?} (A \rightarrow C) \wedge (B \rightarrow C)$. We saw in the previous section how this new subgoal forced the next two applications of inference rules so that the eventual proofs that were attempted were of $A \vdash^{?} C$ and $B \vdash^{?} C$. In IMPLY these goals are suggested immediately, thus saving the application of the two rules which are used in PROVER.

The rule **Simp Imp** does not have an equivalent in PROVER. The rule allows the simplification of a conjunct of the hypothesis of the form $(A \rightarrow (B \rightarrow C))$ to the equivalent form $((A \wedge B) \rightarrow C)$. The presence of this rule increases the power of the prover, since in both PROVER and IMPLY the **Back Chain** rule attempts to match the conclusion of the goal with the consequent of an implication in the hypothesis. This will result in failure if the conclusion is C, and the unsimplified hypothesis is present, but will succeed if the implication has been simplified. Since this is essentially a rewriting step this rule could have been added to IMPLY as a reduce rule. This approach would have the property that the conclusion could be rewritten to this form too, since conclusions are rewritten by the reduce rule as well as the hypothesis.

Another difference between the back-chain rule of IMPLY and that of PROVER, is that PROVER used the routine **Ands** to match the consequent of the hypothesis with the conclusion. Thus, if the hypothesis were: $A \rightarrow (B \wedge C)$, and the conclusion were C, then **Back-Chain** would be used. In IMPLY the conclusion and consequent are matched by the unification routine, and so the consequent must be completely unifiable with the conclusion.

Back Chain O (figure 2-19, IMPLY rule 13O) and **Back Chain =** (figure 2-19, IMPLY rule 14H) are special cases of the back-chaining technique. The first applies if the consequent of the implication is a disjunction where one of the disjuncts unifies with the conclusion of the goal. In this case the matching disjunct is deleted from the disjunction, and the remaining disjuncts are negated and conjoined to the antecedent of the implication. *Delete* is the routine which will delete a disjunct from a disjunction. This new conclusion is proved from

the hypotheses. **Back Chain** $=$ applies if the consequent of some implication in the hypothesis is an equality literal. If it is then the system selects the simplest of the equal terms, and replaces the other with this throughout the goal. The selection of the simplest term is carried out by the routines *Choose* and *Other* which were described in section 2.3.1 on page 41. If the sequent now follows by IMPLY then the system attempts a proof of the antecedent of the implication. Otherwise the rule fails.

Another new rule is called **Ancestry**. It comes in two parts: The prover reports success if it can unify the current goal with the negation of an ancestor goal of the current goal (that is with a goal that lies on the part of the proof tree between the root and the current goal). The proof tree is searched by a routine call *ancestor*. The other part reports failure if the current goal is the same as an ancestor goal and no new hypotheses have been added to the goal since the ancestor. This situation is detected by a routine called *Higher-Goal-Failure* (abbreviated to *H-G-F* in table 2-18). The use of higher-goal-failure prevents loops from occurring in the search for a proof.

Equiv H is not a rule of IMPLY, but **Prove Equiv** remains. Both these rules are present in PROVER, but as observed in section 2.3.1 these rules may never be used.

IMPLY incorporates *smart backtracking*. This is an augmentation of the unification routine which enables the specification of illegal bindings for variables. The problem is that although two subgoals are solved independently, they may have variables in common. Recall the example (G), presented on page 44, recalled below, where the shared variable x makes the solution to the goals dependent. Imply is unable to detect that certain solutions for one of the subgoals will preclude a solution for the other. In IMPLY however there is a technique for dealing with this problem.

$$P(a) \wedge P(b) \wedge Q(b) \vdash^{?} P(x) \wedge Q(x) \qquad (G)$$

The **And Split** rule of IMPLY says that each conjunct of $A \wedge B$ should be proved. Suppose that B cannot be proved after the substitution resulting from the proof of A has been applied, but it can be proved before the substitution is applied. Then the system analyses the substitutions returned from the proof of A and B and determines where the conflict lies. The system has a smart matching routine which will unify formulae without making bindings which are specified in an EXCLUDE list. By using this unification routine and specifying that the binding causing conflict is excluded, the prover attempts the proof again. In the case of (G), the alternative proof of $P(x)$ which binds b/x will be found and then the proof will be completed.

While the composition of substitutions is defined as left, IMPLY is able to return

generalized bindings for a variable. Consider goal (J). The variable x in the hypothesis has a universal interpretation; that is it appears universally quantified in the original, unskolemized conjecture. Thus the conjecture is true. If P is true for all x, then it is certainly true for a and b.

$$P(x) \vdash^{!} P(a) \wedge P(b) \qquad (J)$$

The provers will attempt to prove this goal by proving each of the conjuncts separately, and in the proof of the first subgoal will produce the binding a/x. This substitution must be applied to the conclusion of the second conjunct, but not to the hypothesis of this subgoal. If it were then the proof of the second subgoal is prevented, since a will not unify with b. This failure is due to the inability to perform copying which is discussed in relation to completeness in section 2.5. PROVER and IMPLY can detect this situation and permit the value $\{b/x, a/x\}$ as a legal result for this goal.

This is logically correct since the variable x derives by skolemizing a universally quantified formula. The universally quantified formula $\forall x.P(x)$ can be thought of as an abbreviation for the (possibly infinite) conjunction $P(t_1) \wedge P(t_2) \ldots$, where the t_i stand for the members of the universe under discussion. One can think of the two subgoals being proved from two of the conjuncts in this conjunction, and thus that the two occurrences of x can stand for different objects. Of course, generalized bindings like this can arise only in this way. IMPLY cannot return $\{1/x, 0/x\}$ when given $x{=}x \vdash^{?} 1{=}0$ to prove. This is prevented since the conclusion of the goal is not a conjunction, and so the incompatible bindings for x would have to arise from the proof of the same literal.

A number of alternatives to this approach to the problems of splitting are possible. For example, it would be possible make a copy of the hypothesis when performing such a split, and then rename the variables in the copied hypothesis, to ensure that distinct variables occur in the two subgoals. The work of Nevins is an example of an alternative solution to this problem [Nevins 75b].

2.4.2. The Non-Logical Rules of IMPLY

Some of the database rules of PROVER are not implemented as part of IMPLY. **Reduce** and **Define C** are implemented exactly as in PROVER and so the logical rewriting steps which may be carried out by **Reduce** and the expansion of definitions in the conclusion of the theorem are possible within IMPLY. However **Peek** and **Pairs** are not implemented in IMPLY. The omission of these rules makes IMPLY much less powerful than PROVER.

The omission of **Peek** means not only that IMPLY is unable make sensible use of the

definitions of predicates which appear in the hypothesis of the goal, but that is unable to make use of these definitions at all. This is because there is no rule for unconditionally expanding the definitions of predicates in the hypothesis as there is for conclusions (**Define C**).

Predicate definitions may be added to the hypothesis of the goal to be proved, and then the implementation of chaining would allow IMPLY to make definitional expansions. This approach would cause the definitional expansion to be made under one of two conditions: when a literal involving a defined term is promoted into the hypothesis, and when the conclusion matches the definitional formula of some term. In the first case definitions would be expanded more often than by **Peek**, since any literal that is added to the hypothesis would be expanded in terms of the definition of its predicate. The second case, when the expansion is performed by backward chaining, allows the prover to perform the expansion much less often than by peeking. This is because **Peek** requires only that the expansion of the definition introduces the predicate that appears in the conclusion. Backward chaining requires that the definitional formula unifies with the conclusion of the goal: a stronger condition. However, stating the definitions as explicit hypotheses does not allow IMPLY to implement the peeking heuristic, and thus enables less control over the definitional formulae. Reiter's prover [Reiter 76] which is described in chapter 6 takes this approach to definitions.

The omission of **Pairs** is less serious since the class of lemmas that could be stored in the pair property lists is quite restrictive and can only be used in quite rare circumstances. However this leaves the prover unable to apply this class of lemmas in the same way. Again, these lemmas could be explicitly conjoined to the hypotheses of the theorem, but this approach would not enable such strong guidance to be placed on their use. The condition on the use of a pairs lemma in PROVER is that there is a hypothesis and conclusion which have the same predicate but the atomic formulae do not match. If the lemmas were explicitly conjoined to the hypotheses of the conjecture then they could be used in back- or forward-chaining when the conclusion (or hypothesis) was of the appropriate form but when no corresponding hypothesis (conclusion) were present.

The **Typelist** and **Eq Ineq** rules of PROVER are not implemented as part of IMPLY, however **Inequality** is implemented. The details of this rule are the same as rule 11 of Imply (see page) except the the resulting sequent is proved by IMPLY and not by Hoa as in PROVER.

The rules which deal with equality literals: **Sub** $=$ and $=$ **C** are present in IMPLY. **Sub** $=$ has been changed to select an equality literal from the hypothesis, rather than requiring that the entire hypothesis is such a literal in accordance with the view of the hypothesis as a set of conjuncts.

2.4.3. Summarising IMPLY

In this section I have described IMPLY, the theorem prover described in [Bledsoe 83]. This program has many of the rules of PROVER, but some have been considerably changed. The most important change is that IMPLY does not split conjunctions in the hypothesis of goal explicitly, but that the rules allow the prover to select appropriate conjuncts at certain times. This difference makes IMPLY much closer to *GSC* than PROVER is, since the hypothesis is treated as a set of formulae rather than as a single formula. This leads to the elimination of much of the search caused by **And Fork** in PROVER.

Some of the database rules of PROVER are not present in IMPLY, and this causes IMPLY to be much weaker that PROVER. In particular, the peeking and pairs rules of PROVER are not present, and no ability to perform definitional expansion of predicates occurring in the hypothesis of goals is available. This weakening can be overcome by stating conjectures with necessary lemmas and definitions explicitly conjoined, however this solution to the problem does not allow the use of these explicit hypotheses to be controlled and used sensibly as they were by the rules of PROVER.

2.5. The UT Provers and Natural Deduction

In this section I discuss the relationship between the UT Provers and Natural Deduction. The relationship is not merely one of translation from logic formalism into LISP, although for some of the rules the relationship between the two is quite obvious. Instead many features have been added to make the provers a viable theorem proving system. In particular the inference rules are controlled in novel and powerful ways.

Skolemization, unification and rewrite rules have no equivalent in natural deduction. In the natural deduction system dependencies of objects (variables and constants) on one another are recorded implicitly in the proof tree and the logician is meant to examine the tree to decide whether a particular application of a quantifier rule is legal. This is not possible for the automatic prover as the tree in which the dependencies lie may not yet be constructed (because the prover runs some of the rules backwards). Thus the dependencies have to be recorded some other way, and skolemization, where variables are replaced by terms which explicitly record the dependencies, is an answer. The use of skolemization neccesitates the use of unification: Since the skolemization step replaces variables with terms then it is necessary to be able to determine when two terms can be made the same. The unification step is one of the ways in which the UT provers actually infer conclusions - equivalent to saying that the prover has the match axiom of *GSC* (figure 1-35 on page 14).

The rewrite rule package has the ability to perform two sorts of rewrite: logical, and database. The logical rewrite rules allow the prover to simplify formulae according to the equivalences of logic. For example,

$$\alpha \vee \alpha \Rightarrow \alpha$$

This facility is not available in natural deduction, although proofs of such identities can easily be found.

The non-logical rewrite rules handle the definitions of functions, including rules like:

$$x \in y \cap z \Rightarrow x \in y \wedge x \in z$$

These rewrite rules are dependent on the theory that the prover is working in and are not part of the deduction system. Rather they represent extra hypotheses for the goal, which are left implicit since they are of general use. The rewrite rules, with the predicate definitions handled by the **Peek** and **Define C** rules of PROVER, enable the provers to work in a particular theory without having to record the definitions of the symbols explicitly as hypotheses to the proof.

One of the chief differences between PROVER and IMPLY is the way in which hypotheses are treated. The rule that PROVER has for matching requires that the entire conclusion and the entire hypothesis unify. However it is only necessary for some conjunct of the hypothesis to be unifiable with some disjunct of the conclusion before the goal is proved. The work involved in splitting the hypothesis and conclusion so that they completely match has been avoided in IMPLY by allowing the prover to select conjunctions without explicitly splitting the hypothesis. This has necessitated changes to many of the rules which deal with the hypothesis. The alteration is however quite routine. In PROVER the rule for backward chaining, for example, requires that the hypothesis is an implication and that the consequent of the implication unifies with the conclusion of the goal. In IMPLY the rule applies if there is *a conjunct of* the hypothesis which has the same property. The new hypothesis is the remaining conjuncts, and the new conclusion is the antecedent of the implication.

While PROVER has many features of a sequent based system it is also very close to natural deduction. The change from PROVER's handling of the hypotheses to IMPLY's handling represents a step away from a natural deduction toward a sequent based system. In section 1.2.2 of chapter 1 I described, *GSC*, a sequent based system, and this system manipulated a *set* of hypotheses, and a *set* of conclusions. In PROVER the system manipulates a pair of formulae, and in IMPLY the hypothesis may be viewed as a set while the conclusion remains as a formula.

Both PROVER and IMPLY are incomplete. That is, there are theorems which cannot be shown so by these programs. *ND* and *GSC* are known to be complete so the lack of completeness must originate in the rules of the theorem provers. In [Bledsoe & Tyson 78] Bledsoe identifies the three causes of the incompleteness of PROVER:

- Trapping,
- Inability to copy hypotheses, and,
- Weak back chaining.

Trapping and the inability to copy hypotheses have been discussed above. In IMPLY these problems are tackled by the use of "smart backtracking" and the exclusion of bindings respectively (see pages 60 and on for a discussion of these techniques).

Weak back chaining is the name given to a problem which arises in both PROVER and IMPLY. Back chaining is the technique of matching the conclusion with the consequent of a hypothesis implication (see page 45). If this can be done then a proof of the antecedent of the implication is sought.

$$\dfrac{\dfrac{\overset{\eta}{\prod}}{\alpha} \qquad \alpha \rightarrow \beta}{\dfrac{\overset{\beta}{\sum}}{\chi}}$$

Figure 2-21: Back-Chain

In PROVER Σ is found by using the routine *Ands*, and in IMPLY a simple unification is attempted. In general, the full theorem prover might be used to provide this subproof, and using a weaker proof method for doing this causes some possible proofs to be missed. The reason that Bledsoe gives in [Bledsoe & Tyson 78] for not using the full theorem prover to provide this proof is that, in the cases where the consequent of the hypothesis and the conclusion have no relation to one another, too many resources are spent in attempting this proof. This is a general reason for not placing too much emphasis on completeness of automatic theorem provers; as Bledsoe and Tyson say:

> If we ever expect to prove really difficult theorems in mathematics we must not strangle the mechanism that does it by making sure that it handles every case. Rather we believe ... that it should be allowed to fail on a few cases so that it can succeed on a number of others, especially the hard ones. (Page 79 of [Bledsoe & Tyson 78]).

2.6. Summary

The UT provers [Bledsoe & Tyson 75a, Bledsoe 83] are well-known automatic natural deduction theorem provers. They are described in sections 2.3 and 2.4. In the description of these programs I have classified the inference rules that the programs have into three types. The first type: logical rules, have a very close connection with the rules of $\mathcal{ND}$. The second type, which I call database rules, allow the provers to manipulate definitions in a manner which is very similar to that adopted by human mathematicians. The prover records hypotheses which are global to the theory in which the prover is working, in a number of databases. These can be accessed by certain routines of the system, and used to rewrite the conjecture. Finally, the provers have access to special-purpose rules which allow the system to perform certain classes of inference very efficiently. Both the database and special-purpose rules can be justified by the rules of $\mathcal{ND}$, and the axioms of the particular concepts that they manipulate. These inferences could, in principle, be performed by the logical rules. However there is some advantage to implementing these rules separately, both in terms of efficiency and the manner in which the proof proceeds.

PROVER and IMPLY are very similar programs, but there are significant differences between them. Most of these differences are the result of a different method of handling hypotheses: in PROVER the hypothesis of a goal is viewed as a formula, and in IMPLY as a set of formulae. This alternate view of the goal does not significantly alter the power of the prover, however the proofs that are produced by IMPLY can be much more natural than those produced by PROVER.

The provers both force the rules of natural deduction to be used in one of two ways: either forward or backward. This means that the provers can always ensure that the subgoals that an application of an inference rule produces are always simpler than the original goal. This is important since if the goal becomes more complex, the prover may loop indefinitely. If the goals are genuinely simpler, then the proof must eventually terminate.

Other examples of natural deduction provers are described in [Brown 78, Nevins 74, Pastre 77, Reiter 76, Cvetkovic & Pevac 83]. Discussion of these provers, where relevant, is deferred until chapter 6.

Chapter 3
RUT:
A Rational Reconstruction of the UT Theorem Provers

3.1. Overview

In this chapter I continue the investigation into the UT provers of Bledsoe and his team at the University of Texas [Bledsoe & Tyson 75a, Bledsoe 83]. In chapter 2 it was noted that the descriptions of the provers were different, as one would expect since they were published 8 years apart. The provers have not merely been extended but some techniques have been omitted from the later program. In this chapter I describe the **R**econstructed **UT** **T**heorem Prover (RUT), a rational reconstruction of these provers which I have implemented. It is *rational* in that it is not a *copy* of either of the UT provers described in the literature but rather the implementation of the best features of those two programs.

In addition to the best features of PROVER and IMPLY, RUT has some novel features including:

- The explicit representation of the proof being performed,
- A friendly user interface making communication with the system fairly natural
- Inference rules which implement a slightly different logic to that of either IMPLY or PROVER, and,
- A backtracking interpreter for the inference rules which allows the RUT to select a different inference rule if the first inference rule that is chosen fails to lead to a proof.

It is in the combination of the ideas of the two provers that the reconstruction is most informative. It was necessary to compare the inference rules of the two UT provers to determine where the source of the power of the programs lay, and to retain these features in the rationalized version. Due to the size of the UT provers it was also necessary to omit some features from the reconstruction. Those features that have been omitted form self-contained units of the original provers, and I will show that RUT is more powerful in the areas that I have implemented than either of the UT provers which it rationalizes.

The description of RUT in this chapter differs slightly from those in [Plummer 85a, Plummer 84]. This is because RUT has been extended and rationalized still further since the publication of those papers. The description in this chapter is actually of the "rut" mode of the VOYEUR theorem proving system. This system emulates RUT in this mode, but can also emulate GAZER, the prover discussed in chapter 5. The differences between the description in this chapter and that of [Plummer 85a, Plummer 84] are chiefly the extension to the logic described here in 3.4, and the set of interactive commands which may be used to alter the course of the proof. These are described here in 3.5.

3.2. RUT and the UT Provers

In building RUT most of the better features of both PROVER and IMPLY have been retained. The extra-logical handling of the PROVER system, the rule-base of IMPLY, and a new interactive system have been incorporated into RUT in order to construct a prover combining the best of both UT systems. Additional features, for example, the explicit representation of the proof being constructed, and a slightly improved logic have also been implemented in RUT. RUT is therefore more powerful than either of the programs that it reconstructs.

It should be noted that the following features of PROVER and IMPLY systems have not been reconstructed due to the size of the UT provers and lack of time for the reconstruction.

- The **Pairs** technique for handling a limited class of lemmas (from PROVER)
- The **Inequality** handling package (from PROVER).
- The **Ancestry** rules (from IMPLY)

RUT includes the implementation of the peeking technique of PROVER, but not the pairs technique. This means that while RUT is able to handle predicate definitions in a sensible way, no guidance on the use of lemmas is available. Also, as in PROVER and IMPLY, no guidance is provided for function definitions in RUT. Both lemmas and function definitions have to be stated by the user as rewrite rules for the **Reduce** rule to use. This means that whenever **Reduce** is called any rewrite rule which may be applied will in fact be used. This is an important problem for RUT and the UT provers. This has the effect that conjectures are rapidly rewritten into their simplest terms, even when this is not necessary to carry out the proof.

The omission of the pairs feature of PROVER means that RUT is completely unable to guide the use of lemmas of any form. This is not a major deficiency over the PROVER system since the class of lemmas for which the pairs technique can provide guidance is very limited. The inability to guide lemmas is a practical, but not a theoretical difficulty. In order to utilize

previously proved lemmas the user may either conjoin them to the hypotheses of the theorem or store them as reduce rules. The first of these options is not desirable for two reasons:

- It is then necessary for the user to know which lemmas are necessary for the proof, in advance of the proof being carried out.
- It is not possible, within any of the provers discussed here, to specify control over the use of lemmas when they are presented as additional hypotheses.

In chapter 4 I describe a new technique which may be used to control the use of definitions and lemmas. The technique, gazing, leads to an improved handling of such information when they are cast as rewrite rules as they are in the UT provers and RUT. While **Peeking** appears in RUT, gazing subsumes both the **Peeking** and **Pairs** techniques.

The inequality reasoning package, including **Typelist** and **Eq Ineq** have been omitted since their reconstruction is unlikely to yield new insight into the technique.

As in both UT provers the conjecture to be proved is stated by the user in the full first-order predicate calculus. As a preprocessing step the theorem is skolemized to eliminate quantifiers. A proof of the skolemized formula is then attempted by the inference engine.

3.3. The Rules

Like the UT provers RUT has a number of rules of inference. Each rule consists of a condition part, an action part and a result to be returned. The condition part of a rule indicates the circumstances under which the rule may fire, and the action part states what actions are necessary to complete the proof. Unlike the UT provers, RUT does not return as the value of a rule the substitution necessary for the theorem to be true, rather the result is the *proof* of the theorem. So, in RUT the proof is explicitly created rather than performed as in the UT provers. This means that when RUT completes the proof there is an explicit representation of the proof structure that may be used in any way that the user desires. In the UT provers, the representation of the proof tree is implicit in the LISP stack, and the user may not access this structure. This feature of RUT is considered to be advantageous because it enables the manipulation of proofs by other systems; for instance, performing analogical reasoning about proofs.

The RUT interpreter tests each of the rules in turn until it finds one whose condition part is met by the current goal. When such a rule is found, the actions are carried out. In most cases, the action of a rule results in the creation of part of the proof. This new proof part is inserted in the relevant place in the overall proof. The new part of the proof will have

"holes" which require filling with further proof parts. The proofs of these goals fill in the proof, just as some of the rules of $\mathcal{ND}$ require subproofs to complete the application of the rule. The goals associated with these holes are added to the agenda of the prover, and when the proof of the subgoal is complete the prover fills in the main proof. In this way, by repeatedly creating parts of the proof with an application of an inference rule, details of the proof are filled in. Of course there are rules which do not leave subproofs to be filled in, these rules cause the termination of the proof process.

RUT has the ability to perform backtracking on failure. That is, when RUT has applied an inference rule to a goal, G, yielding some subgoals which cannot be proved, RUT continues to consider the rules that may be applied to G. If another rule can be found for which the conditions are true, this is applied and the new subproofs attempted. In both PROVER and IMPLY the choice of a particular inference rule is binding (see chapter 2 section 2.2). Once a particular rule has been applied to a goal, the goals suggested by the rule must succeed for the whole proof to succeed. The ability for backtracking makes RUT a more powerful prover, and means that the ordering of the proof rules in the program is less critical for RUT.

3.4. Disjunctive Conclusions

The major extension to the logic of IMPLY that is implemented in RUT is the handling of disjunctive conclusions. It was noted above that the handling of conjunctive hypotheses in IMPLY is made possible by the fact that conjunction is both commutative and associative. This makes it possible to treat a conjunction as a set of formulae. Disjunction has exactly the same properties, and in $\mathcal{GSC}$ the conclusion of a sequent is represented as a set of formulae, implicitly disjoined. The step from PROVER's to IMPLY's handling of hypotheses can be seen as a step from an $\mathcal{ND}$ formalization to a $\mathcal{GSC}$ formalization of the logic. In RUT, the obvious further step of handling the conclusion as in $\mathcal{GSC}$ is also taken. This leads to the following change in the rules which deal with the conclusion.

First of all, the IMPLY rule which splits disjunctive conclusions, **Or Fork** is no longer required (as **And Fork** is not required by IMPLY). Each rule that IMPLY has which deals with the conclusion of the sequent has to be altered to be used if some disjunct of the conclusion has the condition, rather than the entire conclusion having the condition. Also, when formulae are moved into the conclusion, the conclusion is updated by disjoining the new formula with the existing conclusion rather than replacing it. The complete rule set of RUT is given in figures 3-1 and 3-2.

Definition 33: A call to RUT, with hypothesis H and conclusion C will be written $H \quad \vdash_R^{\varphi} \quad C$.

As a result of this change to the rule base, RUT is never forced to make an or-choice in the application of an inference rule. Whenever two subgoals are suggested by an inference rule, both of the subproofs must be completed before the main goal is proved. However the ability to remake the choice of rule to apply, introduces or-choices between inference rules.

RUT(H,C)

Rule Name	**Condition**	**Action**	**Result**
1. (Truth)	$\bot$ a conjunct of H		**T**
	$\top$ a disjunct of C		**T**
2. (And Split)	$A \wedge B$ a disjunct of C		
		$NewC := Delete(A \wedge B, C)$	
		$\theta := H \vdash_R^? A \vee NewC$	
	$\theta \neq$ **NIL**	$\lambda := H \vdash_R^? B \vee NewC$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
3. (Cases)	$A \vee B$ conjunct of H		
		$NewH := Delete(A \vee B, H)$	
		$\theta := A \wedge NewH \vdash_R^? C$	
	$\theta \neq$ **NIL**	$\lambda := B \wedge NewH \vdash_R^? C$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$
4. (Reduce)		$NewH := Reduce(H)$	
		$NewC := Reduce(C)$	
	$NewH \neq H$ or $NewC \neq C$		$NewH \vdash_R^? NewC$
5. (Promote)	$A \rightarrow B$ a disjunct of C		
		$Disjunct := Delete(A \rightarrow B, C)$	
		$NewC := Andout(B) \vee Disjunct$	
		$IH := Forward\text{-}Chain(A, H)$	
		$NewH := Orout(IH)$	
			$NewH \vdash_R^? NewC$
6. (Equal C)	$A = B$ disjunct of C		
		$\theta := Unify(A, B)$	
	$\theta \neq$ **NIL**		θ
7. (Flip C)	$\neg A$ a disjunct of C	$\theta := H \wedge A \vdash_R^? C$	
	$\theta \neq$ **NIL**		θ
8. (Match)	H' a conjunct of H		
	C' a disjunct of C	$\theta := Unify(H', C')$	
	$\theta \neq$ **NIL**		θ
9. (Back Chain)	$A \rightarrow B$ a conjunct of H		
	C' disjunct of C	$\theta := Unify(C', B)$	
	$\theta \neq$ **NIL**	$\lambda := (Delete(A \rightarrow B, H) \vdash_R^? A)\theta$	
	$\lambda \neq$ **NIL**		$\theta \circ \lambda$

Figure 3-1: The Rules of RUT - Part 1

RUT(H,C)

Rule Name	Condition	Action	Result
10. (Sub =)	$A = B$ conjunct of H	$y := Choose(A,B)$, $x := Other(A,B)$ $\theta := Delete(A = B,H)^x_y \vdash^?_R C^x_y$	
	$\theta \neq$ **NIL**		θ
11. (Flip H)	$\neg A$ a conjunct of H	$NewH := Delete(\neg A,H)$ $\theta := NewH \vdash^?_R A \vee C$	
	$\theta \neq$ **NIL**		θ
12. (Peek)	A an atomic conjunct of H $Peek(A,C) \neq$ **NIL**	$\theta := Define(A) \wedge Delete(A,H) \vdash^?_R C$	
	$\theta \neq$ **NIL**		θ
13. (Define C)	A an atomic disjunct of C	$C' := Define(C)$	
	$C' \neq$ **NIL**	$\theta := H \vdash^?_R C' \vee Delete(A,C)$	
	$\theta \neq$ **NIL**		θ
14. (Fail)			**NIL**

Figure 3-2: The Rules of RUT - Part 2

3.5. Interaction

When given a goal to prove, PROVER attempts the proof for a pre-specified time. When this time is over the prover either returns success, or reports the still unproved subgoals. In RUT a different paradigm of interaction has been adopted, here the system reports to the user before the application of each inference rule. The user is able to issue commands which may alter the course of the proof. One of these commands enables the user to state which of the rules from the rule base is to be applied to the goal. RUT will ignore all of the others and attempt to apply this rule. In this way the user may prevent the application of a rule which will to lead to failure in favour of a useful rule which appears later in the rule ordering. Below is a complete list of the available interactive commands.

Command	Action
Abort	Abort the proof.
Rule	Select a rule to be used.
Ok	Carry on.
Unleash	Switch off interaction.
Proof	Display the proof so far.

In addition to the inference rules that the program may access, the rules **Expand H** and

Succeed are available to the user by use of the "rule" command. The first of these rules allows the user to instruct the program to expand the definition of a predicate occurring in the hypothesis despite the fact that the peek mechanism does not recommend this course of action. The second allows the user to tell the prover to assume that the current goal is actually true. This is useful if the user can see that the goal is true, but wants to avoid the prover carrying out the work necessary to prove it. Since **Fail** is an inference rule that the prover uses if no other rule applies, the user may instruct RUT to use this rule, thus avoiding the search for the proof of a goal that the user can see is false.

The commands that are available to the user of RUT are a subset of those available to the user of PROVER. Omissions from RUT include the ability to interactively add or delete hypotheses, and rewrite rules. Another omission is the ability to reorder hypotheses, but because RUT has the rule set of IMPLY this feature is not required. The "backup points" option which enables the user to command the prover to back up to a pre-specified point in the proof is not required in RUT because of the built-in ability for backtracking. When a goal cannot be proved by RUT the system attempts to apply a different inference rule to the previous goal. Thus it is as if the user had set backup points at each node in the proof and there is thus no need for commands to instruct the prover to either set, or return to, a backup point. The final option of PROVER not implemented in RUT is the "put" option, which enables the user to instantiate a variable to a specified term.

3.6. The Search Strategy

Like the UT provers, RUT searches for a proof in a depth-first manner. This strategy is simplistic and there are many ways in which it might be improved, however I have found that it is easier to follow proofs constructed in a this way since the "train of thought" of the prover is more clear, becoming closer to success for at least one subgoal at a time. A breadth-first search would lead to a goal being unpacked into subgoals and then these subgoals being forgotten while some other goal is dealt with. A heuristic search would be preferable: here the prover might opt to prove the hardest outstanding subgoal in preference to some easier one, in this way assuring that little redundant work is carried out. Of course the difficulty with this approach is specifying a workable heuristic. A further option would be to allow the user to select the next goal to be proved interactively, but this has not been implemented in RUT. Unlike IMPLY, RUT is not committed to a choice of inference rule once it has been made. If the subproofs required by a particular application of a rule cannot be found then another inference rule will be tried.

RUT is incomplete, like both IMPLY and PROVER. RUT includes the implementation of "smart backtracking" and "generalized bindings" (chapter 2, page 60) which are also present

in IMPLY, to enable it to overcome the problems of trapping and copying. The **Back Chain** rule of RUT (figure 3-1, RUT rule 9) is weaker than that of IMPLY and PROVER, in the sense that RUT can only initiate back-chaining if the consequent of an implication in the hypothesis unifies with the conclusion of the goal. In PROVER, **Ands** could be called to deduce the conclusion from the consequent of the implication (figure 2-3, Hoa rule 7), and in IMPLY **Back Chain O** (figure 2-19, IMPLY rule 13O) and **Back Chain** = (figure 2-19, IMPLY rule 14H) allow the prover to perform back chaining under more complex circumstances.

3.7. RUT in Use

> RUT has been designed with the user in mind. I follow Bledsoe and Tyson in believing that:
>
> As long as the pain in using the system exceeds the help obtained, the potential user will stay away [Bledsoe & Tyson 75a].

When given a goal to prove, the PROVER system attempts the proof for a pre-specified time. When this time is over the prover either returns success, or reports the still unproved subgoals. In RUT a different paradigm of interaction has been adopted, here the system reports to the user after the application of each inference rule. The user has a number of options which may be set to control the amount of information that is displayed about the progress of the proof.

Interaction may be switched on or off. When switched off RUT will work in stand-alone mode, unable to accept guidance from the user. Although this can lead the prover to making some decisions which will not enable it to find a proof for some conjectures, the prover in this mode is still very powerful. Another option is called speak, which determines the output that the prover makes after the application of a rule. The prover can either be "terse", printing a symbol which indicates the rule that has been applied, of "verbose" when a paragraph of text is printed.

All interaction between the user and the RUT system takes place at a high level, so that it is unnecessary for the user to be a programmer or logician before being able to perform proofs successfully using the system.

The RUT system has the ability to apply rewrite rules, thus enabling inference within a mathematical theory. Theories may be constructed using a separate program, called THEORY, which is discussed briefly in chapter 4 (page 86). These rewrite rules are loaded into the prover, and define the theory that the prover is working in. In addition to the rewrite rules of a theory, the user may also store conjectures belonging to that theory. The conjectures may be retrieved from the database and proved using a user-specified mnemonic name.

3.8. Conclusion

In this chapter I have described an investigation into the UT theorem provers of Bledsoe's group at the University of Texas. This work was carried out by building a theorem prover, called RUT, which is a rational reconstruction of the UT provers. RUT is a **rational reconstruction** rather than a **copy** of the UT provers, in that no system with the same performance as RUT has previously existed, at the University of Texas or anywhere else to my knowledge. Rather RUT incorporates the best of the ideas which are embodied in the UT provers with additional features including;

- The explicit representation of the proof being performed,
- A logic which is an extended version of that of IMPLY,
- a more friendly user interface, and,
- a backtracking interpreter which allows RUT to remake the choice of inference rule to use.

In order to reconstruct the UT provers from scratch a detailed examination of the descriptions of the programs has been carried out. This illuminated many features of the prover, and of the way that the design has changed over time. In particular it was possible to isolate the strengths of each version of the UT prover, and to build the strong parts of each into the reconstruction. Broadly, IMPLY has a more efficient logic engine than that of PROVER, but PROVER is much more adept at handling non-logical information. The logic of RUT is a further extended version of that of IMPLY, bringing RUT closer to a *GSC* system, where PROVER is much closer to *ND*, and IMPLY lies in between. The reconstruction highlighted a problem with the design of the UT provers: while a heuristic was specified which enabled PROVER to determine when the expansion of the definition of a predicate was a good idea, no similar guidance was specified for the use of lemmas, or the definitions of functions. This observation led to the development of the technique of *gazing*, which is described in the next chapter.

Chapter 4
Gazing: Using the Structure of the Theory in Theorem Proving

Look before you leap.
PROVERB

4.1. Overview

One of the outstanding problems with RUT and the UT provers is the crude means by which they select definitions and lemmas from their databases for use in a proof. The peeking and pairs heuristics help PROVER to decide which of a number of rewrite rules might be useful in the proof, but some rules cannot be selected by these heuristics, since only a certain class of lemmas can be present in the pairs database. Even if all lemmas were stored in a database there would still be a problem, since the criteria for selecting such rewritings are not very discriminating. In a huge database, there might be many rewrite rules which, for example, introduce a particular predicate, and the problem of choosing between them would become important. The consequences of performing an inappropriate rewriting step can be disastrous; leading the prover down a blind alley that requires a lot of resources to explore, or causing it to loop indefinitely.

In this chapter I describe a new technique, called *gazing*, which enables a theorem prover to plan the use of knowledge from its database. This technique arises naturally out of the work described in chapter 3, in particular by unifying and extending the techniques of peeking and pairs. Gazing gives a significant improvement over the provers described in chapter 3 since those provers often make unnecessary deductions, and (worse) fail to make necessary ones. The descriptions of gazing in [Plummer & Bundy 84, Plummer 85b] differ from the description given in this chapter. This is a result of some rationalization and extension of the technique since those papers were written.

Gazing works by considering abstractions of the rewrite rules which indicate the effect of using the rules on the predicate and function symbols which appear in the goal to be proved.

Definition 34: The *effect* of a rule is the difference in the symbols that appear in the original formula from those in the rewritten formula.

A plan is then made indicating which rewrite rules are to be used in the proof of the goal, by determining which symbols need to be introduced, and which eliminated, from the goal. In this way many irrelevant rewrite rules are eliminated from consideration.

In the next section I will describe the problem of selecting rewrite rules in more detail. I will introduce the *common currency* model, and use this to make a critical examination of the techniques of peeking and pairs. From this examination, it will be possible to see why these techniques are not as powerful as we would like, and how they may be extended. A full description of the gazing technique can be found in section 4.3. Some techniques for dealing with the failure of plans produced by gazing are presented in section 4.4. Finally, in section 4.5, an example of a proof by gazing is shown.

4.2. The Problem of Selecting Rewrite Rules

In chapter 2 I described the three classes of inference rules that the UT provers have available to them: *logical*, *database* and *special-purpose*. Recall that the UT provers record definitions and lemmas as rewrite rules which may be accessed by the database rules. Rewrite rules in the database of a prover will be *applicable* at many points in the proof of a given goal, though a rewrite rule may be *useful* in one context but not in another. For example, if we are proving a goal which involves a defined predicate then the system will have the option of expanding that predicate in terms of its definition at each step of the proof. In many proofs it will be unnecessary to perform this expansion, but in others the proof will not go through unless this step is taken. The problem that is addressed by the technique of gazing is that of when to use a rewrite rule, and which of the many available rules it is useful to use.

The heuristic that is adopted by gazing, is that *all* possible logical deduction should be carried out before the use of *any* database rule. Once the decision has been made that database rules will be useful (because the theorem hasn't been proved by logic alone) then an appropriate set of database rules will be chosen for use and applied before the further application of any logical rule available to the theorem prover. Making this complete separation between logical and database deductions makes guiding the decision to make a database deduction more intuitive.

In the UT provers a very different approach is taken. First of all, because some database inference is performed by the **Reduce** rule logical and database deduction is interleaved.

Define C, and **Pairs** both occur after all logical rules have been tried, and so conform to the heuristic in one respect, but only one rewriting step is performed by these rules, and then logical rules may immediately be used in the proof of the subgoals set up. This approach, in contrast to "logic before theory" I call "logic *between* theory".

The idea of the "logic before theory" heuristic is to work with the symbols which have been used to express the conjecture, before looking to the database rules to exchange them for other symbols. This ensures that the proof takes place at the level that it is stated if this is possible. Three reasons for adopting this approach are:

1. Performing a proof in terms of the concepts used to state the conjecture, if this is possible, will make the proof more intelligible for the human reader.

2. The search that can be carried out without using database inference is quite small compared to that when database inference is allowed. The reason for this is that most of the logical rules decrease the number of connectives that may be acted on by further logical rules, and so eventually we should "run out" of connectives to work on*. Thus if the proof requires database inference, we should be able to determine that fact very quickly, and if it does not the proof will be completed equally quickly.

3. Keeping the conjecture at a high level often means that a single inference will suffice, where many would be needed if many database rules had been used. For example, one class of database rules, definitions, are used to *abbreviate* complex formulae. These abbreviations enable conjectures to be stated at a high level, and unpacking the abbreviations has the effect of making the proof more complex.

As an example of the third reason for preferring logical over database inference: the goal (L) results from (K) when all of the definitions of defined predicates have been unpacked.

$$a \subset b \vdash^? a \subset b \qquad (K)$$

$$(x \in a \rightarrow x \in b) \wedge \neg((y \in a \rightarrow y \in b) \wedge (y \in b \rightarrow y \in a))$$

$$\vdash^?$$

$$(x \in a \rightarrow x \in b) \wedge \neg((y \in a \rightarrow y \in b) \wedge (y \in b \rightarrow y \in a)) \qquad (L)$$

---

*This explanation ignores the role of forward chaining, which can introduce new connectives into the sequent. While this can be a source of new connectives, the controls that the user may place on this rule usually prevent many connectives from being introduced.

The simple deduction that has to be made to prove (K), has become transformed into a number of deductions in (L). The proof would still go through; but it is much more complex than is necessary, and is much less intelligible as a result.

The drawback of the "logic before theory" rule is that if the conjecture is not provable by logic alone, the work that is carried out in attempting a logical proof is wasted. There is clearly a trade-off here, between the possibility of attempting an impossible proof by logic by adopting the "logic before theory" rule, and carrying out database deductions which are redundant by failing to do so. Since the amount of work that can be wasted by using database rules is much greater than that by using logic, adopting the "logic before theory" rule can be seen to be the best choice.

Not only is it important to decide carefully *when* to use a rewrite rule from the database, but once the decision is made the problem of *which* of many possible rules to use arises. For an example of the significance of this choice, suppose that the system is working in the theory given by the rules (v) and (vi).*

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x = y \Rightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

Consider the following goals:

$$a = b \vdash^{?} x \in a \rightarrow x \in b \qquad (M)$$

$$a = b \vdash^{?} a \subseteq b \qquad (N)$$

In attempting to prove (M) RUT has two options; either to use **Promote** (chapter 3, RUT rule 5, described on page 71), or to rewrite the predicate $=$ by using some rule from its database. As I have already observed RUT prefers logical deduction to expanding the definition of a predicate, so the **Promote** rule would be used. The success of the proof of this goal then depends on the particular *definition* of equality of sets that the system has.

If the definition of equality is (vi), the prover would make the definitional expansion and then fail in its proof, since it does not know what to do with the introduced $\subseteq$ atoms. An intelligent prover would recognize that rewrite rule (v) is needed for the proof of (M), but (vi) for the proof of (N). This observation seems trivial, but it is far from trivial to state a general technique for providing guidance of this type to an automatic prover. The technique of gazing attempts to provide this guidance by capturing the intuition that the rewrite rules which are required are those which will introduce hypotheses which are *similar* (in a sense to be made precise) to the desired conclusions. In the next section I describe the type of similarity that is required between hypothesis and conclusion. I show how the peeking

*This example is obviously contrived: In any "real-life" situation the prover would have at least the definition of the $\subseteq$ predicate. However the point remains: that a bad choice of rewrite rule to use can lead the prover down blind-alleys and sometimes to failure.

technique begins to capture this similarity and also that it is too weak to express all the similarities that are required.

4.2.1. Similarity: The Common Currency Model

I will now describe the *common currency model* which is due to Alan Bundy [Bundy 83b]. This is the framework within which the notion of *similarity*, which is necessary for an intelligent prover, is described.

Definition 35: A *concept* is either a predicate symbol or a function symbol.

By an abuse of terminology I will use the term *concept* to refer to the symbol which represents an idea from mathematics, and the idea itself. The context should make clear which interpretation is intended.

Definition 36: A *currency* is some representation of the concepts that appear in a formula.

We will use two different notions of currency in gazing, but each is an abstraction of a formula which represents the concepts present. The idea is that unless the hypothesis and conclusion of a goal contain the same concepts, there is little hope of proving the goal. The distinction between currency and concept is an important one. A concept is a single predicate or function symbol, while a currency may be a set of concepts, or objects with even more structure, which represents the relationships between the concepts in the formula.

Definition 37: If there is a currency which occurs in both the hypothesis and conclusion of a sequent then this is called the *common currency* of the sequent.

If there is no common currency then it is necessary to use a rewrite rule to exchange some currencies in the goal for new currencies, in order to produce a common currency.

The peeking heuristic (described in chapter 3) suggests that a useful similarity between the output side of the rewrite rule and the goal to be proved, is that they have at least a predicate in common. This corresponds to representing the currency of an atom as its predicate. This is a useful notion of currency in many cases, but is often not strong enough. However, in the examples above this notion will suffice. The currency of the conclusion of (M) is $\in$ while the currency of the hypothesis is $=$. There is no common currency but we can consider (v) as a rule whose effect is to exchange $=$ for $\in$. Using this, we can introduce the currency of $\in$ in the hypothesis, thus making it common to both sides. It is the existence of this common currency which leads us to believe that the use of rewrite rule (v) will be useful in the proof of the goal, while (vi) will probably not be.

The next subsections describe the drawbacks of the peeking technique in terms of the

common currency model. One of the weaknesses of the technique is that the notion of currency is too weak. The investigation of this fact will motivate the subsequent development of gazing.

4.2.2. Peeking: Function Definitions

The first observation about the peeking heuristic is that it does not, in general, enable the prover to guide the use of function definitions. This blind spot is due to the fact that the function symbols do not appear in the currencies between which we seek similarities. This often leads to unnecessary inferences being performed. This problem is not too serious, since this doesn't lead to the failure of the proof but rather to a more involved proof, requiring a more complex search, than is required. For an example of this consider $x \in a \cup b \vdash^? x \in a \cup b$. In RUT the goal which results from being given this goal to prove is $x \in a \vee x \in b \vdash^? x \in a \cup b$

This is due to two factors: firstly, that the rule which attempts to unify hypothesis and conclusion occurs later in the rule ordering than that which attempts to simplify formulae. More importantly, the system has no notion of what it is trying to achieve. It "simplifies" the formulae blindly, without knowing that it is attempting to introduce the same concepts in both the hypothesis and conclusion. Here for example it should "realize" that the same concepts appear in both the conclusion and hypothesis before simplification, and that attempting the match is probably a good idea.

For a more serious example of the way in which the peeking heuristic is not strong enough to deal with function definitions consider (O), an elementary theorem from number theory*.

$$Even(x) \wedge Even(y) \vdash^? Even(x+y) \qquad (O)$$

Assume that the system has the three ways of rewriting *Even* given in formulae (vii) to (ix) and additionally has knowledge of the definitions of $+$ and $\times$ and their properties of commutativity and associativity.

$$Even(x) \Rightarrow \exists y.(y+y=x) \qquad (vii)$$

$$Even(x) \Rightarrow \exists y.(2 \times y = x) \qquad (viii)$$

$$Even(0) \Rightarrow \top \qquad (ix)$$
$$Even(s(s(x))) \Rightarrow Even(x)$$
$$Even(s(0)) \Rightarrow \bot$$

The peeking heuristic recommends that the prover attempt to reexpress the hypothesis in

---

*This example was originally suggested by Alan Bundy in [Bundy 83b].

terms of the predicate which appears in the conclusion. The fact that it is already in this form should indicate that consideration of the functions would be useful. A mathematician would have no difficulty in noticing that reexpressing *Even* in terms of (vii) would lead to a trivial proof. The conjecture is after all "about" + and thus the characterization of *Even* which is also expressed in terms of this concept is likely to be useful.

4.2.3. Peeking: One-Step Look-Ahead

Another major drawback of peeking is that the heuristic is short-sighted since it can look ahead only one level of definition. If the predicate that is to be expanded, P, does not immediately contain the predicate that we desire to introduce, Q, but instead contains a predicate which, when *its* definition is expanded, introduces Q, then the definitional expansion of P will not be recommended. For example, consider again (M) above, this time imagining that we don't have rule (v), but that the prover has been given the rewrite rule about $\subseteq$ below (i) (repeated from page 23).

$$x \subseteq y \Rightarrow \forall z.\, z \in x \rightarrow z \in y \qquad (i)$$

In this example there is no single rewrite rule which will enable the introduction of $\in$, so peeking does not give the right guidance and a necessary inference is not performed. To discover the sequence of rewrite rules which would have the desired effect requires a more detailed investigation into the effect of rewrite rules on a goal.

4.2.4. Pairs and the Common Currency Model

The **Pairs** rule from the UT prover (Hoa rule 3, described in chapter 2, page 51) can also be viewed in the light of the common currency model. Recall (from chapter 2, subsection 2.3.2) that the pairs technique is designed to retrieve a lemma from a database if the predicates appearing in the hypothesis and conclusion of the goal are the same, but the goal is not provable. In the common currency model this translates to saying that the goal already has a common currency - namely the matching predicates - but that the goal could not be proved within this currency. The pairs property lists record a means of moving to new currencies within which the proof might be possible. The pair property lists are the encoding of considerable mathematical experience which leads to the suggestion of these new currencies. Unfortunately, these are precoded by the implementors of the theorem prover which means that if none of the pair property lists leads to a proof, or if none are specified, then the prover has no alternative but to give up. If the prover were itself able to deduce new currencies which it could move to, then it would not have to rely on correct and complete specifications provided by the implementor.

4.2.5. Summary of the Common Currency Model

In this section I have described the common currency model. This model captures the idea that in order to prove a goal within a theory it is necessary that there be some similarity between the hypothesis and the conclusion of the goal. If a planner is to use the database of rewrite rules intelligently it must take into account that its goal is to introduce the same concepts throughout the goal to be proved. This is only possible by giving the system some representation of the concepts that are present, and the notion that it is to introduce these common currencies.

I have described the peeking and pairs heuristics in the light of this model and I have shown how these techniques begin to represent this aim, but that the peeking heuristic has 3 main drawbacks:

1. the representation of a currency that is used is too abstract,
2. some rewrite rules cannot be controlled using these techniques, and,
3. the notion of how to introduce common currencies is too limited.

In the next section I will describe the technique of gazing which overcomes these problems. Gazing makes a very rough plan which attempts to introduce the same predicate in the hypothesis and conclusion. To make this plan the system has the same representation of currency as peeking, but is able to search for chains of rewrite rules which achieve this goal, rather than being limited to a single step plan. This rough plan is then refined using a more discriminating notion of currency which includes a representation of the functions which appear in the goal. If the plan is successfully refined then it is executed to produce a new goal which should be provable without further use of rewrite rules.

4.3. Gazing

In this section I describe the gazing technique. The idea behind gazing is to produce a plan indicating which rewrite rules may be useful in the proof of a goal. The program will make a plan to use rewrite rules so that execution of the plan will introduce a currency which is common to the hypothesis and conclusion. This plan is created by examining the currencies of the goal to be proved, and those of the rewrite rules of the theory. The currencies are in fact *abstractions* of the formulae of the problem space which represent the concepts that appear in the formulae.

The gazing technique unifies the ideas of peeking and pairs in the sense that predicate definitions (handled by peeking in Bledsoe's system), function definitions (previously

unguided) and lemmas (partially handled by the pairs technique) are handled uniformly by the gazing technique. The main extensions to the idea of peeking are:

- Gazing can be used to guide the use of all rewrite rules. The use of a particular rule does not depend on its status as a definition or lemma, but only on its potential relevance to the proof.
- The look-ahead of peeking is extended to be arbitrarily deep, and,
- Gazing uses a more discriminating notion of currency.

The idea of using abstract representions of the problem, and using the solution to these abstract problems to guide the search for a proof of the problem in detail, is not new. It originated with Minsky [Minsky 63], and, in the theorem proving context, has been investigated by Plaisted [Plaisted 80, Plaisted 86]. For a discussion of the relationship between gazing and this work see chapter 6, part 6.3.2.1. The use of abstraction spaces is a very powerful technique. The key idea is that a problem can be reduced in complexity by ignoring some aspect of the problem. A solution to the less complex problem can then be found. Once we have this solution, it may be necessary to refine it in some way because of the extra detail of the complete problem, but it is easier to do this than to find the solution "from scratch".

The gazing system will make its plans in a *hierarchy* of abstraction spaces. In a hierarchical system many levels of abstraction are identified. The system begins planning at the coarsest level and gradually fills in the details at each successive level until all of the details are completely satisfied. Failure to complete a plan at any level of abstraction causes the system to replan at the previous (less detailed) level. The whole plan is complete when the planning in the least abstract space is completed.

In gazing three different aims are isolated, and these define the abstraction spaces that the prover uses to produce the plan.

- In the *Predicate Space* the currencies of the goal are the predicates in that goal and the aim of the planner is to introduce a pair of atoms with the same predicates.
- In the *Function/Polarity Space* the currencies of the goal represent the functions, predicates and polarity of the formula. Here the aim of the planner is to introduce the same function symbols into complementary atoms with the same predicate.
- Finally in the *Full Space*, in which all the detail of problem is present, the goal is to unify two formulae.

These aims are the goals of the three abstraction spaces within which gazing carries out its

planning. These aims are increasingly detailed, although the achievement of a more general aim is neither necessary nor sufficient for the achievement of the more specific aim. In particular it is not necessary for two formulae to contain the same function symbols before they can be unified, since a term containing function symbols may be unified with a variable.

It is important to note that a plan produced in abstraction spaces *cannot* be guaranteed to produce the desired effect in the full space unless the detail that is ignored in the abstraction space cannot cause the plan to fail. If the details were irrelevant in this way then the use of abstraction spaces would be pointless, since solving the abstract problem would be as difficult as solving the original problem. In the case of gazing, the detail that is ignored may cause the plan to fail. For example, in no abstraction space are the connectives of the formulae considered, but the connectives of the formulae can cause unification in the full space to fail. Thus the plans that are produced by the use of gazing cannot be guaranteed to produce the proof. On the other side of the coin of course, there is the knowledge that constructing plans while ignoring detail is simpler than the original problem. Thus, the plan may be produced more cheaply and more quickly in the abstraction space. This is the reason for preferring to consider abstractions.

In each space we have abstractions which record the effect of the use of the rewrite rule on the concepts in the formula being rewritten. The currencies of a formula are the results of mapping that formula under the abstraction mapping. Thus we have to design this mapping to remember only the concepts that appear in the formula. Because there are two abstraction mappings, the currencies of the formula are different in the two spaces.

The system will first make a plan using the abstractions with the least detail. That is, it will plan to introduce atoms whose predicates are the same into the hypothesis and conclusion. This plan will be only a sketch since much of the structure of the problem has been ignored. However the amount of search that is required to make this sketch plan is very small and can be carried out very quickly. The planner then considers the next level of detail. Here the system has to consider the functions and polarities of the atoms. The original plan may have to be amended in order to be applicable, and in the worst case it must be rejected and a new plan created in the [illegible] space. Assuming that the plan can be suitably amended it is passed back to the problem space and executed to produce a new goal. If the new goal is provable then it should be provable without recourse to further rewriting.

The actual strategy that is used to make the plan may be varied depending on various features of the theory. In the remaining part of this chapter I describe gazing with a particular strategy in view. This strategy, which I call *SS1*, has been devised by considering set theory. Some of the heuristics that have been adopted in *SS1* may not be general. In

chapter 6, I describe extensions that might be made to the implementation which would allow a user to specify the search strategy that is to be used to make the plan, or allow the program to choose an appropriate search strategy dependent on the rewrite rules of the theory.

Since the plan is constructed by considering abstractions there is the possibility that the execution of the plan in the full space will fail. In section 4.4 I describe 3 ways in which the plan can fail and show that techniques for recovering from failure are available within the gazing technique.

The technique of gazing requires that the theory within which the prover is working has a particular structure. This structure is very general and is described in the next section. Then the formation of plans in each of the abstraction spaces is described. Plans are made by finding chains of abstracted rewrite rules which achieve the aims of the gazing system in that abstraction space.

4.3.1. Constructing a Theory

Gazing requires that the theory that is being developed is presented serially, rather like the presentation of a theory in a textbook. Such presentations are usually rationally reconstructed versions of mathematical activity, but the insistence on this mode of presentation in gazing ensures that theories are well-structured.

As observed above definitions serve only as abbreviations of complex formulae, and we must ensure two things about them. First, that they do not increase the number of things that can be proved within the theory, and secondly, that they can always be eliminated in favour of the primitive notions of the theory. In chapter 8 of [Suppes 57] these principles are stated formally as follows:

- A formula α introducing a new symbol of a theory satisfies the *criterion of eliminability* iff: whenever β is a formula in which the new symbol occurs, then there is a formula γ in which the new symbol does not occur such that $\alpha \rightarrow (\beta \leftrightarrow \gamma)$ is derivable from the axioms and preceding definitions of the theory.

- A formula α introducing a new symbol of a theory satisfies the *criterion of non-creativity* iff: there is no formula β in which the new symbol does not occur such that $\alpha \rightarrow \beta$ is derivable from the axioms and preceding definitions of the theory but β is not so derivable.

A database of rewrite rules for use by a program implementing gazing is built by a program called THEORY. In this program the conditions expressed above are enforced by requiring the user to construct the theory in the following way:

1. **The user must state initially, which of the predicates and function symbols in the theory are to be undefined.**

2. **The definitional formula of a predicate or function may contain only previously defined predicates and functions, or those which have been declared undefined.**

THEORY can detect when a particular rewrite rule is a *lemma* by the fact that it contains no undefined concepts.

These criteria do not allow THEORY to accept recursive definitions. I have carried out no research on the question of whether gazing may be extended to deal with concepts which are defined in such a way, but it is my belief that such an extension is possible.

These criteria alone are not sufficient to ensure the satisfaction of the properties above, but some additional criteria, described in detail in [Suppes 57] are also used. These will not concern us here. I will call 2 the ordering rule. Rule 1 is only necessary in order to allow the ordering rule to be checked at each stage. The ordering rule has two effects; first it ensures, along with other some other conditions, that the definitions that are stated obey the eliminability criterion. Also it ensures that the definitions of the theory induce a partial order on the concepts of the theory.

Definition 38: The *input concepts* of a rewrite rule $\alpha \Rightarrow \beta$ are those concepts which appear in α.

Definition 39: The *output concepts* of a rewrite rule $\alpha \Rightarrow \beta$ are those concepts which appear in β

Definition 40: The *definitional order*, $<_T$, induced by the definitions of a theory T, is defined as follows:

$C_1 <_T C_2$ if C_1 is an output concept of the definition of C_2

Definition 41: If a concept C has no other concept D such that $D <_T C$, then C is *undefined*.

The definitional order $<_T$ is a partial order. This is ensured by the fact that we insist that any concept is defined only in terms of previously defined terms.

The partial ordering $<_T$ may be extended to a total ordering by arbitrarily choosing an order between concepts not ordered by $<_T$

Definition 42: The *criticality ordering* of T is a total ordering $\ll_T$ on the concepts of the theory T. The criticality ordering may be defined arbitrarily provided the following condition holds:

$C_1 <_T C_2 \rightarrow C_1 \ll_T C_2$

Definition 43: The *criticality* of a concept, C, is the number of concepts which occur before C in the criticality ordering

The intention of criticality is to capture the fact that some concepts are more complex, or of **higher level**, than others. This notion is used to orient equivalences in both spaces, and in the predicate space to guide the search.

Since the criticality ordering $\ll_T$ is a total order, we can define *max-crit* as follows.

Definition 44: If S is a set of concepts from the theory T, then

$$max\text{-}crit(S) = C \leftrightarrow \forall D \in S (D \neq C \rightarrow D \ll C)$$

Formulae which involve high-level concepts are themselves high-level statements and so the notion of criticality should be extended to formulae. To do this we have to consider briefly the currencies used in the function/polarity space.

Definition 45: An *f/p triple* is the abstraction of an atomic formula in the function/polarity space. F/p triples are written $\langle$ *Major,Minor,Pol* $\rangle$, where,

- *Major* is the predicate of the atom,
- *Minor* is the set of function symbols that occur in the arguments of the atom*, and,
- *Polarity* is the polarity (definition 16 on page 18) with which the atom occurs in the rewrite rule. This may be unspecified if the rule derives from an equivalence.

Definition 46: The *f/p-abstraction of a formula*, F, is the set of f/p triples which are abstractions of the atomic subformulae of F.

Definition 47: F/p triples are ordered by the ordering $\prec_T$ which is defined by:

$$< P_1, F_1, Pol_1 > \prec_T < P_2, F_2, Pol_2 > \leftrightarrow]$$
$$max\text{-}crit(P_1 \cup F_1) \ll_T max\text{-}crit(P_2 \cup F_2)$$

This order extends to sets of f/p triples, where a set of triples is greater than another if the maximum of the triples in the first set is greater than the maximum of the triples in the second.

Definition 48: If the currency set of a formula is larger than the currency set of another formula, then the first formula is said to be of *higher criticality*.

The notion of criticality gives the gazing system the power to orient automatically the rewrite rules in its database. We need to ensure that the rewrite rules always rewrite formulae to *simpler* formulae, i.e. formulae with lower criticality. If we can do this then the possibility of infinite rewriting will not occur, since the order $<_T$ is bounded by the undefined concepts of the theory. Any sequence of rewriting steps has to terminate with a rewriting to the simplest concepts of the theory. The use of predicate definitions in PROVER

---

*Functions of arity 0 (constants) are not included in this set.

ensured that this was true for such rewrite rules, but in the context of gazing we seek to control the use of all rewrite rules. We have therefore, to orient all equivalences so that the criticality measure decreases when they are used.

Before considering the orientation of equivalences in general, I consider a special class of equivalences, the *null equivalences*.

> **Definition 49:** A *null equivalence* at a particular level of abstraction is one which has identical left- and right-hand sides at that level of abstraction.

Notice that an equivalence may be null at one level of abstraction, but not at another. This is because, at one level of abstraction the differences between the input and output sides of the rule are not represented, where at less abstract level, those differences are retained. For example (iv) is a null equivalence at the predicate level, but not at the f/p level.

$$x \cap (y \cap z) == (x \cap y) \cap z \qquad (iv)$$

If an equivalence is null at some level of abstraction then it is not included in the database for this level. This is because the equivalence could not possibly be of any use at this level of abstraction. The rewrite rules that derive from equivalences are used to exchange concepts for other concepts of the theory, and these equivalences do not have this effect.

The orientation of a non-null equivalence $\alpha \leftrightarrow \beta$ is made so that the criticality of the output formula is less than that of the input formula. Since criticality of formulae is a total order, this definition will ensure a unique orientation for all non-null equivalences.

It is a corollary of this rule that all definitions will be oriented to eliminate the defined term.

The assumption underlying the orientation rule is that the rewrite rules will be used for forward-chaining and so the application of a rule oriented by these rules will cause a decrease in the criticality of the formula. If a rewrite rule is stated as an implication then there is no choice as to its orientation, but we do have a choice as to whether to use the rule to back-chain or forward-chain. We use the rule to back-chain if the consequent of the implication has higher criticality than the antecedent, otherwise we use the rule to forward-chain.

4.3.2. The Predicate Space

In the predicate abstraction space the only things that the system is concerned with are the predicates in the goal. Thus the rewrite rules are represented as two sets of predicates: the input and output predicates of the rule.

For example, (v) and (vi) are abstracted to (x) and (xi) respectively.

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x = y \Rightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

$$\{=\} \Rightarrow \{\in\} \qquad (x)$$

$$\{=\} \Rightarrow \{\subseteq\} \qquad (xi)$$

The goal that is to be proved is similarly abstracted to be two sets of predicates, the set of goal predicates, and the set of hypothesis predicates. For example, (P) is abstracted to (Q).

$$x \subseteq y \wedge y \subseteq z \vdash^{?} x \subseteq z \qquad (P)$$

$$\{\subseteq\} \vdash^{?} \{\subseteq\} \qquad (Q)$$

The result of deduction in this space is a plan which, when executed, will rewrite a goal to a new goal in which the same predicate appears in both the hypothesis and conclusion.

$<_T$ can be represented in a structure that I call the definitional gaze graph.

> **Definition 50:** The *definitional gaze graph* of a theory T is a directed graph where each node in the graph is labelled with a predicate from T. Arcs exist in the graph from a node labelled by a predicate P to all of the nodes labelled by the output predicates of the definition of P.

For example, consider the theory made up of the definitions (v), (i), and (xii). The definitional gaze graph of this theory is shown in figure 4-1.

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x \subseteq y \Rightarrow \forall z.\, z \in x \rightarrow z \in y \qquad (i)$$

$$x \subset y \Leftrightarrow x \subseteq y \wedge \neg(x = y) \qquad (xii)$$

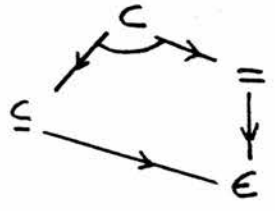

Figure 4-1: An Example Definitional Gaze Graph

The definitional gaze graph is extended to a full gaze graph by including arcs deriving from lemmas of the theory to the structure.

> **Definition 51:** The *full gaze graph* for theory T is a directed graph where each node in the graph is labelled with a predicate from T. Arcs exist in the graph from a group of predicates P_i to a group of predicates Q_j iff there is a rewrite rule in T, which has input predicates P_i and output predicates Q_j.

Notice that, because rewrite rules derived from lemmas may have more than one predicate in the input side, the structure of the full gaze graph of the theory may become very complex. For example the rewrite rule (xiii) has arcs from the nodes labelled by $\in$ and $\subset$, to the nodes labelled $\in$ and $=$.

$$x \in y \wedge y \subset z \Rightarrow x \in z \wedge y \neq z \qquad (xiii)$$

Facts introduced late in the development of the theory are very high-level, but they often mention low-level concepts. We wish to indicate that such facts are "about" the high-level concepts, and is not to be used as a means to rewrite the low-level concepts. For example (xiv) is primarily concerned with the concepts of *Open-Cover* and *Compactness*. The formula mentions the concepts $\in$ and *Finite* in addition to these, but we understand these to play a secondary role in the formula.

$$\begin{aligned} &\mathit{Compact}(s,t) \Leftrightarrow \\ &\quad \forall o.\ \mathit{Open\text{-}Cover}(o,s,t) \rightarrow \\ &\quad\quad \exists z.\ (\mathit{Finite}(z) \wedge z \subseteq o \wedge \mathit{Open\text{-}Cover}(z,s,t)) \end{aligned} \qquad (xiv)$$

This intuition is captured by distinguishing the predicate in the input side with highest criticality.

> **Definition 52:** The *primary predicate* of a formula is the predicate with the highest criticality in the formula. The remaining predicates are the *secondary predicates* of the formula.

The problem of finding a plan in the predicate abstraction space can be thought of in two different ways. Here we consider the problem as the search for paths through the gaze graph. The alternative view is as a search for a proof in a simple logical system, a discussion of which can be found on page 95. The key idea however is the same, we wish to rewrite two predicates to a common currency.

It should be clear that planning in the predicate abstraction space and peeking have a lot in common. In fact, the search in the predicate abstraction space is a generalization of peeking. In peeking only the single application of one predicate definition to the hypothesis was considered. In gazing, rewriting using a sequence of rules is considered, and both the conclusion or the hypothesis may be rewritten.

4.3.2.1. Planning: Paths Through the Gaze Graph

First of all recall that the goal in this space is to prove an abstracted sequent, which consists of two sets of predicates, the goal set, and the hypothesis set. Since we have abstracted away the connectives of the formulae that are involved in the proof, we have to make a decision about what we will consider these sets to represent. We can deduce some information from the inference rules of the system. Since gazing is being attempted after the application of all other rules have failed, none of the conditions of the other rules are met. For RUT this means that the conclusion of the sequent is a disjunction of atomic formulae, and that the hypothesis is a conjunction in which each conjunct is either an atomic formula or an implication. This can easily be checked by considering figures 3-1 and 3-2 on pages 71 and 72.

In *SS1*, the simplifying assumption that the conclusion set is a set of *disjuncts* and the hypothesis a set of *conjuncts* is made. That is, the possibility of implications in the hypotheses will be ignored. This means that we can choose any predicate from the conclusion set to prove, and we can use all of the predicates in the hypothesis set. Each rewrite rule is thought of as requiring all of the predicates in the input set to be present in the set to be rewritten. The assumption is that the predicates are conjoined in the full space. If the rule may be applied, all of the predicates which appear in the output set of the rule are added to the rewritten set. These assumptions may lead to a plan that does not apply, since in the final analysis these assumptions may be false. As observed above, this is an inherent limitation of the abstraction space technique. The trade-off is that plans may be made very quickly in the abstraction space.

The first notion that is needed to describe the task of introducing a common currency is that of a P–Q path in a gaze graph.

> **Definition 53:** A *P-Q path in G* is a sequence of arcs $A_1, \ldots, A_n$ from gaze graph G which obey the following rules:
>
> 1. The primary predicate of the input of A_1 is P,
> 2. The primary predicate of the output of A_n is Q, and,
> 3. The primary predicate of the output of each A_i is the primary predicate of the input of A_{i+1}, for $1 \leq i < n$,

The intuition behind this definition is that a predicate may be rewritten using some arc in the graph, the result will be a set of predicates which can themselves be rewritten. To restrict the search, the primary predicate of the output of the rule is the next to be rewritten.

The definition of P–Q path ignores the secondary inputs of the rules. Clearly this would

lead to the formation of plans which could not be executed, so the definition needs to be refined. The idea of a P-Π-Q path is that at each step the secondary inputs must be present before the path may be followed. For the first step, the secondary inputs must already be present in the set to be rewritten. For subsequent steps however, the secondary inputs may derive from the original set, or have been introduced by the previous application of a rule.

Definition 54:

A *P-Π-Q path in G* is a *P–Q* path in G, such that,

1. $P \in \Pi$, and,

2. For each A_i in the path, the secondary inputs of A_i occur in $\Pi \cup O_1, \cup \ldots, \cup O_{i-1}$, where O_j is the output of A_j.

The definition of a P-Π-Q path enforces the following heuristics about the use of rewrite rules.

1. Suppose the gaze graph includes a lemma L, with primary predicate I, and a set of secondary predicates S, and that we are seeking a path from I to some other predicate. We can consider using L iff every member of S already appears in the abstracted goal.

2. Suppose we are searching for a path from predicate A to some other predicate, and that the gaze graph contains a lemma L, with some input predicate and a set of secondary predicates which has A as a member. L is not a valid member of the path.

Heuristic 1 insists that the secondary inputs are present in the goal. This is a way of limiting the search for a plan. If we allowed the prover to construct subplans to introduce these secondary inputs, the planning could quickly get out of control. This also specifies that the rewrite rule that is being used "fits" in the sense that it is immediately applicable. For example, suppose that *Open-Cover* is the predicate with highest criticality in the input side of (xv). This rule could be used to rewrite a formula involving *Open-Cover* only if the other predicates $\{Finite, \in, \subseteq, Open\text{-}Cover\}$ were immediately present in the formula.

Heuristic 2 enforces the notion that a rewrite rule is "about" the predicate with highest criticality. The idea is to forbid the use of a rewrite rule as a means for rewriting a predicate P, if there is a predicate with higher criticality in the input of the rule. This prevents, for example, the use of (xv) to rewrite an atom involving $\in$, even when all of the other predicates are present in the formula to be rewritten. It is my contention that this heuristic has great intuitive appeal. In addition to limiting the search space for gazing, it allows the prover to consider only those rules in which the concept being rewritten plays an important role in the input of the rewrite rule.

$$\forall o.(Open\text{-}Cover(o,s,t) \rightarrow \exists z.\,(Finite(z) \wedge z \subseteq o \wedge Open\text{-}Cover(z,s,t))) \Rightarrow Compact(s,t) \qquad (xv)$$

Both of these heuristics render the gazing technique incomplete, but restrict the search that may be carried out in the attempt to construct the proof. This is in much the same spirit as the arguments about completeness of general theorem provers (see page 17). They are adopted in order that the search for the proof of non-theorems terminates quickly, rather than wasting large amounts of resources. This is at the expense of the inability to prove some theorems.

An alternative to heuristic 1, would be to allow the planner to construct plans which will introduce any missing subsidiary inputs of the rule. Such an approach would allow the prover to consider applying many rules which are inappropriate, in the sense that the subsidiary inputs may never be introduced, and expend possibly a great deal of resources in trying to make the rule "fit". Another possible approach would be to have the prover attempt rules which fit most closely first, and then revert to less obviously appropriate rules later. Again, however, the amount of resources used in the attempted proof of non-theorems may be very large. I prefer that the prover is incomplete, and that when it fails to prove a conjecture it does so without a large investment of resources in the attempt.

Construction of the plan to introduce a common currency into a goal $H \vdash^{?} C$, is exactly the task of selecting $P_h \in H$ and $P_c \in C$ and then finding a P_h–H–Q path in G, and a P_c–C–Q path in G for some Q. In *SS1* we choose P_c and P_h to be the primary predicates of the conclusion and hypothesis respectively. The justification for this is that the goal is "about" the concept with the highest criticalities, and that these should be taken as the starting point of the proof.

The common predicate Q could be one of P_c or P_h, in which case one of the paths will be empty. This corresponds to rewriting one of the predicates in terms of the other, and is quite desirable. Best of all is that a common predicate already exists between the hypothesis and goal. In this case the paths are both empty. This situation would arise, for example, when considering (O), where *even* appears in both the hypothesis and conclusion sets of predicates.

$$Even(x) \wedge Even(y) \vdash^{?} Even(x+y) \qquad (O)$$

To return to our examples of earlier this chapter consider the gaze graph 4-1. When asked to prove (N) the system will find that there is one path from the node labelled by $=$ to the node labelled $\subseteq$, and that this path represents an application of the rewrite rule (vi).

$$a = b \vdash^{!} x \in a \rightarrow x \in b \qquad (M)$$

$$a = b \vdash^{!} a \subseteq b \qquad (N)$$

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x = y \Rightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

$$x \subseteq y \Rightarrow \forall z.\, z \in x \rightarrow z \in y \qquad (i)$$

Similarly when asked to prove (M) the system will search for a path from the node labelled by $=$ to the node labelled by $\in$ and will find two paths, of which the shortest represents the application of rewrite rule (v). The other path - representing an application of (vi) followed by an application of (i) - will not be selected unless the first is rejected at a later stage.

4.3.2.2. Planning: Proofs in Propositional Logic

The problem of finding a plan in the predicate abstraction space can also be construed as finding a proof in propositional logic.

> **Definition 55:** *Propositional logic* is the subset of the first order predicate calculus which admits only predicates of arity 0.
>
> **Definition 56:** An atomic formula in propositional logic is called a *proposition*.

The propositions of the abstract system are the predicates of the full space, and the hypotheses of the proof are the set of hypothesis predicates, and the set of abstractions of rewrite rules. The goal of the proof is the predicate of the conclusion atom. For example, if the conclusion predicate is $\in$, the hypothesis predicate is $=$, and the theory is (v), (vi) and (i). Then the axioms of the proof system are:

$=$
$\neg = \vee \in$
$\neg = \vee \subseteq$
$\neg \subseteq \vee \in$

and the desired conclusion is $\in$.

Unlike the predicate calculus, propositional logic is decidable. That is, there are decision procedures which will determine whether any statement of propositional logic is a theorem or non-theorem. We can use such a procedure to prove any theorem of this system. Resolution, described in chapter 1, is a decision procedure for propositional logic, and is particularly appropriate, since the axioms are in a normal form. Remember that the input and output side of the rewrite rules are both considered to be conjunctive in this space. Two resolution proofs of $\in$ from the axioms above would be as in figures 4-2 and 4-3.

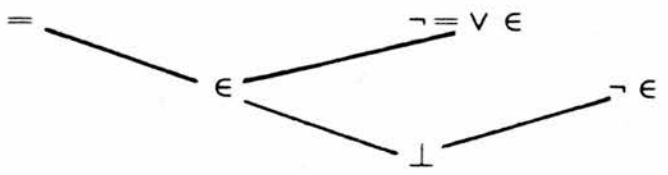

Figure 4-2: First Resolution Proof of Path from = to ∈

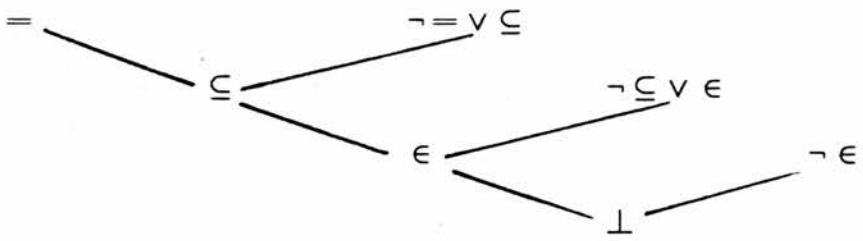

Figure 4-3: Second Resolution Proof of Path from = to ∈

Extracting the plan from these resolution proofs is not difficult. The steps that are to be applied are represented by the implications which are resolved against. The order in which these rules are to be applied in the plan is exactly the order in which the resolutions take place. To ensure that a unique ordering may be extracted from the plan, the linear input strategy (described in chapter 1, on page 26) may be used. Should the result of a resolution contain more than one literal, a particular literal from that set should be chosen. This should be the literal representing the primary predicate of that set.

4.3.3. Planning in the Function/Polarity Space

When a plan has been constructed in the predicate abstraction space, it is passed to the function/polarity space. The planner has to check that the plan can be applied when considering the extra detail of this space. Each planned step is examined in turn in the context of the abstraction of the current goal. There are two things that are considered here which were not in the predicate space:

- Polarity, and,
- Function Symbols.

The function symbols present in the goal and rewrite rules must be considered when planning the rewritings to be carried out, since they may only be exchanged by means of rewrite rules. We saw in section 4.2.2 (page 81), that one of the drawbacks of the peeking technique is that it does not consider functions in planning rewriting.

The polarity part of the f/p triple is used to protect the system from performing rewriting steps which rewrite a provable conjecture into one which cannot be proved. Recall the

definition of polarity from chapter 1, definition 16. The polarity of a formula records the role that the formula plays in the conjecture. A subformula with polarity + is a conclusion of the goal, that is, the formula must be shown true before using it. A formula with negative polarity, on the other hand, may be assumed for use in the proof and so is considered to be a hypothesis. Consider (R) and the rewrite rule (xvi).

$$a = b \;\vdash^{?}\; a \subseteq b \tag{R}$$

$$x = y \Rightarrow x \subseteq y \tag{xvi}$$

The goal (R) is provable from the definitions of set theory, but if (xvi) were used to rewrite the hypothesis the resulting goal would not be provable. This is because (xvi) derives from an implication, and using it to rewrite a hypothesis causes the hypothesis to be weakened. On the other hand the use of (xvi) to rewrite the conclusion of (S) does not have this effect.

$$a \subseteq b \;\vdash^{?}\; a = b \tag{S}$$

In general we can weaken the conclusion of a goal as we please since it is impossible to transform a provable goal into one which is not provable by this type of step. We can capture this intuition by considering the polarity of the formulae in the conjecture. As remarked above hypotheses are negative and conclusions are positive. The polarities of (R) and (xvi) are given as superscripts in (T) and (xvii) respectively.

$$a = {}^{-}b \;\vdash^{?}\; b \subseteq {}^{+}a \tag{T}$$

$$x = {}^{+}y \Rightarrow x \subseteq {}^{-}y \tag{xvii}$$

Notice that the rewrite rule has been treated as a hypothesis in assigning polarities, as the database plays the role of storing implicit hypothesis. The irreversibility of weakening hypotheses or strengthening conclusions can be expressed by disallowing the use of a rewrite rule on a formulae when this would involve matching formulae of opposite polarities.

The definition of polarity does not allow the assignment of a unique polarity to the subformulae of equivalences. This is because the equivalence may be considered as an abbreviation for two implications, with each subformulae appearing with opposite polarity in each implication. Rewrite rules derived from equivalences may be used to rewrite formulae of either polarity since the rewriting step may use either of the implications that the equivalence represents, and one of these implications may match. This is equivalent to selecting one of the implications that the equivalence abbreviates. Thus the problem of polarity did not arise for the peeking technique, since it was used only to control the use of predicate definitions, which are necessarily equivalences.

An additional effect of considering polarity at this level of abstraction is that the number of

possible causes of connective structure failure (see section 4.4.2.1 on page 107) are lessened. When connective structure failure occurs, a connective in the rewrite rule does not match the connective in the formula to be rewritten. This occurs, for example, when attempting to match $A \wedge B$ against $A \rightarrow B$. This difference in connective can be detected by considering the polarity, but the clash between $A \vee B$ and $A \wedge B$ cannot, since A and B have the same polarity in $A \vee B$, but opposite polarity in $A \rightarrow B$.

Both the polarity and the function symbols can cause a planned step to be reconsidered or amended. Formulae are represented at this level of abstraction by f/p triples as already defined. Rewrite rules are represented by *triple exchanges*.

Definition 57: A *triple exchange* is the abstraction of a rewrite rule.

Definition 58: The f/p abstraction of the input side of a rewrite rule is called the *input set* of the rule

Definition 59: The f/p abstraction of the output side of a rewrite rule is called the *output set* of the rule

Examples: The rewrite rules (xviii) and (xx) have the triple exchanges (xix) and (xxi) respectively.

$$x \in 2^y \Leftrightarrow x \subseteq y \qquad (xviii)$$

$$\{ \langle \in, \{2\}, Pol \rangle \} \Leftrightarrow \{ \langle \subseteq, \{\}, -Pol \rangle \} \qquad (xix)$$

$$x \in y - z \Leftrightarrow x \in y \wedge \neg x \in z \qquad (xx)$$

$$\{ \langle \in, \{-\}, Pol \rangle \} \Leftrightarrow \{ \langle \in, \{\}, Pol \rangle, \langle \in, \{\}, -Pol \rangle \} \qquad (xxi)$$

Suppose that the plan made in the predicate space requires that the predicate P be rewritten, using some rewrite rule, to predicate Q. In this abstraction space, we will have one or more f/p triples $\langle P, F, Pol \rangle$. This is the abstraction of the atom which was abstracted to P in the predicate space. The abstraction of the rewrite rule will be of the form $In \Rightarrow Out$ where In is a set of f/p triples, of which $\langle P, F', Pol1 \rangle$ is a member and Out is a set of f/p triples of which $\langle Q, G, Pol2 \rangle$ is a member.

It is necessary to record the predicate of the atom in the corresponding triple despite the plan created in the predicate space. This is because some triple exchanges are effective only within the context of a particular major part, for example, if we have a rewrite rule which rewrites $x \in a \cap b$, then it is not enough to have the function symbol $\cap$ present, but it must appear in the second argument of an atom with predicate $\in$. The triples do not record which argument position the function occur in, but recording the predicate may eliminate triple exchanges from consideration when attempting to perform the rewriting step.

Planning for Polarities

In the f/p space the polarity part of a triple will be one of $\{+, -, Pol, -Pol\}$. The polarity *Pol* is used to indicate an unspecified polarity when the exchange arises from an equivalence. All equivalences will contain the polarity *Pol* which denotes that the polarity of the formula is not defined. All occurrences in different equivalences are independent. That is, if a decision is made about the use of a particular equivalence that requires *Pol* to be, say, + for that equivalence, this does not effect the value of *Pol* in any other equivalence. $-Pol$ merely denotes the opposite polarity to *Pol*. Consider the rewrite rule (xxii) and its associated triple exchange (xxiii).

$$x \subset y \Leftrightarrow x \subseteq y \wedge \neg(x = y) \wedge \neg(x = \emptyset) \qquad (xxii)$$

$$\{ \langle \subset, \{\}, Pol \rangle \} \Leftrightarrow \{ \langle \subseteq, \{\}, -Pol \rangle, \langle =, \{\}, Pol \rangle \} \qquad (xxiii)$$

When (xxiii) is applied to $\langle \subset, \{\}, + \rangle$ the result is $\{ \langle \subseteq, \{\}, + \rangle, \langle =, \{\}, - \rangle \}$, Notice that *Pol* has been given the value -, from the match because for (xxiii) to apply the corresponding triples must have opposite polarities. This constraint has to be propagated through the resulting currencies, and so *Pol* in the output side of the f/p exchange has been replaced by $-$.

Assigning an unspecified polarity to a value dependent on the polarity of the triple it is matched against is equivalent to choosing to use one of the implications that the equivalence represents.

The constraint that the polarities of matched formulae be complementary cannot be considered in the predicate space. This can lead to plans which require the application of a rewrite rule which when the polarity is considered cannot be applied. Only in the f/p space can such a plan be rejected or amended.

The first step in checking a step of the plan in the f/p space is to consider the polarities of the members of the triple exchange and conjecture. If each member of the triple exchange has a partner in the conjecture which has the same predicate, and opposite polarity then the step is accepted on the grounds of polarity. If there are variable polarities in the conjecture or triple exchange, then these may be assigned values where necessary, or left variable if possible. This latter case is preferable since this solution is less constraining.

The step may be rejected if there is some member of the triple exchange which has no corresponding triple in the goal. Since the reason for the step is known (the step must have had the effect of exchanging some predicate for another), another step is found which will have the same effect if possible. *SS1* fails to find a plan if no replacement step may be

found, but this is not necessary - a more persistent planner may persevere with this step if there is no replacement. Recall from the discussion above, that the polarity constraint was designed to avoid the weakening of a hypothesis to produce an unprovable goal from one which was provable. If there is no alternative, then this should be tried. The result may be a provable goal, but we have no guarantee of that. The heuristic is, in general, that we wish to be assured that we are not destroying our chances of proving the goal, but if there is no other option then we can perform such "risky" steps.

Planning for Function Symbols

The remaining feature of the f/p space which is not available for inspection at the predicate level is the set of function symbols, recorded in the minor part of the triples. Two triples are said to match in the f/p space if they have the same major and minor parts, and complementary polarities. To apply a triple exchange to an abstracted formula we have to ensure that each triple in the input side of the exchange can be matched in this way to some member of the goal.

In *SS1* we insist that the secondary inputs of the exchange have partners in the goal without expending any effort to introduce them. However the system is prepared to invest some effort to introduce a partner for the primary input of the triple. The system matches the secondary inputs; if they cannot be matched then the step is rejected and a new step found which will have the same effect. If they do all match, then that task is to match the primary input triple. The best case is when the primary triple of the input set of the exchange has an matching triple in the goal. Then the output set of the rewrite rule is added to the goal to be proved and the step is accepted by the f/p planner. If the step was originally planned to introduce the predicate Q into the goal, then the triple which has Q as its major part is distinguished in the new goal.

Suppose that there is no matching triple in the goal to be rewritten. We are guaranteed by the predicate space that there is a triple with the same major part, and by the polarity matcher that this has the opposite polarity, so the difference must be in the members of the minor parts of the triples.

One simple case is that the minor part of the triple in the goal is a superset of those which appear in the exchange. In this case the step is accepted, and the extra functions which appear in the goal, are carried into the output set of the exchange.

Example: Suppose that (xxiv) is the formula to be rewritten, and (xxvi) the rewrite rule whose use has been planned to move from predicate P to predicate Q. The abstractions are (xxv) and (xxvii).

$$P(x,f(y),g(z)) \wedge R(y,z) \qquad (xxiv)$$

$$\{ \langle P, \{f,g\}, + \rangle, \langle R, \{\}, + \rangle \} \qquad (xxv)$$

$$P(x,f(y),z) \Rightarrow Q(z,x) \wedge S(y) \qquad (xxvi)$$

$$\{ \langle P, \{f\}, - \rangle \} \Rightarrow \{ \langle Q, \{\}, + \rangle, \langle S, \{\}, + \rangle \} \qquad (xxvii)$$

There are no secondary inputs to the rewrite rule, and there are triples with the same predicate and complementary polarities. However the minor parts of the triples involving P are not the same, the triple in the formula is a superset of that in the triple exchange. Clearly the rewrite rule is applicable in this case, the question is where in the output of the triple exchange g appears. Any decision that is made here has to be arbitrary, since in the abstraction space we do not have the details of the variables of the full space, we cannot say where the extra function will appear. In *SS1* we assume that such extra function symbols will appear in the minor part of the primary triple of the exchange, so that the output of the rewrite rule in this example will be $\{ \langle Q, \{g\}, + \rangle, \langle S, \{\}, + \rangle \}$.

For the example above, this is the correct choice, but this is merely coincidental. Had the variable z appeared in the S atom of the output of the rewrite rule and not in the Q atom, then the choice would be wrong. The justification for this heuristic is that it is a "worst case" assumption. This is so since a triple involving Q will be the primary triple of the next step, and the problems that g will cause if any will arise in this step.

This assumption may result in the plan either,

1. containing steps which are not eventually necessary, or,

2. not being executable.

Case 1 can arise when later in the plan we seek to eliminate the extra function from its assumed location. If it is not in fact there, then these steps will be unnecessary.

Case 2 arises when a later step *requires* the function symbol to appear in this location, but that when the plan is executed, the function appears somewhere else. Then the execution of the plan will fail.

In either case the failure cannot be detected at the time the planner is working in the f/p abstraction space, and so we leave it to the program which executes the plan to deal with these causes of failure (see section 4.4.2.1 on page 107).

The worst case for the f/p planner is when there are function symbols in the primary triple of the input of the rewrite rule which are not present in the corresponding triple in the

formula. In this case the system has to find a sequence of rewrite rules which will exchange the set of function symbols in the goal triple for those in the exchange triple. If such a sequence can be found then these steps are added into the plan before the step that is being checked.

The system has the heuristic that it should not alter the major part of the triples to be matched in making the minor parts the same. This is because the predicate space plan is designed to make the predicates the same, and if the f/p planner introduces steps to alter the major part then the plan may no longer apply. However, if this heuristic cannot be obeyed the system will move to another major part if it can also find a rewrite rule which does not alter the minor part, and returns the major part to what it was before. Finally, if the minor parts can still not be corrected, then the system will attempt to find an exchange which would help to exchange the functions, but that requires a different major part. If such can be found, then the system will try to find a sequence of subgoals to move to this new major part.

Example: The rewrite rule (xviii) has the effect of exchanging $\langle \in, \{2\}, Pol \rangle$ for $\langle \subseteq, \{\}, -Pol \rangle$.

$$x \in 2^y \Leftrightarrow x \subseteq y \qquad (xviii)$$

If this is the only rule that the system has to eliminate the function 2, then it must be used. The planner will then attempt to find a plan which will rewrite $\langle \subseteq, \{\}, -Pol \rangle$ to $\langle \in, \{\}, Pol \rangle$. The definition of $\subseteq$ will do this, so these two steps will be added to the plan, in order to eliminate the function 2 within the major part, $\in$.

Example: goal (U) has f/p triples:

$$\{ \langle \subseteq, \{\cap\}, - \rangle, \langle \subseteq, \{\}, + \rangle \}$$

$$a \subseteq a \cap b \ \vdash^? \ a \subseteq b \qquad (U)$$

The minor parts of the triples cannot be made the same by any of the exchanges that have been given. The system notices that the only undesirable difference between the two triples is the function $\cap$. It further notes that there is no exchange which will eliminate this symbol while the major part is $\subseteq$, but that the triple exchange associated with (iv) would eliminate this if the major part were $\in$.

$$x \in y \cap z \Rightarrow x \in y \wedge x \in z \qquad (iv)$$

$$\{ \langle \in, \{\cap\}, Pol \rangle \} \Rightarrow \{ \langle \in, \{\}, -Pol \rangle \} \qquad (xxviii)$$

The program then has to find a exchange that will exchange $\subseteq$ for $\in$. (i) will do this. The plan that results from this search then is the application of (i), followed by the application of (iv).

In the predicate space the goal was to produce the same predicate as a conclusion and as a hypothesis, similarly, in the function/polarity space the goal is to introduce triples with the same major and minor parts, but with complementary polarities. When all of the steps in the plan have been approved or amended by the function/polarity space planner it remains to check that there is such a pair of triples in the abstracted goal. If there is, then the plan has succeeded - at least as far as this level of detail is concerned. If on the other hand, there is no such pair then there is still more work to be done. We are guaranteed that there will be two currencies with the same major part (by the predicate space plan); all that remains is to correct the minor parts by introducing or eliminating elements of the minor parts by using further exchanges within that major part. A good example of this is (U) above. Since the predicates of the hypothesis and conclusion are both the same, the predicate space planner will produce empty plans to be executed. In the f/p space these plans are accepted and only then does the planner consider matching the hypothesis and conclusion in this space.

4.4. Execution and Recovery From Failure in the Full Space

When the plan is complete it must be executed in the problem space. This involves applying each of the rewrite rules in turn to the given goal. Since the plan was created by virtue of abstractions from the actual rewrite rules there is the possibility that some rule will fail to apply. The following are common causes of failure:

- *The Permuted Arguments Problem*: There is unification failure resulting from functions occurring in different argument places than those required by the rule.
- *The Shielding Function Problem*: The concepts appear in the right argument place, but the internal structure of the argument is incorrect for the application of the rule.
- *Connective Structure Failure*: There is a difference in connective structure between the formula to be rewritten and the input of the rewrite rule.
- *Plan Assumption Problem*: An assumption about the location of a function symbol has been made in the course of planning, and the function does not appear in that location.

When any of these problems arise it is necessary to make an analysis of the arguments and connectives of the formulae, since it must be these features of the goal that cause the plan to fail.

In the following subsections I suggest some techniques to deal with these types of failure.* As with all the problems that were discussed in relation to the previous abstraction spaces, these problems can cause the plan to fail irreparably. Quite simply, it may be impossible to apply the step as anticipated. In such a case it is necessary to attempt to replan. Some degree of failure is inevitable as a result of using abstraction spaces in which to plan.

In general it is the currency exchanges which have no effect at one level of abstraction which are the most useful at the next level up. These are the *null currency exchanges*. This is because the planner has spent some time planning to get a particular currency into the goal to achieve some result. In the f/p space there was a preference for triple exchanges which altered the minor part of a triple without altering the major part (predicate). These exchanges are just the null exchanges of the predicate space. Since the predicate plan has introduced identical predicates, we do not want to use exchanges which alter them in the f/p space, unless it is impossible to construct a plan otherwise. The use of the null triple exchanges in the full space follows this pattern and is a powerful tool for fixing both the shielding function and permuted argument problems.

Null currency exchanges are never considered in the abstraction space to which they belong. This is because currency exchanges are chosen on the basis of their effect on the goal to be proved. The planner will never include a step that has no effect as far as that abstraction space is concerned. No mechanism is required to ensure that these exchanges are not considered, it happens as a consequence of the method of planning, and of the view of rewrite rules as having effects. For this reason, the problems that are associated with, for example, commutativity and symmetry axioms are competely sidestepped in gazing.

4.4.1. The Permuted Arguments Problem

The *permuted arguments problem* is the name given to failure caused by arguments not being in the "right" positions for the execution of a planned step. This arises in the use of gazing because there is no record kept of the argument positions in the abstraction spaces. Thus the system might try to apply the rule $a \cap c = b \Rightarrow C$ to the atom $b = a \cap c$. The abstractions of the input side of the rule and formula to be rewritten are the same in the abstraction space, so the step appears to be appropriate. This problem can be tackled by considering the null currency exchanges of the f/p space. In particular, null currency exchanges in this space arise from facts in the theory which assert for example: symmetry or transitivity of predicates or commutativity or associativity of functions. If null currency exchanges exist for the currency under consideration, it is necessary for the system to determine whether the rewrite rule of which it is an abstraction "says the right thing" about the concept involved. In this case it is symmetricity that is required of the predicate $=$. The

*Each of these techniques may be used to solve failure due to one of the above causes. However, it is difficult to apply these techniques when these problems occur in combination. The solution to this problem requires more research.

user of the VOYEUR system is allowed to state metafacts about the predicates and functions of the theory (Eg = is symmetric), and then have the system use these in this way.

Associativity and commutativity present problems for traditional theorem provers which have been circumvented by the approach adopted in gazing. In particular adding the axioms to a theorem prover is disastrous unless great care is taken. The problem is that the output of, for example, the commutativity rule is a term of exactly the same form as the input, and the commutativity rule may then be applied again. Of course, repeated application of such a rule is pointless, and should be avoided - at best the prover does redundant work, and at worst loops infinitely.

One approach to solving this problem is to adopt a *canonical form* for terms and require the prover to rewrite all terms into this canonical form. For example, a canonical-form for the associative and commutative operator +, might be left associative, with the arguments in alphabetical order. In such a scheme, $((((a+b)+c)+c)+e)$ is the canonical-form of $(c+a)+(b+(e+c))$.

Thus the associative and commutative axiom are only used on terms which are not in canonical form, and they are used repeatedly on such terms. When matching two terms in canonical form the properties of the operator can be ignored. There are two objections to this solution to the problem. The first is the difficulty of defining the canonical form. The chief constraint on the canonical form is that any two terms that are unifiable when considering associativity and commutativity have unifiable canonical forms. In some cases the definition of an appropriate canonical form can be non-trivial. A more important consideration is that, since it is not possible to determine in advance which terms are going to require canonicalization, all terms appearing in the proof have to be rewritten. In particular, if two canonical terms unify with a substitution θ, the result of applying θ to the terms need not be another canonical term. Thus it is necessary to canonicalize all terms that are derived in the course of the proof, as well a those used in the statement of the conjecture. In proofs involving many complex terms, this may involve a significant amount of work, much of which may be redundant.

Another approach to the problem of commutativity and associativity is to build knowledge of the properties of the operators into the unification routine. Thus to build in commutativity we alter the unification routine so that it will unify $a \bullet b$ with $b \bullet a$ if $\bullet$ is commutative, and not if otherwise. Similarly we allow $(a \bullet b) \bullet c$ unify with $a \bullet (b \bullet c)$, whenever $\bullet$ is associative. There are two problems with this approach. The first is that when given complex terms, the number of possible ways in which the unification must be attempted grows rapidly. More importantly, the unification algorithm can no longer produce

a single most general unifier of any two terms. To see this consider the terms $(x+y)+a$ and $a+(b+z)$. The substitutions: $\{b\backslash x, z\backslash y\}$ and $\{b\backslash y, z\backslash x\}$ are both unifying substitutions, neither of which is more general than the other. The consequence of building knowledge of these properties into the unification algorithm, is that the algorithm becomes very complex and thus very slow to execute.

The advantage of the approach implemented as part of gazing is that knowledge of the properties of the operators has been built into the system, and thus the axioms expressing the properties are not added to the theory, but at the same time the penalty of adding this knowledge to the unification routine has been avoided. The only time this knowledge is invoked, is when the proof has almost gone through, and it is worth the effort to check that manipulating the arguments will lead to a proof.

4.4.2. Shielding Functions Problem

The shielding functions problem occurs when the rule which we are attempting to apply does not match the formula which we are attempting to rewrite because there is a function symbol preventing unification. For example, suppose we have the formula $x \in \mathit{complement}(a \cap b)$ which we are attempting to rewrite with the rule $x \in a \cap b \Rightarrow A$. Clearly this will not apply because of the presence of the set complementation function. This situation can arise in the gazing technique in two ways:

- when the function symbol appears in both of the currencies which are to be connected, and thus the planner has not considered this function symbol.
- when the function symbol is specifically eliminated in a later step of the plan.

For an example of the first kind of failure consider the goal (V):

$$x \in \mathit{complement}(a \cap b) \vdash^{?} x \in \mathit{complement}(a) \cup \mathit{complement}(b) \qquad (V)$$

In the function/polarity space the currencies of the goal are $\langle\ \in, \{\mathit{complement}, \cap\}, -\ \rangle$ and $\langle\ \in, \{\mathit{complement}, \cup\}, +\ \rangle$.

The planner notes that the difference between the minor parts is only the occurrences of the functions $\cap, \cup$. The planner should plan to eliminate these functions. When the system comes to apply the step to eliminate the function $\cap$ it will fail because of the occurrence of the *complement* symbol.

In both cases the problem arises because in the gazing technique the occurrences of functions in the term structure are flattened into a set. The problems can be dealt with some

further analysis of the concepts and exchanges of the theory. In the case where the plan is badly ordered it is a trivial action to reorder the plan to bring the elimination step forward to be executed immediately. However, if no such step already exists in the plan, it is necessary to find a rule in the theory which will effect the elimination of the shielding function. This can be done in exactly the same way as if the need to eliminate the function had been detected in the process of constructing the plan. The system should note that in the case where the function occurs on both sides of the turnstile, but has to be eliminated from one of its occurrences, the other occurrence should simultaneously be eliminated in order to keep the currencies the same.

4.4.2.1. Connective Structure Failure

Because the connectives are ignored by the abstraction space planners, it is possible that a given rule in the plan will not be applicable to the formula that it is meant to rewrite. Consider attempting to rewrite the formula $A \wedge B$ with the rule $A \vee B \Rightarrow C$. This situation might legitimately arise as a step in the gazing plan, since the only thing that is considered in making the plan is the presence of the concepts A and B in both the formula and the rule.

This is a problem of logic. Whether it is possible to apply the rule depends only on the connectives present. For example, in the situation above it is quite legitimate to apply the rule to deduce C. This is because the formula $A \vee B$ is a logical consequence of the formula $A \wedge B$. Similarly we might recognize that a rule which has as input $Y \rightarrow Z$ would validly apply to the formula $\neg Y \vee Z$. The use of a few inference rules of this type would enable the prover to patch faulty plans in some cases.

Solving the connective structure problem is the only use for logical inference within the gazing technique. However the rule set of the theorem prover is not available to solve this problem, just a small set of carefully selected rules which alter connectives. Thus the "logic before theory" heuristic of page 78 has not been violated. This heuristic says that the logical rules of the theorem prover are not allowed to intervene once the planning of database inferences has begun. The rules available to solve the connective structure problem are useful only in this specific context.

The number of cases that can arise to cause the connective structure problem is quite small, this is because the polarity of corresponding formulae are checked in the function/polarity space. This causes many of the possible clashes between pairs of connectives to be ruled out at this level. It is possible, in principle, to supply rules for each of the remaining cases which determine whether it is possible to complete the application of the rewrite rule, but this has not been done in the implementation. In the extreme case a theorem prover could be used to deduce the input side of the rewrite rule from the formula to be rewritten. However this approach reintroduces the full theorem proving problem. If the

rewrite rule may not be applied to a particular formula it is necessary to find a replacement step for the failed rewrite rule. Since the failed step was to be used for a particular reason the planner may be guided to find a replacement rule which has the same effect on the sequent to be proved. The connective structure failure problem, like the other problems considered in this section, could be dealt with at the abstraction level with a suitably defined abstraction space. However, I have preferred to allow for these problems at execution time since otherwise search in the abstraction space could become expensive as more factors have to be taken into account.

4.4.3. The Plan Assumption Problem

The plan assumption problem arises when the planner has assumed, in the absence of other information, that a function appears in the primary output triple of a rewriting step when in fact it does not. This can have two effects corresponding to cases 1 and 2 of page 101.

1. The plan contains steps which are not eventually necessary, or,

2. The plan is not executable.

In either case the plan will fail due to the absence of the required function symbol. In case 1, the step has been planned specifically to eliminate the function. But since the function does not appear in the assumed location, we can simply ignore the planned step. We have no need to eliminate the function symbol if it is not already there.

In case 2 the plan requires the presence of the function symbol in order to carry out the next step. Here we cannot apply the step, since the function is missing, and eliminating the function is not the intended effect of the step. In this case we must return to the planning stage and attempt to find a new step which has the required effect. Of course, having got this far in the execution of the plan, the prover now knows the exact location of the function symbols in the formula being rewritten, and so may be able to find a new step in the light of this new information.

4.5. An Example Proof by Gazing

In this section I present an example proof as it is carried out by gazing. The proof is of the goal (W).

$$a \cap b = a \;\vdash^{?}\; a \subseteq b \qquad (W)$$

We will assume that the database contains (v), (vi), (i) and (iv).

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x = y \Rightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

$$x \subseteq y \Rightarrow \forall z.\, z \in x \rightarrow z \in y \qquad (i)$$

$$x \in y \cap z \Rightarrow x \in y \wedge x \in z \qquad (iv)$$

These are the rewrite rules that are necessary in the proof. The effect on the proof of having additional rules depends on those rules and their associated exchanges. In particular, suppose that the prover already had this conjecture in its database as a rewrite rule, then we would expect that the proof would proceed by the single application of that rule.

The first step is to abstract the goal to the currencies of the predicate space. The result is as in (X)

$$\{=\} \vdash^{?} \{\subseteq\} \qquad (X)$$

In this case the choice of conclusion and hypothesis predicate is trivial, we will have the conclusion predicate $\subseteq$ and the hypothesis predicate $=$. Examining the gaze graph shows us that there is a path directly from $=$ to $\subseteq$, namely the application of the rule (vi). So the sketch plan uses just this rule. Notice that if the statement of the goal was already in the database, this would itself have been a candidate for the path.

Having made a plan in the predicate abstraction space, the system passes this to the function/polarity space planner. The goal in this space abstracts to (Y).

$$\{\langle =, \{\cap\}, - \rangle\} \vdash^{?} \{\langle \subseteq, \{\}, + \rangle\} \qquad (Y)$$

Retrieving the function/polarity abstraction of (vi) which is (xxix), and applying it to the atom involving $=$ gives us the new goal in (Z).

$$\{\langle =, \{\}, Pol \rangle\} \Rightarrow \{\langle \subseteq, \{\}, -Pol \rangle\} \qquad (xxix)$$

$$\{\langle \subseteq, \{\cap\}, - \rangle\} \vdash^{?} \{\langle \subseteq, \{\}, + \rangle\} \qquad (Z)$$

Since all of the plan is now completed, the final step is to attempt to connect the conclusion and hypothesis triples. However, the system will notice that there is an extra function symbol in the hypothesis, namely $\cap$. The system will first attempt to find a new major part within which it would be able to eliminate this function, and then attempt to move to that major part through the use of another rule. (iv) has the effect of eliminating $\cap$ with the major part of $\in$, and (i) moves from $\subseteq$ to $\in$, so the system would apply first (i) and then (iv) to the goal, producing first (AA), and finally (BB).

$$\{\langle \in, \{\cap\}, -\rangle, \langle \in, \{\cap\}, +\rangle\} \vdash^{?} \{\langle \in, \{\}, -\rangle, \langle \in, \{\}, +\rangle\} \quad (AA)$$

$$\{\langle \in, \{\}, -\rangle, \langle \in, \{\}, +\rangle\} \vdash^{?} \{\langle \in, \{\}, -\rangle, \langle \in, \{\}, +\rangle\} \quad (BB)$$

Notice that step (i) was applied to both sides of the sequent, since its aim was to move the entire goal into another currency since the major parts were already the same in both the conclusion and hypothesis.

Moving to the full space, and applying the rewrite rules (vi), (i) and (iv) leads to goal (CC), which is provable by logic alone.

$$(x \in a \rightarrow (x \in a \wedge x \in b)) \wedge ((x \in a \wedge x \in b) \rightarrow x \in a) \quad (CC)$$

$$\vdash^{?}$$

$$(z \in a \rightarrow z \in b)$$

4.6. Summary

In this chapter I have described the technique of gazing, which enables a theorem proving program to select relevant rewrite rules from the mathematical theory within which it is working in order to prove theorems. The technique works by abstracting from the rewrite rules of the theory, and using these abstractions to create a plan specifying which rewrite rules are to be used in the problem space. The plan is made by considering concepts that appear in the goal and the effect of a rewrite rule on those concepts. A plan is constructed to obtain the same concepts with opposite polarities in the goal. Since the plan is made using abstracted information it is possible that the plan cannot be executed as desired. This problem can often be dealt with in the gazing technique and examples of the recovery techniques are also presented.

Chapter 5
Characterizing the Search Spaces of Gazing and RUT

5.1. Overview

In chapter 3 I described RUT, a theorem prover which carries out proofs in a natural deduction system, and in chapter 4 I described the gazing technique which enables the prover to carry out the database inferences of RUT in a different way. In this chapter I show that search is saved by carrying out database inferences using gazing rather than using the techniques of RUT. In so doing, I demonstrate that gazing is a powerful, and at the same time computationally feasible, inference rule.

The argument of this chapter will fall into two parts. The first part is motivated by the fact that gazing, by performing all of the necessary database inferences in sequence, eliminates the possibility of carrying out a number of logical inferences which would be redundant. This is because the logical inference rules are not applied to the sequents that occur between the applications of rewrite rules. The effect of this is that the proofs that are produced by gazing are shorter and involve fewer redundant steps.

The second part of the argument shows that the rewrite rules in the database may be preprocessed so that the gazing technique can be carried out without search when given a conjecture to be proved. This means that gazing can be applied very quickly when required. Clearly the preprocessing step requires computation and this overhead cannot be ignored. However I will show that the preprocessing can be carried out reasonably efficiently. This means that it is feasible to carry out the search required by gazing when proving a conjecture. This is a necessary part of the argument since, if the computation involved in applying the gazing rule were greater than the saving that is made by using the rule, then its use would not be advantageous.

In section 5.4 I consider the problem of when the use of gazing is genuinely beneficial. When given a theory in which the rewriting steps cannot "go wrong", in the sense of leading

to an unsuccessful search for a proof, the worst that can happen to a prover not guided by gazing is that it carries out redundant inferences. In such a circumstance gazing may not aid the prover.

In this chapter I imagine a prover, called GAZER*, which has the same logical inference rules as RUT, with the exception that all of the database inferences are carried out by gazing. GAZER differs from RUT in the following ways:

1. Reduce rules which implement database deductions (Eg, the definitions of function symbols) are not available to GAZER.

2. Peek forward chaining is disallowed in GAZER, although normal forward chaining is still legal.

3. GAZER is not allowed to use the peek, expand-hypothesis, and expand-conclusion inference rules.

Also, we will suppose that the gazing inference rule appears last in the database of inference rules. The effect of this is twofold: first, it ensures that GAZER implements the "logic before theory" rule (chapter 4, section 4.2) rigorously. Thus the database rules are used only if the conjecture cannot be proved by the logical inference rules. The second effect is that once database inferences have been found necessary for the proof, their use is carefully planned, and the plan is then applied without allowing the application of logical rules until all of the planned inferences have taken place. This is in accord with the common currency model which was described in chapter 4 (section 4.2.1). The common currency model suggests that the goal of database inferences is to exchange the currencies of the conjecture until there is a currency which is common to both the hypothesis and conclusion. First the proof of the conjecture is attempted within the given currency, only if this fails is the gazing technique invoked to introduce a common currency, and only then are the logical rules used to attempt a proof of the resulting goal.

In section 5.2 I will indicate how the use of gazing, and the "logic before theory" heuristic, cause GAZER to carry out less search, in general, than RUT. However the gazing inference rule itself involves some search in the abstraction spaces, and if this were computationally expensive then the technique may still not save search overall. In section 5.3 I will show that this is not the case, and so demonstrate that gazing is a computationally feasible technique for adoption by a theorem prover.

---

*Both GAZER and RUT have been implemented as modes of the *Voyeur* theorem proving system. This system, when in 'Rut' mode carries out proofs according to the inference rules of RUT, and in 'Gazer' mode according to the inference rules of GAZER. For the argument of this chapter it is easier to think of these modes as two separate programs. Voyeur is implemented in the PROLOG programming language.

5.2. The Savings made by Gazing

The use of the gazing inference rule saves search in two main ways. First, the use of gazing prevents the redundant application of rewrite rules. This is possible because gazing plans to use only those rewrite rules which are necessary to the proof. The effect of this is not only that the rewrite rules themselves are not applied, but also that the complex formula that would result from the application of the rewrite rule does not appear in the proof. This results in the saving of logical inference. In subsection 5.2.1 I will give an example of this saving. Gazing also saves search by not allowing the logical rules to act upon the sequents that would occur between applications of rewrite rules. In RUT, once a rewrite rule has been applied, the rewritten goal is "given back" to the entire prover. It is possible that a rewrite rule will introduce a connective that can be acted upon by a logical rule, where gazing would have rewritten the new formula by another rewrite rule. The immediate application of the logical rule might split the rewritten formula into two parts, and the rewrite rule that gazing would have applied may not apply to the parts. Hence the intervention of the logical rule might produce goals that cannot be proved, where applying a second rewrite rule would result in a proof. This is described in more detail in subsection 5.2.2.

5.2.1. "Logic Before Theory"

The implementation of the "logic before theory" heuristic, which was described in chapter 4, can result in GAZER carrying out much less work than would be performed by RUT. In particular, some rewriting that is carried out by RUT is redundant. Consider, for example, sequent (DD).

$$\vdash^{?} (x \in a \cap (b \cup c) \wedge A) \rightarrow (x \in a \cap (b \cup c)) \qquad (DD)$$

The reduce rules, which expand function definitions, occur before the matching rule in the RUT database. This has the effect of causing (DD) to be reduced before the matching of hypothesis and conclusion is attempted. The proof, which should be quite straightforward, then becomes quite complex. It is shown in figure 5-1.

I present the proofs here in a standard representation for proof trees. Each proof consists of a number of lines each of which consists of three parts: the *line label*, the *statement* and the *justification*.

- The line label appears in parentheses to the left of the line, it serves only to name the line for reference by the justifications.
- The statement is the sequent that is asserted to be true by the line, and,
- The justification is a reference to the line(s) which show that the statement is

(8) $sc1 \in a \wedge sc1 \in c \wedge A \vdash^{!} sc1 \in a \vee sc1 \in b$ ()

(6) $(sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \vdash^{!} sc1 \in a \vee sc1 \in b$ (8)

(10) $sc1 \in a \wedge sc1 \in c \wedge A \vdash^{!} sc1 \in c$ ()

(11) $sc1 \in b \wedge sc1 \in c \wedge A \vdash^{!} sc1 \in c$ ()

(7) $(sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \vdash^{!} sc1 \in c$ (10 11)

(5) $(sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \vdash^{!}$
$(sc1 \in a \vee sc1 \in b) \wedge sc1 \in c$ (6 7)

(4) $\top \vdash^{!} (sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \rightarrow$
$(sc1 \in a \vee sc1 \in b) \wedge sc1 \in c$ (5)

(3) $\top \vdash^{!} (sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \rightarrow sc1 \in a \cup b \wedge sc1 \in c$ (4)

(2) $\top \vdash^{?} (sc1 \in a \vee sc1 \in b) \wedge sc1 \in c \wedge A \rightarrow (sc1 \in (a \cup b) \cap c)$ (3)

(1) $\top \vdash^{!} sc1 \in a \cup b \wedge sc1 \in c \wedge A \rightarrow sc1 \in (a \cup b) \cap c$ (2)

(0) $\top \vdash^{!} sc1 \in (a \cup b) \cap c \wedge A \rightarrow$
$sc1 \in (a \cup b) \cap c$ (1)

Figure 5-1: Proof Of (DD), by RUT*

indeed true. The justification appears to the right of the statement, and may be empty. If it is empty the statement is true by an axiom of logic.

All of the search in the proof is eliminated in GAZER since the logic before theory rule is strictly applied. In this case the proof proceeds by promoting the conjunction, and then matching the conclusion with the appropriate conjunct of the new hypothesis. This proof is given in figure 5-2.

(1) $sc1 \in ((a \cap b) \cup c) \wedge A \vdash^{!} sc1 \in ((a \cap b) \cup c)$

(0) $\top \vdash^{!} sc1 \in ((a \cap b) \cup c) \wedge A \rightarrow sc1 \in ((a \cap b) \cup c)$ (1)

Figure 5-2: Proof Of (DD), by GAZER

It is important to note that the logic before theory heuristic doesn't just prevent some reductions from taking place. The effect is further-reaching than this. Many of the steps in the RUT proof are performed as a result of applying logical rules to the connectives that were introduced by applying the reductions. Thus if the formulae that are introduced by the reductions are complex, then the amount of search that is generated can be very large. In figure 5-1, sequents (1) to (4) are produced redundantly by the application of rewrite rules. The remaining sequents are produced by logical rules acting on the reduced goals in order to

*The constant $sc1$ is a skolem constant in this theorem.

perform the required matching. Notice too, that (7) could be proved immediately by matching, but because the logical rule **Cases** can act upon the disjunctive hypothesis (10) and (11) are also produced. This redundancy is caused by the relative positions of the **Cases** and **Match** rules in the rule-ordering, and the use of gazing has no effect on this.

The gazing technique therefore has two effects: first, if a match is possible it will be found before any database rules can be applied, and secondly, if no match is possible, only those database manipulations that are necessary to the proof will be performed. So, if the goal had been (EE) the necessary reduction (expanding the definition of complement) would have been performed, but the manipulation of the intersection and union symbols would still not be carried out.

$$\vdash^{?} (x \in \mathit{complement}(a \cap (b \cup c)) \wedge A) \rightarrow \neg (x \in a \cap (b \cup c)) \qquad (EE)$$

The use of the gazing technique prevents the application of redundant rewrite rules. This leads to a considerable saving of search since the result of applying a rewrite rule is often the production of a complex formula. A number of logical rules might act on this more complex formula, which would not have applied to the original formula, and thus a large amount of redundant search may be carried out.

5.2.2. The Problem with Splitting

A further problem with the RUT inference rules is that the logical rules are allowed to act on formulae between a number of rewriting steps. This is the "logic *between* theory" approach discussed in chapter 4, page 78. The interleaving of theory and logical inference can cause problems for RUT, because the logical rules can split up formulae that "belong" together. For example, consider the axioms (xxx) and (xxxi) and theorem (FF):

$$P(f(a)) \vdash^{?} C \qquad (FF)$$

$$P(f(x)) \Rightarrow A \vee B \qquad (xxx)$$

$$A \vee B \Rightarrow C \qquad (xxxi)$$

In RUT the proof is approached by first using (xxx) to rewrite the hypothesis: the result is the new sequent (GG).

$$A \vee B \vdash^{?} C \qquad (GG)$$

Then the logical inference rule **Cases** is applied to produce two subgoals (HH) and (II).

$$A \vdash^{?} C \qquad (HH)$$

$$B \vdash^{?} C \qquad (II)$$

Both of these subgoals are unprovable. Even if (xxxi) were available to RUT as a reduce rule, the system would not be able to apply it, since the disjunction has been split. After some search the prover will fail with one of these subgoals. Since RUT has the ability to backtrack and remake the choice of inference rule, (xxxi) will then be applied to (GG), and the proof will succeed. Since the UT provers, do not have this ability, they fail to prove this theorem. The crucial point here is that the **Cases** and **And Split** rules break up formulae, and create two subgoals with part of the broken formula in each goal. If a rewrite rule applies to the formula, but not to either of its parts, then the application of this rule will be prevented by a previous application of the logical inference rule. This is an example of the failure of the "logic between theory" approach.

Since A, B and C could be complex formulae, the amount of search that might be expended in the proofs that do not succeed is very large. If, for example, the concepts in A, B and C are defined in the theory, then the expansion of these concepts will take place. This rapidly makes it computationally infeasible to carry out this search since the number of definitional expansions that might be tried, and the logical inferences that might be possible, is very large. Exactly how much extra search is performed depends on the criticalities of the predicates, and on the number of function symbols, which occur in the goal. In either case when an expansion is carried out the logical prover is called on the new goal.

In GAZER, all of the database inferences that are necessary in the proof of the goal are planned in advance, and then applied serially. The planner would plan to use (xxx) and (xxxi) in that order and then apply the two rules without letting the logical rules intervene. That is, once the proof attempt using logic alone has failed, the planner plans to use some database inferences, carries out these inferences, and then readmits the logical rules. Thus, we have "logic before theory", but once it is determined that the theory must be used, the inferences are planned and all carried out before retrying a logical argument. The result of gazing applied to (FF) is the trivial sequent: $C \vdash^{?} C$.

Preventing the application of logical inference rules between database rules can never lead to the failure of a proof which would otherwise have been successful. This is because GAZER plans to apply the smallest number of rewrite rules necessary to produce a goal which has a common currency between the hypothesis and conclusion. Therefore, no intermediate goal can be proved by logic, and so nothing can be gained by applying logical rules to any such goal.

Goal (GG) above gives an example of the "logic before theory" heuristic providing the wrong guidance. If this were the initial goal for GAZER, then the subgoals (HH) and (II) would be produced, their proofs would fail and only then would gazing be invoked. This is

exactly the same as RUT, since **Cases** appears before **Reduce** in the RUT rule base. **And Split**, which is the only other splitting rule of RUT, also appears before **Reduce**, and so RUT will always have exactly the same problem with this type of splitting that GAZER has.

Note that the example above is the result of an optimistic view of the behaviour of RUT. Firstly, I have assumed that (xxxi) is available as a reduce rule of the system. Because of its form, it could not be a definition, and it would therefore have to be present in the table of reduce rules as a lemma. Secondly, if (xxx) had instead been (xxxii), and this rule had been the definition of the predicate P, then the use of this rewriting would only be available through the peeking technique (since it is the definition of a predicate occurring in the hypothesis). However, neither of the predicates occurring in the definitional formula (A and B) occur in the conclusion of the goal, so the expansion would not have been made, and the proof would have failed at a much earlier point.

$$P(x) \Rightarrow A \vee B \qquad (xxxii)$$

Notice that if (xxx) and (xxxi) are added to the hypothesis of the conjecture, then RUT is able to find a proof, despite the application of the **Or Split** rule. This is because the **back-chain** inference rule causes the conclusion C, to be rewritten to the antecedent of (xxxi).

$$(P(x) \rightarrow A \vee B) \wedge ((A \vee B) \rightarrow C) \wedge P(f(a)) \;\vdash^{?}\; C$$

$$(P(x) \rightarrow A \vee B) \wedge P(f(a)) \;\vdash^{?}\; A \vee B$$

$$P(f(a)) \;\vdash^{?}\; P(x)$$

This indicates an alternative method for solving the problems that gazing solves. The solution is to add the rewrite rules that are necessary in the proof to the hypotheses of the goal to be proved, and then let **Back Chain** do the work of selecting the appropriate rules to use. This method however requires that the statement of the theorem implicitly contains the knowledge of the proof. When the user of the theorem prover does not know the exact pieces of information that are necessary in the proof, then this method is not workable.

5.2.3. Summary of the Gazing Saving

In this section I have shown how the use of the gazing technique in a RUT-like prover, can save much search over the use of RUT's theory-inference rules. In particular two main savings are made possible by the use of gazing. The first is that redundant application of reduce rules are avoided. This not only saves the application of the rule, but also the logical inference that might be carried out on the formulae that result from the application. The second saving is made because the gazing technique plans and applies appropriate database

rules serially, without letting the system apply logical inference rules between the applications.

Having shown that implementing the gazing technique eliminates some of the search of RUT, it remains to show that the gazing technique doesn't introduce more search than it eliminates. In the next section I will show that the amount of search that is necessary to apply the gazing inference rule is less than that saved by its application.

5.3. The Cost of Gazing

In the previous section I showed that, by using the gazing technique, it is possible to reduce the amount of search that RUT needs to do to produce proofs. However, no account was taken of the amount of search that is carried out in the application of the gazing rule itself. It is clear from the description of the technique in chapter 4 that the application of the rule to a given sequent involves some search. For example, there may be a number of possible paths through the gaze graph linking two predicates, and the system must choose one that is appropriate given the current goal. However, the question that remains is exactly how much search has to be performed when the rule is applied, and how expensive it is to carry this out.

In this section I show that gazing can be carried out quickly once the conjecture is known. In addition I show that it is necessary to search the theory only once for each plan that might be required, since the possible plans depend only on the facts of the theory. This leads to the conclusion that it is possible to precompute all possible plans in the theory as soon as the rules of the theory are known. This would make the selection of a preconstructed plan to prove a particular sequent in the theory an almost trivial operation. The preprocessing step is not very expensive for the predicate abstraction space, but is more so for the function/polarity space.

The idea here is to divide the work that is required by the gazing technique into two parts: that which requires reference to the particular conjecture that is to be proved, and that which is dependent only on the facts of the theory. The work that is not dependent on a particular conjecture need be performed only once and the result stored for future use. The task is to produce a general representation of all plans in each abstraction space, in such a way that the checking which must be performed when the conjecture is known may be carried out quickly.

To show that this approach is sensible it is necessary to consider two things: first, that the amount of work that may be done without reference to a particular conjecture is significant, and therefore leads to a significant saving of work when the conjecture is known, and secondly, that it is possible to produce a representation of *all* plans producing a specific effect.

Recall that in both abstraction spaces the strategy for constructing plans is to find a fact which has the desired effect in the abstraction space, and then to check that it may be applied in the context of the current conjecture by considering the subsidiary inputs of the fact. In the predicate abstraction space, the goal might be to exchange P for Q, and an appropriate fact might also require subsidiary inputs S. Whatever the conjecture this fact will be a candidate for exchanging P for Q, but whether the use of this fact is acceptable depends on the presence of S in the conjecture. The expense of gazing is to determine the sequence of facts which will cause the desired effect, since this involves search among the facts of the theory. Once all ways of achieving an effect are known it is necessary only to consider each of the possible ways, to determine those whose subsidiary inputs appear in the given conjecture. This is a less expensive step since first, there are, in general many fewer possible plans for any specific effect than facts in the theory, and second, this checking can be carried out very quickly.

That it is possible to compute all plans is a corollary of the fact that the currency exchanges are isomorphic to formulae of propositional logic.* This means that whenever we ask the system to determine all possible paths in either abstraction space, we can be sure that we will eventually get a result, which will either be that there are no paths, or the set of paths that have the desired effect. This is not a strong result. While it is important to know that the task can, in principle, be carried out, it is more important to know whether it is practical to do so.

In the next two subsections I indicate the extent of the search that has to be carried out in the gazing abstraction spaces, and the degree to which this search can be carried out ahead-of-time. Finally I will outline three different approaches to carrying out the search, and suggest a preferred method.

First, some definitions that aid the discussion of algorithms:

> **Definition 60:** The *complexity of an algorithm* is a function of the amount of data that the algorithm is to process. The function describes how the number of operations required to execute the algorithm changes as the amount of data changes.

> **Definition 61:** The complexity of an algorithm is usually expressed in "Big-Oh" notation: $O(f(n))$. If the complexity of an algorith is said to be $O(f(n))$, this means that for sufficiently large n, the algorithm requires not more that $M|f(n)|$ operations, for some positive constant M.

* See facing page

5.3.1. Search in the Predicate Abstraction Space

In the predicate abstraction space, the task is to find a common rewriting of two predicates. The only part of the planning task that requires knowledge of the goal to be proved is checking that the required subsidiary inputs are present. Thus all plans which alter predicates can be precomputed provided that each plan records the subsidiary inputs that are required for its execution. This preprocessing in not expensive.

In section 4.3.2 of chapter 4 I described how the search for a plan to exchange one predicate for another is isomorphic to the task of finding paths through the gaze graph. It should be clear that since the gaze graph is constructed when the theory axioms are stated, the computation of all paths joining all pairs of predicates may also be carried out at this point. The implementation of VOYEUR allows this preprocessing step to be carried out as an option.

Clearly the particular representation that is used for the plans will effect the speed with which the necessary operations can be carried out. As an example of this, the gaze graph itself represents all of the paths between all nodes, but the information is represented only implicitly, and computation is required to extract the information that is needed. The goal of preprocessing the theory is to ensure that the amount of work that is necessary while the conjecture is being proved is kept to a minimum. The next paragraphs outline a method of preprocessing the graph which was suggested to me by Don Simon.

The first observation that may be made is that the gaze graph is a *multigraph*. That is, the graph may contain many arcs which exchange the same two pairs of predicates. These arcs will differ in the subsidiary inputs and outputs. Finding all paths through a multigraph is much more expensive than finding all paths through a monograph (a graph which has only one arc joining any two nodes), and we can simplify our task by first preprocessing the multigraph into a monograph. This is done by replacing all arcs $A_1, A_2, \ldots, A_n$ from P to Q by the corresponding multiarc *multiarc*(P,Q,[$A_1, A_2, \ldots, A_n$]). Each A_i must contain a record of the subsidiary inputs and outputs. The transformation from a graph represented by a set of arcs, each deriving from a fact of the theory, to a multigraph of this form can be performed in $n \log n$ time, where n is the number of arcs in the original graph. This is because the transformation is equivalent to the problem of sorting the set of original arcs, using their in- and out- nodes as keys, and merging those arcs with identical keys. The complexity of merging two arcs does not grow with the number of arcs, and sorting using the quicksort algorithm, [Knuth 73], is $O(n \log n)$. The number of multiarcs in the resulting graph is no larger than n.

Once the gaze graph has been transformed into a monograph G, the following algorithm may be used to construct all possible paths in G:

1. Locate a node, N, in G which has no out-arcs, if there is no such node then return the empty set of paths.

2. Delete this node and all arcs, A_i leading to it, forming graph G'.

3. Find all of the paths p, in G'.

4. Add to p the set of A_i, and the paths which may be constructed by adding a deleted arc to the end of any path in p. The result is the set of all paths in the complete graph.

This algorithm is of complexity n^2, where n is the number of multiarcs in the monograph. This is because:

- The base case, when no nodes exist in the graph, can be performed in constant time,

- In the inductive case, we are to add the deleted arcs to some of the paths already constructed. There may be at most n such paths, and we have to consider each of them, so the complexity of this step is of order n.

- At each level of recursion at least one arc is deleted, so there can be at most n levels of recursion.

Since each level of recursion "costs" n operations, and there are at most n levels of recursion, the algorithm is of complexity n^2.

A path from P_1 to P_n may be represented as a list of multiarcs:

$$
\begin{aligned}
[\ & multiarc(P_1,P_2,[A_1,A_{12}, \ldots, A_{1n}]) \\
& multiarc(P_2,P_3,[A_{21},A_{22}, \ldots, A_{2m}]) \\
& \ldots \\
& multiarc(P_{n-1},P_n,[A_{n-1\,1},A_{n-1\,2}, \ldots, A_{n-1\,q}])\ )\ ]
\end{aligned}
$$

Each such list represents all paths through the gaze graph which link P_1 with P_n, by visiting the same sequence of nodes in the graph. For any two predicates there will be a set of such paths, each visiting a different sequence of nodes. This set represents all possible paths between the two predicates. These sets of paths may be stored in a two dimensional array where one dimension represents the initial predicate, and the other the final predicate.

Once all of the paths have been computed, the problem of determining the paths between two specific predicates when a proof is being performed is a two-stage process. First, all paths linking the predicates have to be retrieved. This is a simple array access which, as is well known, can be performed in constant time; that is, the time taken to perform an array access does not depend on the size of the array, and therefore does not become expensive for large theories. Secondly, the subsidiary predicates required for each path have to be present in the conjecture to be proved. Checking this involves comparing the set of predicates that

are in the conjecture against those required for each arc in each step of the plan. For example, if we have a conjecture which involves finding a path from P_1 to P_n and we have subsidiary predicates S, then we would retrieve the path above, and then check to see whether S was a subset of the subsidiary predicates of some A_{1i}. If it is not, then this plan would be rejected, since there is no way of getting from P_1 to P_2. If there is a least one arc which enables the transition, then we examine the A_{2i} and so on. As soon as some step does not go through, this particular plan will be rejected and the next selected (the next plan will visit a different sequence of nodes, by a different sequence of arcs). At worst we have to compare S with all n arcs in the original graph*, and so this step can be performed in linear time.

To summarize the result of this subsection, the preprocessing that is necessary to produce an efficient representation of the plans takes in the worst case $O(n^2)$ where n is the number of arcs in the gaze graph. The checking that is necessary when the particular conjecture is known is $O(n)$.

If, instead of precomputing all paths joining all pairs of predicates, we prefer to wait until given a specific conjecture to prove, we would need an algorithm to determine any specific path. Such an algorithm should not be of greater complexity than that to compute all plans, since in a last resort all plans could be computed and then the desired result selected from the complete set. However, a sensible algorithm would examine only that subgraph of the whole graph whose root is the predicate with highest criticality of the two that are to be joined. Since this is at worst the entire graph, the algorithm is at worst equally expensive as the computation of all plans.

5.3.2. Search in the Function/Polarity Abstraction Space

The problem of exchanging currencies in the function/polarity abstraction space is much worse than that in the predicate abstraction space for two reasons:

1. Each triple contains a *set* of function symbols, and,
2. The set of currency exchanges that are possible depends on the major part of the triple.

The first point causes a problem because a triple exchange can be used to exchange a number of different triples for others. Consider the triple that arises from the definition of $\cap$, this is triple (xxviii) from chapter 4 and repeated below.

---

*provided that we store the results of comparisons to avoid duplicating effort.

$$x \in y \cap z \Rightarrow x \in y \wedge x \in z \qquad (iv)$$

$$\{ \langle \in, \{\cap\}, Pol \rangle \} \Rightarrow \{ \langle \in, \{\}, -Pol \rangle \} \qquad (xxviii)$$

This triple exchange can be used to rewrite any currency $<\in, Minor, Pol>$, provided that $\cap$ is a member of the set *Minor*. The resulting triple will be $<\in, Minor \backslash \{\cap\}, -Pol>$. So this triple exchange represents a number of possible exchanges. To determine exactly how many exchanges are represented by a specific currency exchange, suppose that the theory contains F function symbols. Suppose further that the function set of the input currency of the exchange has f members. Then this exchange represents 2^{F-f} exchanges.

Point 2 above indicates that the system will have to compute a different set of plans for each major currency of the theory. That is, it is not enough to have a way of matching two sets and then assuming that these sets can be matched whatever the major part of the triple within which the sets appear. The major part plays an active role in determining what exchanges are available.

First of all we should work out the total number of plans that it would be necessary to compute to precompile a particular theory. We will assume that the theory has P predicates and F functions. We will have $P \times 2^F$ different currencies in the theory. Call this number C. There are C^2 possible pairs of currencies, but for each pair we can only make an exchange in one direction. There will therefore be $C^2/2$ plans to compute. This is a very large number for even very small theories and it will be very expensive to compute all such paths, even if we could compute each individual path very efficiently.

If the theory is so small that constructing all paths is feasible then the best approach is to make sure that plans are constructed from early in the order $\prec$, working upward. Recall that the order $\prec$ orders all f/p triples, and that all rewrite rules rewrite an f/p triple, to another that is lower in the order. Thus if, $C_1, C_2, \ldots, C_n$ is the list of triples in ascending order, then the approach is to compute the paths from C_2 to C_1 first, and then C_3 to C_2, followed by C_3 to C_1. If this approach is adopted then all subplans of the plan being computed are already known, and so these results may be reused. This algorithm is however exponential in the number of concepts in the theory since in finding a plan from C_i to C_j all subsets of C_i which are supersets of C_j must be considered. In the worst case this is all (2^{F+P}) possible sets of concepts.

Exactly as in the predicate space, when given a particular conjecture and the set of plans for linking the prime triples of that conjecture, the set of plans must be checked, and those which require secondary triples not present in the conjecture are discarded. This checking step is also linear in the number of triples in the conjecture.

Computing all plans for a specific pair of currencies in the theory can also be a very expensive operation. Since in the worst case, when the triples to be joined are the highest and lowest in the theory, it is exactly the same task as finding all paths. In the general case the problem is not so serious, since the lower in the triple ordering the given triples are, the less expensive the task is.

In this section I have described the problems involved in precomputing plans in the function/polarity abstraction space. This space, like the predicate space, is propositional. This means that it is, in theory, possible to precompute all possible exchanges for a given theory. This computation could be done as a compilation step as soon as the axioms of the theory are stated to the prover, but requires an algorithm which takes time exponential in the number of symbols in the theory. This makes the precompilation of plans in this space prohibitively expensive for large theories. Despite the fact that preprocessing is likely to be out of the question for many theories, it is still the case that each result only has to be computed once for each theory. I have also shown that the cost of any such computation is linear. This means that the results of any computation may be stored and reused later if required.

5.3.3. Preprocessing The Theory

In the previous subsections I have introduced the idea that the theory can be processed to make the retrieval of plans when a proof is performed a much faster operation. However the preprocessing of a theory with many predicates and functions may be very expensive. Also, storing all of the precomputed plans may be expensive in memory. Three possibilities for addressing these problems are discussed below.

1. Compute the values each time they are required,
2. Compute the values when they are first required, and then store them for retrieval if they are required subsequently,
3. Compute all of the values as soon as the axioms are known, and then store them for retrieval each time they are required.

Each of these approaches has its advantages and disadvantages, the relevant factors being the time taken to perform the computation and the space required to store the results.

The first approach is not to precompile the theory, and to compute the plans "on the fly" each time they are required. This avoids storing the table but might require the prover to derive the same path (or subpath) many times in different proofs. However all preprocessing is avoided and no extra storage space is required, at the expense of possibly having to recompute results that have already been determined. The problem with this is that

computing the results becomes an integral part of the gazing inference rule, and thus the amount of time required to apply the inference rule is greater in this method and there may be no net gain.

In the third approach there is the possibility that some results that are never required will be computed. This wastes both the time taken to compute the result and the space taken to store it. However if enough proofs are performed, that all possible plans are required at least once then this solution is optimal in the amount of processing that is carried out. A further effect of this method is that the amount of time perceived by the user of the theorem prover as required for the application of the gazing rule, and thus the amount of time required to complete proofs is less.

The second approach combines the best aspects of the other two. The idea here is to carry out only the work that is required, as and when it is needed, and to then store the result for future reference. Initially no storage space is required, since no entries are computed in advance. Whenever the value of some entry is required by the program for a proof, this entry is computed as in the first method, and then stored as a table entry for immediate access when it is required in subsequent proofs. This has the advantage of having to compute the entries only once (as when computing the entire table), but ensures that entries are only computed on need (as in the first method). The drawback is that whenever the program is computing an entry for the first time, the time taken to perform this computation will be added to the time taken to perform the proof. This method should be preferred over the other two when the theory gets rather large, since this method ensures that only the information that is required is computed, but the best possible use is made of the results of the computation.

The current implementation of GAZER allows the user to specify which of the strategies above is to be adopted by the system.

5.3.4. Summary of the Cost of Gazing

I have demonstrated that the search required to apply the gazing inference rule could be reduced to a very small amount, providing that extensive preprocessing of the theory were carried out. This might be done by having a preprocessor which turns the theory axioms into the possible plans in the theory. This preprocessing step could be performed as soon as all of the axioms are known and has to be performed once for each theory.

In the predicate abstraction space, all of the plans can be computed in $O(n^2)$, where n is the number of arcs in the gaze graph. This is not a significant overhead. In the f/p abstraction

space however, much more work is necessary, - the best possible algorithm must be of exponential complexity. However, it is possible to compute values only by need and to store the result. Thus the set of all paths in the f/p abstraction space may be constructed incrementally.

As an alternative to computing the paths as soon as the axioms are known, it would be possible to record the results of the search for a particular plan as it was completed. This would involve the prover in carrying out the search once and only once for each entry, as in the previous method, but then recording this result in case it were required in a subsequent proof. This has the advantage of never requiring the system to carry out the computation of a plan that it does not need, but the disadvantage of requiring the computation of the initial plan to be carried out as part of the proof.

5.4. When is the use of Gazing Beneficial?

In the preceding sections I have described the savings that are gained by the use of gazing, and the cost incurred in its use. The question of when gazing is beneficial, in the sense of saving more than it costs, is still open. In this section I will consider briefly some of the features that will determine the answer to this question. The main criterion determining this is the particular theory that the prover is expected to work in. For certain theories it is more efficient to use the search techniques of RUT, and in others to adopt gazing. One of the open questions that this work brings to light is how to assess which circumstances are which.

In section 5.2 I described the way in which the use of gazing can save search over the techniques of RUT. In the example describing the elimination of redundant rewriting steps, I presented a proof, performed by RUT, which performed many redundant steps (figure 5-1). Given a very efficient rewriting rule and equally efficient logical inference rules, this proof could be performed very quickly despite the number of redundant inferences. This is true since the number of possible rewritings of the formulae in question is very small, in fact it is impossible to go wrong in the application of the rules that the system knows. Under these circumstances, the use of gazing itself appears to be redundant since RUT performs well enough. In the example describing the usefulness of performing rewriting steps without letting the inference rules intervene, I remarked (page 116) that the size of the space that would be searched redundantly was dependent on the criticalities of the predicates, and the number of functions, in the goals that could not be proved. In a theory which has a small number of predicates and functions (like the set theory that we have discussed) this amount of search might be tolerable. We might prefer to let the prover search the blind alley, and then once the subproof has failed return and discover the successful path.

From these observations we can conclude that gazing becomes more useful as the theory becomes more complex. In a simple theory, with few predicates and functions, and few rules relating these concepts, the amount of redundant search that RUT would carry out is probably tolerable. Given a theory which has many more concepts and large numbers of possible rewritings for each combination of them, then the search that gazing can eliminate becomes significant.*

5.5. Summary

In this chapter I have made a comparison of the search that is carried out by RUT with the search that would be made by a prover which performs database inferences using the gazing technique.

The main result of this chapter is in showing that the search carried out in GAZER is smaller than that of RUT, which performs the same inferences in a less structured way. There are two ways in which the gazing technique effects this: by not allowing the logical rules to be applied to sequents that occur between applications of rewrite rules, and second by being computationally efficient in the means of storing information about possible rewritings.

By carrying out all database inferences in the application of one inference rule GAZER is able to complete the job of making the concepts in the sequent correspond across the turnstile before attempting any logical deduction. This can cause a considerable saving in the search for a proof.

Since the gazing technique makes use of information that does not change once the theory is specified, the system can represent this information efficiently, in particular by computing paths through the gaze graph and storing the pre-computed results along with the axioms of the theory. While the pre-computation phase may be expensive, it is possible to carry it out in background while the user is not waiting for the proof to be completed. An alternative strategy would be to perform a compute-by-need, and store the result after computation to be looked up if it is ever needed again. This has the advantage of not requiring the pre-computation of all paths, but the overhead of having to perform each computation exactly once during prove-time. Whatever strategy is adopted for the computation of this information, and whenever it is carried out, we are guaranteed a result in finite time since the abstraction spaces are isomorphic to propositional logic.

In section 5.4 I observed that the trade-off between the saving made, and the cost incurred, by gazing depends on the complexity of the theory that the prover is working in. If the theory is sufficiently complex then gazing is certainly a useful technique. If the amount of

---

* In appendix A the savings that can be gained by the use of gazing are demonstrated. I compare the time taken to perform proofs using the *Voyeur* theorem proving system emulating first GAZER and then RUT.

search that the prover may redundantly perform is limited by the fact that the theory is not too complex, then it may be more efficient to allow the prover to perform redundant inferences provided that the system can recover from its errors.

The effect of these observations is to make an argument for preferring the use of the gazing technique over the techniques of RUT in certain circumstances. The power of GAZER is not less than that of RUT, and the required search is less in complex theories and can be performed more efficiently.

Chapter 6
Further Work, Related Work and Conclusions

6.1. Overview

In this chapter I discuss work which is related to gazing, further directions in which research into gazing might be taken, and finally draw some conclusions from the investigation presented here.

In section 6.2 I describe some of the questions that arise from the use of the gazing technique, and some suggestions for extending gazing to answer these questions. I also describe how the abstract representations of the theory to which gazing is being applied can be used to deduce more information about the properties of the theory than has been used by gazing.

In section 6.3 I describe work which is strongly related to that reported in this thesis. This work falls into 3 categories:

- Natural deduction theorem provers,
- Techniques separating logical and database inferences, and,
- The use of abstraction spaces to construct plans.

Finally in section 6.4 I bring together the main threads of the work reported here.

6.2. Further Work

In this section I suggest some possible extensions to the work presented so far. One of the interesting features of gazing is that the representation of the theory includes the relationships between the concepts explicitly in the graph structure. We can examine this graph structure for regularities, and features that might constrain the search for proofs. In particular we might make the search strategy that we use to perform the gazing step dependent on the existence of certain features of the theory.

6.2.1. Modifying the Search Strategy

Gazing allows us to select appropriate facts from the system's database of knowledge providing a unified method of dealing intelligently with definitions, lemmas and axioms. A side-effect of this is that shorter and more natural proofs are sometimes produced by the system.

One thing that might be noted about the description of gazing in chapter 4 is that the strategy used to search the abstraction spaces is a parameter of the technique. Only one strategy has so far been implemented and tested. This strategy, *ss1*, was devised by considering naive set theory, which has a small number of predicates and a slightly larger number of functions. It is not clear that *ss1* will behave well in theories with a different form. For example, in a theory with only one predicate and many functions the predicate space search will be completely trivial, and the function space search will be very complex. Indeed the rules presented for searching the function space may not be strong enough to cope with very large numbers of functions without some other constraints, which can be provided by the predicate space search. For this reason, the choice of strategy is perceived as a parameter which may be varied within the gazing technique. The question of deciding which particular search strategy to use to prove a specific conjecture within a given theory then arises.

We might conceive of a system which will analyze a given theory according to some predefined set of features. The analysis is facilitated by the representation of the theory as a graph. In fact the abstraction level is not important here, but the sort of structure that is currently used in VOYEUR aids thought about this subject. The gaze graph is the representation of the theory at the predicate level. Each fact is represented as a set of directed arcs in a graph which has the nodes labelled with predicate names. Arcs go from a set of predicates P_{IN} to a set of predicates P_{OUT} if there is a fact which rewrites some formula involving the predicates in P_{IN} to a formula involving those in P_{OUT}. Thus we can conceive the entire theory as a graph and use graph theoretic concepts to analyze the structure. The result of this analysis will determine not only how the prover will search for

proofs of conjectures, but might also enable the system to recognize certain "key" facts in the theory, and perhaps even make requests for more information about the theory.

The open question here is what features of the theory are relevant to the choice of search strategy. In the following subsections I outline a few possibilities.

6.2.2. On "Key" Arcs

One useful concept from graph theory is that of a bridge. An arc in a connected graph is a *bridge* if, the graph that results by removing that arc is disconnected. Clearly we can generalize this concept to apply to sets of arcs, and this allows us to label certain facts about the theory as crucial to that theory, as without them the theory would literally "fall apart". As an example, consider the theory formed by the definitions and lemmas given below.

$$x = y \Leftrightarrow \forall z.(z \in x \leftrightarrow z \in y) \qquad (v)$$

$$x = y \Leftrightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

$$x \subseteq y \Leftrightarrow \forall z.(z \in x \rightarrow z \in y) \qquad (i)$$

$$x \in \{y\} \Leftrightarrow x == y \qquad (xxxiii)$$

I use the notation $==$, to represent equality of individuals as opposed to $=$ for equality of sets. Without (xxxiii) the whole theory falls into two parts: no rewrite rules exist which allow us to exchange any concept for $==$. Thus this equality might be considered to be "key". No other fact in the theory has this property.

It is clear that the property of being a key lemma is relative to the particular conjecture to be proved. (xxxiii) is clearly only important in proofs that involve concepts from each of the parts of the graph between which the fact is a bridge, but it is vital in such proofs.

This notion might be used by the prover in one of two ways:

- Altering the strategy,
- Requesting information.

In the first case the system is called upon to notice that certain arcs are bridges in the theory. The system might then partition the predicates in the graph into the sets which lie on either side of the bridge, I will call these sets *islands*. When given a conjecture to prove which requires deducing a conclusion involving a predicate from one island, from hypotheses involving predicates from another island the system should then note that the bridge fact will be needed, and adjust its search strategy to incorporate this fact. This could enable the

system to "turn the search space inside out", searching from the ends of the bridge to the hypothesis in one direction and to the conclusion in the other.

If the graph of the theory were originally disconnected then the system could ask the user to suggest conjectures which would link the islands. One proved, the conjecture could be used as a rewrite rule and added to the theory.

6.2.3. On Conjecturing

An analysis of the graph might lead the system to suggest facts that it would be useful to know. The current representation of the theory does not have enough information to make conjectures as to theorems of the theory, but it could make requests of the user that a direct link between two predicates would be a useful thing to have, or that a particular weak link would be more useful if it were supplemented by additional arcs.

There are two occasions when the system might be able to suggest the addition of new arcs. First, by analyzing its failure to perform particular proofs. In an attempt to find a set of facts which link two predicates, the system could fail because a required link was not present. When this happens, the system might request that the user supply some lemma which could be absorbed into the theory to provide the link. Secondly, as a theory develops, the system might notice that a particular set of steps is used frequently, or that a particular pair of predicates has no link, and suggest to the user that a theorem be proved, or a definition made, to remedy this situation.

One of the first things that happens when the concept of ordered pairs are defined in set theory, is that a lemma is proved which enables the mathematician to "forget" the definition and to use this property. The definition is given in (xxxiv) and the lemma in (JJ).

$$x \in \langle y,z \rangle \Leftrightarrow x = \{y\} \vee x = \{y,z\} \qquad (xxxiv)$$

$$\vdash^{?} \langle x,y \rangle = \langle x',y' \rangle \leftrightarrow x == x' \wedge y == y' \qquad (JJ)$$

The definition provides us with a weak link between $\in$ and $=$. The mathematician clearly realizes that this link is not of much use, and constructs a much more useful link between $=$ and $==$. Notice that there is no such link already in the theory, so despite the fact that this link is still weak (it has no variable arguments at the top level), it creates a new link between these predicates.

6.2.4. Summary

In this section I have described gazing and some of the directions in which it would be possible to take this work from its current state. I have focused on the possibilities of analyzing the theory through its representation as a gaze graph, suggesting that it might be possible to characterize theories by such an analysis and use their characterization to determine the best ways of carrying out proofs in the theory. I have suggested that the presence of bridges in the graph of the theory is an important feature, since the facts which result in the presence of bridges are "key" in some important sense.

These factors, and possibly many others, might be used to determine the best strategy to use in the search for a proof. The presence of bridges in the theory allows us to guide the search towards the end of the bridges when we know that they have to be crossed.

Important questions for this work are:

- Are the features suggested here powerful enough to provide useful categorizations of theories?
- What other features are also important?
- Is it possible to devise special purpose search strategies for searching theories of particular forms?

6.3. Related Work

In this section I describe work that is related to that reported in this thesis. This work may be related in one of 3 ways:

1. The implementation of natural deduction to produce an automatic or interactive theorem prover, [Pastre 77, Reiter 76, Reiter 73, Brown 78],
2. The uses of abstraction spaces: to construct plans which when executed will have specified effects, [Sacerdoti 74, Sacerdoti 77], or to produce abstract proofs which can be used to guide search [Plaisted 80, Plaisted 86, Cvetkovic & Pevac 83].
3. The separation of inferences into what I have called logical, and database, inferences, [Stickel 85].

I shall consider work in each of these classes in turn.

6.3.1. Natural Deduction Theorem Provers

In chapter 2 I described two programs which implement natural deduction theorem provers. These are far from being the only examples of such provers, and in this section I describe the main features of some other provers. The provers reviewed here were chosen to illustrate possible means of controlling database inferences.

Pastre's DATTE

Pastre's theorem prover, DATTE, [Pastre 77] is very similar to the UT provers and RUT, however an interesting feature of her program is the use of a graph to represent some hypotheses.

The basic idea is similar to that of Bledsoe's inequality mechanism, namely that some hypotheses may be removed from the conjecture and recorded in a more useful form for the prover to work with. All binary relations are treated this way in DATTE, they are removed from the conjecture and stored in a graph representing known relations between objects. Consider for example the representation of the conjecture (KK) as the goal (LL) and graph 6-1.

$$x \in a \wedge a \subseteq b \wedge b \subseteq c \wedge P(x,a,b,c) \vdash^{?} x \in c \qquad (KK)$$

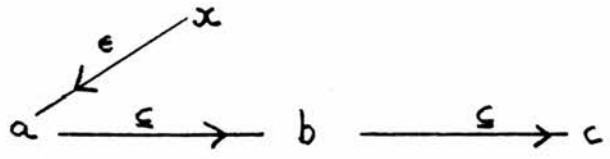

Figure 6-1: Graphical Part of Conjecture

$$P(x,a,b,c) \vdash^{?} x \in c \qquad (LL)$$

The goal is proved by manipulating both the sequent and graph representation of the goal until one of the representations give a proof. The goal may be manipulated by a set of inference rules similar to those of RUT or by *statement rules*. A statement rule is derived from a fact of the theory and is also represented in graph and formula parts. Statement rules, like the conjecture, are normal-formed before application. The normal form is similar to that produced by the inference rules of RUT, namely the conclusion is a disjunction of literals, and the hypothesis is a conjunction of either literals or implications.

DATTE does not skolemize the conjecture before the proof is attempted, and so there are inference rules which deal with the quantifiers within the proof. Only the simple quantifier elimination rules exist in this program, ie only those rules which eliminate quantifiers and do

not introduce new constants. Consequently, the conjecture may have universal quantifiers in the hypothesis and existential quantifiers in the conclusion. These are dealt with by examining the graph. An existential conclusion is proved if there are objects in the graph satisfying the conclusion. Universal hypotheses are treated as rewrite rules and converted into statement rules if possible.

Statement rules which contain defined terms which are not binary, and therefore cannot be represented in the graph, are expanded by using the definition of the defined term. Thus statement rules are always expressed in the most primitive terms of the theory, ie. binary relations or primitive (undefined) terms.

The statement rule for the definition of $\subseteq$ is given in 6-2.

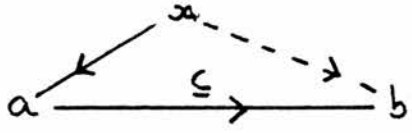

Figure 6-2: Statement Rule for the Definition of $\subseteq$

The rule allows the addition of the dotted arc to the graph if the solid arcs are present. The x and y of the statement rule graph are variables to be bound. The rule may be applied if all of the hypotheses of the rule are true (ie if the left-hand side of the rule matches the hypotheses of the goal). This is exactly rewriting, except that the conjecture is split into two parts, and so is the left hand side of the rule. The output of the rule may, in general, add hypotheses to the conjecture and add arcs to the graph. The goal is proved when there is a hypothesis which matches the conclusion, or when there is an arc in the graph representing the conclusion.

The prover attempts the proof of a conjecture by, first applying the rules of inference of the system which reduce the conjecture to a set of goals in the normal form described above. These are exactly like the rules of inference of RUT and the UT provers. Once this is done, if the conjecture is not proved, the statement rules of the system are applied repeatedly. When no more rules can be applied, and if the conjecture is still not proved, the definition of predicates in the conjecture may be expanded and the process repeated. Once definitions have been expanded, rules which deal with existential and disjunctive hypotheses are used. Both of these rules can be very expensive. The first may require the addition of new objects to the graph, and the second a new copy of the graph for use in a case analysis.

DATTE admits no function symbols which means that the system performs in a decidable subset of first order predicate calculus. In order to represent the union of two sets for example, the system uses the representation: $Union(x,y,z)$, meaning that x is the union of y and z. This leads to a greatly simplified graph, but to unnatural expression of conjectures.

DATTE is not guided in the application of its statement rules, or definitions expressed as rewrite rules. As remarked above, the rewrite rules of the theory are preprocessed by normal forming before the proof commences. One of these normal forming steps involves the expansion of all defined terms. Therefore, after preprocessing, definitions will always cause the goal to be rewritten into a new formula involving only primitive terms. The use of the gazing technique for rules in this form is certain not to produce useful guidance, since all rules will rewrite to a small subset of terms. The approach that is adopted in DATTE is to rewrite the formula to be proved into the primitive terms of the theory and then crunch the resulting set of goals until the proof is found. The rewriting can be performed very quickly by use of subgraph matching algorithms, and proved goals can be similarly detected. DATTE demonstrates that this approach is adequate for theories which do not involve very complex formulae and definitional structures, but the generality of the approach is questionable. As theories become more complex, the amount of rewriting that can be performed increases dramatically. The resulting set of goals grows similarly - in short this approach will succumb to a combinatorial explosion. A final argument against this approach of DATTE is the observation that the proofs that are produced by the program are unlikely to be the simplest possible in all but trivial cases. One of the motivations of Gazing is that the amount of rewriting that is performed should be limited as much as possible.

The graph representation of hypotheses has much in common with Bledsoe's inequality work, as remarked above. In addition Bundy, [Bundy 73] has used a similar technique to control inference in the domain of arithmetic. As remarked by Pastre, the representation of the hypotheses in this way is more efficient, and the graph can be searched quite efficiently. However, the power of the prover is not affected by the choice of this representation.

DATTE has been used to prove theorems in many domains, for example: set theory, function theory, and the theory of orderings.

Reiter's Prover

Reiter's prover [Reiter 76, Reiter 73] is again based on a PROVER style logic, however the system is guided by the use of a model of the semantics of the theory.

Each constant of the theory is mapped to an object in the model. The functions and predicates of the theory are mapped to sets of tuples in the model which indicate the result of applying the function or predicate to those objects. For example, the predicate $=$ would be mapped to a set of pairs in which the first and second objects are equal. The model provides an example which may be referred to in order to verify the plausibility of the conjecture to be proved. The best example of this technique is to consider the drawings of mathematicians proving conjectures in plane geometry. Frequently when proving a conjecture about triangles

a mathematician will draw an arbitrary triangle and verify the plausibility of the conjecture, and subgoals in the diagram. This technique was used in Gelernter's [Gelernter 59] program. Reiter's program is an extension of these ideas to arbitrary domains.

One of the inference rules of the program illustrates the technique. To prove a conjunction, prove the left conjunct. The result is a substitution making this conjunct true. Apply this substitution to the right conjunct, and determine whether the resulting formula is true in the model. If it is not then reject this substitution as a solution for the left conjunct. Otherwise, prove the right conjunct.

Just as the geometer must choose the diagram carefully in order to avoid coincidental properties of the diagram, so must the model be chosen carefully. Indeed the prover is able to update the model in its attempts to refute subgoals. If there is some model which makes the conjecture false, then this model is chosen for the prover to use. Thus not only can the prover use the model to guide the search for the proof, but the model is successively updated in order to refute as many conjectures as is possible.

Reiter's prover does not have the rewriting rule of inference. Definitions and lemmas can therefore only be represented by conjoining them to the hypothesis of the goal to be proved. As discussed in chapter 2 (section 2.4.2), this approach severely limits the power of the prover. This is because no guidance can be made available to the prover to determine whether it is likely to be beneficial to carry out the rewriting step. Since definitions and lemmas occur as implications (equivalences) in the hypothesis of the conjecture, and since all such definitions must appear for the prover to be general, the prover will carry out many redundant steps in the proofs that it performs. The motivation for both the peeking and gazing techniques is that definitions and lemmas, particularly definitions, are too powerful to allow them to be used whenever possible.

It must be stated, of course, that Reiter's goal in devising his prover was to show how the use of the model can guide the search for the proof. The problem of controlling definition instantiation was therefore not of importance to his experiment.

Reiter's prover is an implementation of a sequent based logic which has the added ability to refer to a model of the domain within which the prover is working. The model is used to prune the search for a proof of obviously false subgoals, thus allowing the prover to be more directed in the overall proof. The prover requires that all definitions and lemmas are explicitly conjoined to the hypotheses of the conjecture and therefore no guidance is available, apart from the model, to determine whether the use of such a rule is appropriate.

Brown's Set Theory Prover

A prover written by Brown [Brown 78] has been used to prove theorems in set theory. The prover again has a sequent based logic which is implemented by successive rewriting of the conjecture to be proved. Brown's prover treats both the conclusion and hypothesis of the conjecture as a set of formulae. This is the approach adopted by RUT, and as remarked in chapter 2 differs from PROVER's view of the hypothesis and conclusion as formulae.

Brown's handling of quantified formulae and variable binding is rather novel and should be mentioned at this point. First of all it is important to note that the prover does not skolemize the initial conjecture as RUT and the UT provers do. Universally quantified hypotheses and existentially quantified conclusions are treated specially, effectively causing them to be skolemized on the fly. The advantage of this approach, and the fact that the quantified formula is retained after the "skolemization" step, is that many copies of the same universal hypothesis may be used in the course of a proof. With skolemization, the skolem variable that is introduced may only be bound to one value and thus the hypothesis may only be instantiated once. This is a restriction of natural deduction since universal hypotheses assert that all objects have a particular property. RUT and the UT provers finesse this problem by allowing variables to have more than one binding in certain cases (see page 60 of chapter 2). Under no circumstances is it necessary to a proof to construct two identical instances of the same quantified formula. To avoid this situation Brown's prover attaches a record of the substitutions that have been made to the quantified formula. Before making a further substitution instance the prover will check that this is a new instance rather than a repeat of an old instance.

Brown's prover implements a strategy for guiding the selection for the bindings of variables. Unification of variables is restricted by a *forcing* strategy. The idea of this strategy is to ensure that the binding of a variable makes as much contribution to the proof of the conjecture as possible. All outstanding subgoals are considered in parallel, and when attempting to bind a variable, the binding which makes as many subgoals as possible tautologous is the one that is chosen. This has the effect of reducing the amount of work that the prover has to do, since the binding will complete the proofs of the maximum number of subgoals. Additionally this ensures that each sequent has an effect on the eventual binding. This is a significant restriction since if subgoals are treated in isolation, a binding that makes one goal true, may prevent the solution of another subgoal. If the subgoals share a variable, they should both contribute to the decision of a binding for that variable.

Brown's theorem prover has been used chiefly to prove theorems in the theory of sets, and is designed to be extensible. That is, whenever a conjecture of the theory is proved, the conjecture is considered as a rewrite rule and added to the database. Brown's strategy for

guiding the application of the rewrite rules of the theory is to prioritize the rules by recency. That is, whenever a new rule is introduced to the system this is placed at the beginning of the list of rules to be applied. This approach is also adopted in the Boyer-Moore theorem prover [Boyer & Moore 79]. This simple heuristic has intuitive appeal, particularly when the prover is designed to follow the development of a theory as it is being constructed. Brown's theorem prover was tested by following the development of [Quine 69]. The prover was presented with definitions as they are made, and as each theorem is proved it is added to the theory as a new rewrite rule. Since conjectures are often proved in order to enable the proof of later conjectures (the use of lemmas in the mathematical sense), the idea of working chronologically through the rewrite rules seems sensible.

However, there are problems with this approach. In particular, the theory that is being followed may not be well-behaved, or there may be no theory to follow. In either case, the guidance gained by the chronological approach may be harmful. A further problem is that lemmas may be proved, in order to prove a particular conjecture, which are never useful outside of this proof. In such cases, particularly when the lemma is general, adding the lemma to the theory as a rewrite rule would be harmful. In such cases the technique of gazing could be used to avoid the inappropriate use of the lemma.

The generality of theorems is often indicated in mathematics books by the use of labelling, and sometimes naming, conjectures which are to be referenced later. Making such knowledge available to a theorem prover could enable guidance of the system which would allow the avoidance of inappropriate steps. However the gazing technique references mathematically relevant features of the facts to determine the application of the rules. For these reasons, gazing is seen as a more powerful tool for selecting applicable facts.

6.3.2. Abstraction Spaces

The technique of planning in abstraction spaces is due to Sacerdoti [Sacerdoti 74], who devised two programs, ABSTRIPS and NOAH, which are able to construct plans in a hierarchy of abstraction spaces. While Sacerdoti was interested in the problem of automatic planning, this technique is generally applicable. In particular Plaisted, [Plaisted 80] shows in theory how this technique can be applied to the problem of theorem proving by showing how to define abstraction spaces so that useful guidance of a proof can be gained from the corresponding proof in the abstraction space. Plaisted's work leaves open the question of which of the many abstraction spaces that meet his definition are generally useful, and also the problem of guiding the abstract proof. These questions are addressed by gazing, and by the work of Cvetkovic and Pevac, [Cvetkovic & Pevac 83]. Cvetkovic and Pevac use a gaze graph to guide the selection of facts in a manner reminiscent of gazing, but the work differs

from that reported in this thesis in many important respects. Most noticibly, their planning process is much more expensive, and less likely to be accurate than that of gazing.

In this section I describe the work mentioned in the preceding paragraph. I begin by describing the work of Sacerdoti.

6.3.2.1. Planning in Abstraction Spaces: ABSTRIPS and NOAH

The rules that are used by ABSTRIPS are called *operators*, and correspond to rewrite rules in the gazing system. Operators have *preconditions* which correspond to the input to the rule, and *postconditions*, the output of the rule. ABSTRIPS, [Sacerdoti 74], has a fixed number of abstraction spaces, and each predicate that the program knows about is associated with a unique abstraction space. In this way, some predicates are recorded as inherently detailed goals, while others are seen as inherently high-level goals. The operators of each space differ only in the preconditions: the postconditions of the operator remain the same. In a high-level abstraction space, only the preconditions which belong to that space are examined before the operator is accepted into the plan.

When given a goal to prove ABSTRIPS considers the difference between the current state and the goal state, and then finds an operator which reduces this difference. If the operator had preconditions which were true in this abstraction space, or if the preconditions could be made true, then the operator would be accepted into the plan. Once the plan is completed in this abstraction space the planner moves to the next abstraction space. In the new space the planner reconsiders each of the steps that are used in the plan, checking that the preconditions of this new level are, or can be, met. This process is repeated until the plan is complete in all abstraction spaces.

ABSTRIPS uses the *means-ends analysis* technique of the STRIPS program, [Fikes & Nilsson 71, Fikes *et al* 72] to construct plans within each abstraction space. This differs greatly from the techniques employed in gazing and indicates one of the chief differences between ABSTRIPS and the gazing technique. In the means-end analysis technique, the system compares the current state of the world with the goal state, and chooses an operator which most reduces the differences between these two states. ABSTRIPS determines whether the operator can be applied, if not it is because some preconditions of the operator are not true in the current world. In this case, these preconditions are set up as a new goal, and the system plans to make them true. If the operator can be applied then the world that results from applying the operator becomes the current world, and the planner attempts to plan to reduce the differences between this world and the goal.

In gazing terms, choosing an operator which reduces the most differences is equivalent to

choosing the rewrite rule which produces the currencies which are closest together. Notice however, that ABSTRIPS performs only a single-step look ahead. It chooses the single operator which reduces the difference the most, regardless of the fact the the goal world may not be producible from the new world, or that the solution may take many more steps than are required if a different operator had been chosen. In gazing this situation is avoided by considering, for example in the predicate space, all paths through the gaze graph which link the current sequent and the goal, and then choosing to consider the shortest. Guidance of this type is not available in ABSTRIPS.

In ABSTRIPS the abstraction space to which a predicate belongs is called the *criticality* of that predicate. This notion is very similar to that of criticality as defined in chapter 4, since in both cases it is the predicate of highest criticality which keys the use of the operator. At this point however the similarity stops. Consider rewrite rule a (the superscripts indicate the criticality of the predicates).

a. $\neg\, x = {}^1y \wedge x \subseteq {}^2y \Rightarrow x \subset y$

b. $x \subseteq {}^2y$

c. $\{\subseteq, =\}$

In ABSTRIPS abstraction space 2 the only precondition that has to be met is that $x \subseteq y$, so the preconditions in this space are b. Once this has been verified, the operator is deemed usable, and only in the later abstraction spaces is the other precondition considered. For the gazing planner the preconditions are as in c. The step is considered provided the predicate $\subseteq$ is a hypothesis, but accepted only if the predicate $=$ is also present.

Notice that the ABSTRIPS planner has to deal with the *arguments* of the predicate with highest criticality, where gazing deals with these only in the less abstract function/polarity space. The ABSTRIPS planner works through the formula in a depth-first manner, considering all of the detail of each literal before moving on to the next. The gazing planner searches the formula in a breadth-first manner, considering only the predicates at the first pass, then the term structure in later abstraction spaces.

Like the gazing planner, ABSTRIPS completes the plan at each abstraction level before going on to refine the plan at a less abstract level. This is clearly a good decision, and must be adopted by any abstraction based planner since, if the plan cannot be completed at any level then replanning must occur at the previous level. The whole idea of using abstraction spaces is not to consider the detail until the system is sure that it is on the right track. This information can only come from completing the plan at some level.

The major differences between NOAH and ABSTRIPS are:

1. NOAH attempts to leave the order in which the planned actions are to be carried out as unconstrained as possible, enforcing an order only when necessary.

2. NOAH has no fixed abstraction spaces - they are determined dynamically, and,

3. NOAH allows for the patching of plans, rather than blindly backtracking.,

The first of these points does not concern us here, since there is no analogous problem in the gazing technique. Since each rewrite rule is planned to rewrite a subformula introduced by the previous application of a rewrite rule, the plan is necessarily ordered. Of course, one must choose whether to rewrite the hypothesis or conclusion first, but this is no importance, the choice cannot effect the success or failure of the gazing step.

In NOAH the abstraction spaces depend on the goal that is to be proved, and the definitions of the operators which are available to solve given goals. This is achieved by associating with each goal a number of operators which may be used to solve that goal. These are the only operators which are used to attempt to achieve the goal, and so the operators themselves define the abstraction spaces. Thus, for example, if the initial goal is very low-level, then the operators which may be used to achieve it will be a small subset of the total number of available operators, namely those which deal with what were previously considered to be "details". Thus planning in NOAH begins in a comparatively detailed abstraction space, where in ABSTRIPS the more abstract spaces would have been considered although the plans that they would have produced are trivial.

NOAH, like gazing, allows for the patching of a plan rather than blindly backtracking and replanning as performed by ABSTRIPS. If a plan produced by ABSTRIPS fails at some abstraction level, then the planner undoes the work and attempts to make some other plan at the previous level. No record of the reason for failure is noted and hence the replanning attempt cannot be guided by this failure. In the cases of NOAH and gazing the reason for failure is recorded and this information is used to ensure that the minimum of redundant work is recreated. This is achieved by recording along with each step of the plan, the reasons for carrying out this step. When the application of a step is found to be impossible, replanning can begin by replacing this step with another with the same effect, leaving the rest of the plan as intact as possible.

The language which is used by both ABSTRIPS and NOAH to represent goals, and world states is much more restrictive than that used in gazing in two respects:

- ABSTRIPS and NOAH do not allow function symbols, and,
- only conjunctions of facts are allowed.

These facts, when taken together, show that ABSTRIPS and NOAH have no need for a function/polarity space, such as that used by gazing. Since there is only conjunction no

polarity considerations arise, and there are no functions to appear in the f/p triples. This representation is quite adequate for the robot planning domain that ABSTRIPS and NOAH work in, since for example, the state of the world may be represented as a conjunction of facts like *In-Room*(*Robot*,*Room*1). The negation of a fact is represented by its absence from the world description. At any one time the state of the world is assumed to be known, and so no need for disjunction arises. The system does have *laws* which are of more complex form, for example *Connects*(*Door*2,*Room*4,*Room*3) $\Leftrightarrow$ *Connects*(*Door*2,*Room*3,*Room*4), but these are used only to check preconditions, and not as part of the world state. In these systems, therefore, the abstraction spaces are all copies of each other, the only thing that changes from one abstraction space to the next is the number of predicates that are considered by the planner. This is in contrast to gazing, where the predicate and function/polarity abstraction spaces are quite different.

6.3.2.2. Abstraction Mappings in Theorem Proving: Input Abstraction and Generalization

Gazing consists of two distinct parts: the use of abstraction mappings to simplify the problem to be solved, and the use of the common currency model to guide the search for the abstract proof. The work of Plaisted, [Plaisted 80, Plaisted 86], on abstraction mappings is closely related to the first part of gazing in that abstraction spaces are used to create abstract proofs which are then used to guide the construction of a complete proof.

In [Plaisted 80] Plaisted describes an approach which he later calls *input abstraction*. In this technique, each member of the set of clauses to be refuted is mapped to a set of abstract clauses by some abstraction mapping. A refutation of these abstract clauses is then sought, and if found the prover attempts to pull this proof back into the full space. The *generalization* technique described in [Plaisted 86] is Plaisted's attempt to remedy some of the shortcomings of the input abstraction technique. Here the idea is that each input clause is abstracted, and then resolutions performed between these clauses. The resolvent is then mapped under the same abstraction before being added to the set of clauses for further resolution. This technique allows control of aspects of the search space which are not accessible to input abstraction. Of these techniques, only input abstraction is directly related to gazing.

Both input abstraction and generalization are cast in the context of the resolution inference rule which is used in both the full and abstract spaces. Plaisted asserts, however, that the techniques are applicable to other logics and inference rules. The main idea of Plaisted's work is to formalize the conditions under which it is possible to use abstracted proofs to guide the search for a proof in the problem space, and to give a strategy for recovering the full proof from the abstract one.

The definition of input abstraction given in [Plaisted 80] is:

> **Definition 62:** An (ordinary) abstraction is an association of a set $f(C)$ of clauses with each clause C, such that f has the following properties:
>
> 1. If clause $C3$ is a resolvent of $C1$ and $C2$ and $D3 \in f(C3)$ then there exist $D1 \in f(C1)$ and $D2 \in f(D2)$ such that some resolvent of $D1$ and $D2$ subsumes $D3$.
>
> 2. $f(NIL)=\{ NIL \}$. (NIL is the empty clause.)
>
> 3. If $C1$ subsumes $C2$, then for every abstraction $D2$ of $C2$ there is an abstraction $D1$ of $C1$ such that $D1$ subsumes $D2$.

Notice that each clause maps to *a set of* abstract clauses. This introduces a tension into Plaisted's technique: on the one hand the clauses that are produced by an abstraction mapping are supposedly simpler in some respect than the original clauses, but in the abstraction space there may be many more clauses then there were originally. There seems no guarantee that the search for a proof in the abstraction space will be any simpler than in the original space. In gazing this tension does not exist, the abstraction spaces are both decidable, and thus very much simpler than the full problem space. In gazing each formula abstracts to at most one abstraction, some formulae disappear completely in abstraction spaces since their input and output sets are the same and they are classified as null lemmas. Plaisted points out that the input abstraction technique may be used to construct a hierarchy of abstraction spaces as in ABSTRIPS and NOAH. This is one approach to ensuring that the abstraction spaces are eventually simpler than the original space, however if at each level the number of clauses increases then this may compound the problem.

The need to be able to produce a set of abstract clauses from a single input clause appears to derive from the fact that resolution is used as the inference rule in both the original and abstract space. Consider, for example, constructing the analogue of the predicate abstraction mapping on clauses. This mapping "forgets" all detail of the input formula except the predicates which appear in the formula. Thus the two clauses (4) and (5) both map to the set (6).

$$P(x) \vee Q(a,y) \vee R(f(a),g(b)) \tag{4}$$

$$\neg (P(x)) \vee Q(a,y) \vee \neg (R(f(a),g(b))) \tag{5}$$

$$\{ P, Q, R \} \tag{6}$$

Under this abstraction mapping any clause containing only literals in the predicates P, Q and R will map to the set (6), and inferences between any pair of such formulae will be permitted in the predicate abstraction space. To allow the same inferences Plaisted has to map each clause first into a clause which has the same literals, but with the arguments "forgotten", and then into the set with all permutations of negations included: (7).

$$
\begin{array}{ll}
\{P \vee Q \vee R, & P \vee Q \vee \neg R, \\
P \vee \neg Q \vee R, & P \vee \neg Q \vee \neg R, \\
\neg P \vee \neg Q \vee R, & \neg P \vee \neg Q \vee \neg R, \\
\neg P \vee \neg Q \vee R, & \neg P \vee \neg Q \vee \neg R\}
\end{array} \tag{7}
$$

This is the smallest set of clauses which allows all of the inferences permitted in the gazing abstraction space. Notice that using this mapping a clause containing L atomic subformulae is mapped to a set of 2^L clauses. Resolution steps which ignore the polarity of atoms in the original formula may be performed by choosing the abstract formula which allows the inference. Carrying out such a step clearly involves searching among the formulae in the abstraction set.

This difference between gazing and Plaisted's formulation can be summed up by saying that Plaisted holds the inference rule fixed while abstracting to a set of clauses which will allow more inferences, while gazing produces a representative abstract formula but uses a different inference rule to produce the same effect. The amount of search required by gazing is therefore less than that carried out by Plaisted's system, implementing the same abstraction space.

The generality of Plaisted's work means that this is not a fatal flaw, only that there are some abstraction spaces, in particular those used by gazing, which cannot be implemented cheaply in his framework. Some abstractions that Plaisted uses may produce only singleton sets of formulae in the abstract space, thus no more search need be carried out in the abstraction space. Careful selection of abstraction mappings is required to ensure that the technique will be successful. For each abstraction that it is possible to define within Plaisted's technique, however, the problem of predicting how it will behave is going to arise. Even when singleton sets are produced in the abstraction space the abstract search problem may remain undecidable. In [Plaisted 80] Plaisted cites many examples of abstraction mappings, for example "forgetting" the nth argument to a predicate, which produce an undecidable abstraction space . Plaisted suggests that different abstraction mappings might be used to create a hierarchy of abstraction spaces in a manner reminiscent of ABSTRIPS. The problem of expanding the number of clauses might become a problem if this approach were adopted, but after some number of abstraction steps, the clauses that are produced would be very much simplified.

The first clause of Plaisted's definition of abstraction insists that if an inference can be performed between two full clauses, then some analogous inference can be found in the abstraction space between the abstractions of these clauses. The abstraction to the predicate space used by gazing also has this property. In the case of gazing this says that if a formula may be rewritten by a rewrite rule, the abstraction of that rewrite rule may be applied to the

abstraction of the formula. The function/polarity abstraction mapping does not have this property. This is because to perform an inference in the f/p abstraction space we insist that the triples contain the same function symbols, and this is not necessary in the full space, thus some inferences in the full space will not be acceptable in the abstraction space.

Plaisted shows that his abstraction mappings preserve the ability to perform proofs. That is, if there is a proof of a set of clauses, then there will be a proof of the set of clauses obtained by mapping them under an abstraction mapping. This is simply an extension of the result concerning inferences, and as such can be seen to hold for the predicate abstraction space of gazing, but not necessarily for the function/polarity space. Further, in Plaisted's system the abstract proof is of a similar form to the full proof. The analogy is well defined; the significant property being that a resolution in the abstraction space corresponds to a sequence of resolutions in the full space.

The main point of [Plaisted 80] is to give a strategy for guiding the search for a proof in the problem space when given an abstract proof. Basically the idea is to use the analogy that I have already described to guide the inferences in the full space. The clauses in the abstraction space are mapped back to their full space counterparts, and inferences between these clauses is attempted. When inferences which correspond to those in the abstract proof can be found, a part of the full proof has been found. This task is non-trivial for two reasons:

- Many clauses in the full space may abstract to the same clause in the abstract space.
- Some subgoals may be inherited in the full proof which were not dealt with in the abstraction space, since in the abstract proof some of the detail has, by definition, been forgotten.

Neither of these problems may be solved by guidance from the abstract proof produced by Plaisted's method. Where many clauses are mapped to the same abstract clause, the theorem prover will be forced to choose the inference which is analogous to that in the abstraction space. In the case of an incorrect choice it is to be hoped that the analogy between the abstract proof and the full proof attempt breaks down quickly. This problem arises in gazing too, when two or more formulae share the same abstraction. In gazing, the plan is made on the basis of the abstraction's effect, and if there is more than one formula with the same effect then they are equally suited to bring about the desired effect. If in a less abstract space the plan breaks down, then there is a record of alternative ways of achieving the same effect.

Input abstraction allows unconsidered subgoals to arise in the full space, since the definition

of abstraction allows that literals from the full clause may be completely absent from the abstraction. When such subgoals are derived in the full space they may be proved by a second application of the abstraction technique using a different abstraction mapping. Alternatively, it may be judged that such subgoals may be proved by a brute-force such as exhaustive search. The course taken will depend on the precise nature of the abstraction used in the constructing the abstract proof. This problem cannot arise in gazing, since every literal is represented by its abstraction in the abstraction spaces. When unification is attempted between the conjecture and a fact the gazing system is guaranteed that a pair of literals with matching f/p triples are present in the formula. It is necessary to worry about the differences between these two literals which are not represented in the abstraction space when the unification is attempted. There are two possible ways of reducing these differences suggested by gazing, the first is the use of "null" steps, if the f/p triples are the same, the second is to introduce additional steps to take care of such differences. The minimal guarantee offered by the abstraction space is that there will be at least one pair of literals which are candidates for unification.

A significant difference between Plaisted's technique and gazing is that the use of input abstractions does not provide guidance in the search for the abstract proof. That is, once the abstraction has been performed the system has the same means of solving the problem that are available to the prover as a whole. The idea is that the abstraction space should be simple enough to limit the potential search for the abstract proof. As observed above, this is a questionable assumption, since there is no guarantee that the abstraction space will be any simpler than the original. In gazing the search for the abstract proof is guided by constraints on the rules that will be considered for use in the proof. For example, the gaze graph is used to enable the prover to choose the shortest sequence of rewriting steps which will lead to the occurrence of complementary predicates in the goal to be proved. A trivial example of how the use of the gaze graph limits the possible inferences, is that the search strategy does not allow looping in the plan. An exhaustive search of the abstraction space is infeasible if the abstraction space can contain loops, as the predicate abstraction space can. An unguided prover has to use trial-and-error to find an appropriate sequence of inferences. Even assuming that the prover has enough resources to explore all possible sequences of rewriting steps, this approach is wasteful when some guidance might be provided. Other examples of this type of guidance provided by gazing are the heuristics presented on page 93. These heuristics eliminate many facts from the prover's consideration when constructing abstract proofs, while they would be considered in by a prover implementing Plaisted's techniques. There is nothing inherent in Plaisted's work which would prevent a prover from being given this additional guidance.

In [Plaisted 86] a different approach to using abstraction mappings is presented. Plaisted calls this approach *generalization*, and it differs from input abstraction in that all resolvants are mapped under a generalization mapping when they are produced, before they are added to the clause set. This allows much finer control over the search for a proof. In particular, Plaisted uses generalization to prevent the clauses present in the proof becoming too complicated. For example, it is possible using generalization, to specify that no terms of depth greater than n, for some n, will appear in the proof by ensuring that the generalization mapping replaces such terms by shallower terms. This is not possible using input abstractions, since the abstraction mapping only works on the input clauses, and resolution does not respect clauses abstracted in such a way. That is, even if the abstraction replaces all terms of depth greater that n with a term of depth exactly n, the application of resolution may again produce a clause of depth greater than n. Thus in generalization all clauses produced are generalized, not just those which are in the input set.

The technique of generalization, while an interesting use of abstraction in theorem proving, is not so directly related to gazing as input abstractions. Generalization is only really necessary because resolution does not respect the effect of the abstractions. That is, resolving on two generalized clauses does not necessarily produce a generalized clause. The gazing abstraction spaces, and the abstraction spaces defined by Plaisted's input abstractions, are closed in this way. No inference in either system can produce a formula that does not belong to the abstraction space. Because of this property of generalization, the technique is qualitatively different from gazing and input abstractions.

In summary, Plaisted's work on abstraction mappings, particularly the work on input abstractions, is closely related to gazing, but differs in many respects. The most important of these are:

- Neither of Plaisted's techniques guide the search for the abstract proof, and,
- Neither of Plaisted's technique guarantee that the abstract space will be any simpler than the original problem space.

The justification for the lack of guidance of the abstract proof is that the abstraction mapping will abstract sufficiently to allow the proof to be found without requiring guidance. I have observed above, that this assumption is questionable, but that careful selection of abstraction mappings can significantly lessen the complexity of proofs. Of course, there is nothing in Plaisted's technique to prevent such guidance being used, but the technique itself does not require such heuristics to be used. In gazing the analysis of the effects of the abstracted rules are used to guide the choice of rules to be used in the proof, and therefore the course of the proof.

The fact that there is no guarantee that the abstraction space is simpler than the original space is somewhat worrying, but is partly a consequence of the fact that the same inference rule is used in both the abstract and full spaces. This means that certain abstractions can only be caried out by mapping single clauses into sets of clauses, thus multiplying the number of possible inferences. Also, the abstraction mapping may define an abstraction space which is undecidable, and although the clauses are "simpler" in the abstraction space the amount of search required to find the proof may still be prohibitive. Indeed, as pointed out by Alan Bundy, the fact that clauses are simpler means that there are more candidate unifications. For example, $P(x)$ unifies with more clauses than $P(f(g(x)))$. Thus in Plaisted's techniques search increases because of the increased number of clauses, and the larger number of candidate resolutions.

6.3.2.3. Abstraction Mappings in Theorem Proving: The GRAPH Theorem Prover

Cvetkovic and Pevac [Cvetkovic & Pevac 83] use a structure which is very similar to a gaze graph to guide the selection of facts to be applied to a conjecture. While similar to gazing, their technique differs in many important respects. However the independent discovery of the graph representation of a theory lends credence to the belief that it is a powerful tool for guiding the proof of theorems.

Cvetkovic and Pevac have implemented a graph theoretician's aid. It consists of three parts: bibliography, algorithms and theorem provers. The system has two theorem provers, either of which may be called by the other. The interactive prover works by a natural deduction style logic, while a stand-alone prover uses resolution. Only the interactive natural deduction theorem prover is of interest to us here.

The interactive theorem prover, GRAPH, has the ability to use facts of graph theory to transform the current conjecture. The theorem prover has a representation of all of the predicates of the theory in a graph structure. While they do not call it this, this structure is the gaze graph of the theory. One slight difference in their representation is that for lemmas which are equivalences, arcs are added to the graph in both directions, but one set of arcs is labelled "blue" and the other "red". Red arcs correspond to using the conjecture in one direction and blue to the other. Thus for every red arc in the graph, there is a corresponding blue arc linking the same nodes, but in the opposite direction. This is contrary to the orientation of lemmas in one direction in the gazing technique. Arcs which derive from definitions are uncoloured and are oriented away from the defined term, as in the gaze graph.

I begin by giving some of the definitions from [Cvetkovic & Pevac 83].

> **Definition 63:** A *path* through the graph is a path which contains only uncoloured and "blue" arcs.

Definition 64: An *antipath* is a path which consists of only uncoloured and "red" arcs.

Definition 65: A *trace* from a predicate A to a predicate B is a path from A to some predicate C, followed by an antipath from C to B.

Definition 66: The *complexity of a predicate* appearing in the conclusion of a sequent is the length of the shortest trace joining that predicate to some predicate in the hypothesis.

Definition 67: The *complexity of a sequent* is the average of the complexities of the predicates of the conclusion.

The notions defined above correspond very closely to those used in the gazing technique. In gazing there is no distinction between paths and antipaths since all rules are oriented to decrease criticality. In gazing, the notion which corresponds to a trace is a pair of paths (rather than a path and an antipath), but this is a rather slight difference. The complexity of a predicate in the conclusion of a sequent is the sum of the lengths of the shortest pair of paths in the gaze graph rewriting the conclusion predicate and the hypothesis predicate to a common predicate. While this notion is not used explicitly in gazing, when there is a choice of plan, the shortest is chosen.

If the original sequent is transformed by the use of a fact to a less complex sequent, then this step is recommended by GRAPH. In GRAPH the strategy is to find those facts α_i which are recommended in this way and then to chose at most two or three such transformations.

For example, if the sequent is (R), and the facts (v), (vi) and (vi) were known, then the path consisting of only (vi) would rewrite $=$ to $\subseteq$.

$$a = b \vdash^{?} a \subseteq b \qquad (R)$$

$$x = y \Rightarrow \forall z.\, z \in x \leftrightarrow z \in y \qquad (v)$$

$$x = y \Rightarrow x \subseteq y \wedge y \subseteq x \qquad (vi)$$

$$x \subseteq y \Rightarrow \forall z.\, z \in x \rightarrow z \in y \qquad (vi)$$

The path (v) followed by (i) would also introduce a common predicate, but the length of this trace is 2, and the length of the first trace is 1, so the complexity of $\subseteq$ is 1 in this sequent. Notice how similar this is to gazing in the predicate abstraction space.

Once the candidate rewrite rules (α_i) have been found in GRAPH these facts are used to rewrite the sequent (in parallel), resulting in sequents σ_i. Clearly each of the facts (v), (vi) and (i) may be applied to the sequent, and are thus candidates for α_i. Suppose however that these three rules are the chosen α_i. Each rule is used to rewrite the sequent to σ_i. Notice this is very different from gazing, in that a number of candidates are each considered in parallel, whereas in gazing only the most promising candidate proceeds for further analysis.

$$x \in a \wedge x \in b \vdash^{?} a \subseteq b \qquad (MM)$$

$$a \subseteq b \wedge b \subseteq a \vdash^{?} a \subseteq b \qquad (NN)$$

$$a = b \vdash^{?} x \in a \rightarrow x \in b \qquad (OO)$$

The σ_i are then passed to the logical inference part of the prover, which returns, for each sequent, a set of subgoals τ_i to be proved. For example, when given (MM) the system might return (PP), and (QQ).

$$x \in a \vdash^{?} a \subseteq b \qquad (PP)$$

$$x \in b \vdash^{?} a \subseteq b \qquad (QQ)$$

The complexity of each sequent from τ_i is measured and these measures averaged to get a measure of the complexity of the proof that would have to be performed if the rewriting step α_i were used. The α_i which results in the minimum for this measure is chosen as the rewrite to be performed. In the example above, the rule (vi) would be selected since the complexity of the resulting sequents is 0, indicating that the same predicate appears in both hypothesis and conclusion.

The application of this technique clearly involves a lot of work. The complexity of a sequent may be measured quite quickly by using fast graph searching algorithms, but a number of α_i are to be applied, and logical inference is carried out on each of the resulting sequents. Only logical inference is allowed here, so the sequent must eventually be proved, or reduced by logic to a set of outstanding subgoals, however this step could be arbitrarily complex. Once this is done, choosing the appropriate step can be performed quite quickly.

Since choosing the appropriate step involves partially building the proof of the resulting sequent. The work that is done for the step that is eventually chosen is not wasted. But unless the prover makes a wrong decision and remakes this choice, the work performed on the steps which were not chosen is wasted. It is precisely this work that techniques providing guidance are supposed to avoid, so the use of this technique comes into question.

It would be acceptable to carry out such wasted work if the step that was eventually chosen is very likely to lead to a proof. However, in [Cvetkovic & Pevac 83] the authors report:

> Such a criterion prevents useless definition instantiations in the hypothesis A of $A \Rightarrow B$* . On the other hand, simple examples show that the definition instantiations in B, although promising at the beginning, can lead to a shortcoming after some steps, simply because the predecessors of elements of $L(B)$ [the predicates of B] do not have traces of attack any more or have very long traces.

---

*Cvetkovic and Pevac's notation for $A \vdash^{?} B$

These remarks are similar in spirit to the critique of Bledsoe's peeking technique in chapter 4 (section 4.2.3). This is not a coincidence, Cvetkovic and Pevac's technique suffers from the same problems as peeking because both techniques only plan one step ahead. It was this shortcoming that gazing was designed to overcome.

In their attempt to overcome this problem Cvetkovic and Pevac define an improved complexity measure for predicates.

> **Definition 68:** The *direct predecessors* of a predicate P are those predicates which occur in the definition of P.
>
> **Definition 69:** The *improved complexity* of a predicate, P, of the conclusion of a sequent is defined to be one larger than the average of the (simple) complexities of the direct predecessors of P.

However it is clear that this only takes the problem one step further away since this measure only considers those predicates which are direct predecessors of P.

Another similarity with peeking is evident in Cvetkovic and Pevac's work. Their technique, like peeking, does not consider the functions which are present in the sequent in the complexity measure. This was discussed in the case of peeking in chapter 4 (section 4.2.2). Clearly the same problems would occur in this technique. This is not a problem for Cvetkovic and Pevac since the logic that GRAPH deals with does not admit of function symbols.

The major difference between peeking and gazing is the use of the gaze graph to represent the entire theory. This leads quickly to the notion of complexity of sequents, and to the ability to determine steps which decrease the differences between the hypothesis and conclusion. Peeking does not have this ability: it is an "all or nothing" rule. If the use of the definition does not immediately introduce a predicate which already appears in the sequent, peeking does not perform the rewriting step.

The complexity of a sequent, as defined above, is also used to guide the logical inference of GRAPH to a limited extent. There are two cases:

- If the prover is to show $A \vdash^? B \vee C$ and the complexity of $A \vdash^? B$ is much greater than that of $A \vdash^? C$ then the prover tries to show $A \wedge \neg B \vdash^? C$*.
- If the prover is trying to show $A \wedge B \vdash^? C$ and there are no traces from the predicates of C which end in a predicate of B, then the prover will try to show $A \vdash^? C$.

---

* [Cvetkovic & Pevac 83] gives the new sequent as $A \wedge \neg C \vdash^? B$ but this is surely an error.

In summary, the techniques used in the GRAPH theorem prover are very much in the same spirit as peeking and gazing. The major differences between gazing and the GRAPH techniques are:

Gazing selects a possible rewriting of the conjecture and explores this to the exclusion of all others until either it has been successfully applied, or it has failed and some other possible rewriting must be selected. The GRAPH prover chooses a number of possible courses of action and carries them all out before determining which is the most promising.

The exploration of the possible rewritings is performed by gazing completely in abstraction spaces designed to measure the promise of a sequence of rewriting steps. In the GRAPH prover, before a particular sequence of rewriting steps is chosen all the candidate rewriting steps are applied to the conjecture, thus incurring the penalty of performing at least some of the work that the techniques are designed to avoid.

These two points, that the GRAPH prover carries out much more expensive work on a number of possible options means that the technique is much more expensive than gazing. Further, since it does not take into account the functions in the conjecture and looks only one step ahead it is subject to the same drawback as peeking.

6.3.3. Theory Resolution

In this section I describe the work of Stickel [Stickel 85] on a technique called *theory resolution*. This work is related to the technique of gazing in that inferences are separated into two classes: inferences within the theory, and logical inferences.

Theory resolution is a single inference rule. Like resolution the inference rule works on clauses, which are a disjunction of literals. The theory resolution rule applies to a set of clauses, $C_1, \ldots C_m$ which are each decomposable into the theory literals K_i and other literals L_i: ie $C_i = K_i \vee L_i$ for all i. Suppose there are unit clauses $R_1, \ldots R_n$ such that $\{K_1, \ldots, K_m, R_1, \ldots R_n\}$ is unsatisfiable within the theory, T. Then the T-resolvent of the C_i is $L_1 \vee \ldots \vee L_m \vee \neg R_1 \vee \ldots \vee \neg R_n$.

The key idea of theory resolution is that it is beneficial to treat inferences with the theory separately from inferences within the conjecture. This is exactly the approach that I have taken in the gazing technique. However a contrast between gazing and theory resolution is in the relationship between the theory and logical inferences. In gazing, theory inferences and logical inferences are separated. The proof is attempted without using the theory, and only if this proof fails are theory inferences attempted. Once these have been carried out the system

reverts to carrying out only logical inference. Theory resolution is a single operation which combines theory and logical inference, but the mechanism by which the theory inferences are carried out, and that by which the logical inferences are performed are different. Stickel refers to the component that carries out theory inferences as the *TBox*, and that which carries out the logical inference as the *ABox* (*A*ssertional).

One advantage of the separation of theory and logical inferences is that each component need not know how the other performs, only the effect of the other component. Notice that the theory inference rule does not specify how the prover is to show that the set of clauses is T-unsatisfiable. The TBox must merely determine whether a set of assertions is unsatisfiable within the theory. The ABox need not know how this is done, only that it has been done. This is the case with gazing too: the particular search strategy that has been used to carry out the theory inferences is not important to the logical component. Clearly the task of determining the T-unsatisfiability of a set of clauses depends on the theory T. Any special purpose mechanism which can determine unsatisfiability within the theory may be used.

PROVER embodies the natural deduction equivalent of theory resolution. The decision procedure for inequalities reported in [Bledsoe *et al* 79], and in chapter 2 can be considered as the TBox, and the natural deduction inference rules as the ABox. Rather than the residue of a TBox inference being rejoined to the conclusion of the inference, as the R_i literals are in theory resolution, in PROVER the residue is represented by the typelist.

In the natural deduction framework any number of TBoxes may be specified simply by adding extra inference rules which fire an appropriate decision procedure if the conjecture is of the correct form. It is a weakness of the uniformity of the resolution rule which has led to the discovery of the power of specific TBoxes much later than within the natural deduction framework. This is wholly in accord with the intuition expressed when justifying the study of natural deduction theorem provers (in chapter 1), namely that the ability to devise and express heuristics within this framework far outweighs the disadvantage of the inefficiency of the underlying logic.

In this subsection I have described the theory resolution inference rule. Like gazing, theory resolution allows the separation of inferences into those within the theory, and logical inference. This distinction has been used to construct an inference rule which has two components, one for each type of inference, and allows special purpose techniques to be used to carry out the theory inferences. This approach is exactly that adopted by gazing and earlier by Bledsoe in his work on inequality reasoning. The approach of theory resolution differs from that of gazing in that a prover implementing gazing attempts to prove a conjecture by using only logical inference initially. If this attempt fails the system uses

theory inference to transform the conjecture into one which can be proved by logical inference alone. Thus the theory is ignored completely for some of the proving process. Theory resolution by contrast uses a single inference rule which has two components, one for theory and one for logical inference. Thus the theory may be used at each application of the rule. The relationship between theory resolution and the natural deduction techniques is clearly seen when considering PROVER, which implements exactly the theory resolution technique but within a natural deduction framework.

6.3.4. Summary

In this section I have described work that is related to that reported in this thesis. This work falls into three categories: natural deduction theorem provers, the use of abstractions to produce plans and guide theorem provers, and the separation of inference into logical and database categories.

Other Natural Deduction Provers

In section 6.3.1 I described a number of natural deduction provers. In particular I concentrated on their handling of definitions and lemmas.

Reiter's prover [Reiter 76, Reiter 73] explicitly conjoins definitions and lemmas to the hypothesis of conjectures. This is necessary since the prover has no ability to use the rewriting rule of inference. As observed in chapter 2 this severely limits the power of a prover in any significantly complex domain. Since Reiter's main aim in the construction of his system was to show how the search for a proof can be guided by a model, this drawback was peripheral to his experiment.

Brown's prover [Brown 78], uses a technique for choosing rewrite rules which is also used by the Boyer-Moore theorem prover [Boyer & Moore 79]. The rewrite rules in these provers are stored ordered by recency: the rules at the beginning of the list being the most recently learned by the prover. The prover attempts to apply each rewrite rule, starting from the beginning of the list and working down, when a rule is found which applies it is then used to rewrite the goal. The heuristic is simple, and appropriate for these theorem provers. In both cases the user of the theorem prover is to develop a theory (or proof) incrementally by proving conjectures. Since the sequence of conjectures is supposed to mirror the development of some theory, then the idea that recently proved facts will be more useful is a good heuristic. The gazing technique does not make such assumptions, and as a result, prefers to choose rewrite rules on the basis of their effect on the conjecture to be proved. Recency could however be used to tie-break for a gazing system, although this is not implemented as part of VOYEUR.

Pastre's prover, DATTE, uses a graph to record certain of the hypotheses of the conjecture. The deduction of new hypotheses results in updating the graph and adding hypotheses. This technique allows fast graph matching algorithms to be used to apply rewrite rules, but does not provide guidance as to the choice of rewrite rules to apply. The rewriting of conjectures to their primitive terms, and the solution of the resulting goals has be demonstrated as a plausible method for proving theorems by DATTE provided that theories are not too complex. The generality of this method is somewhat in doubt.

Abstraction in Theorem Proving

Gazing consists of two parts: the first is the idea of using abstractions to create a simpler task to prove, and then using the simpler solution to guide the solution of the harder task. This idea is due to Sacerdoti [Sacerdoti 74], although he originally applied it in the domain of planning systems. In this work the goal is to produce a plan to carry out a specified action. The planner works in a hierarchy of abstraction spaces planning to carry out the goal first in broad generality and then by successively refining the plan to take more details into account. This is achieved by representing the operators in a number of abstract ways. Each level of abstraction has associated with it a number of predicates. A predicate is invisible below the level of abstraction associated with that predicate, thus the fact that the predicate records a precondition that must be true before the operator may apply is ignored until the planner gets to this level of detail. The implementation of gazing differs from this in that the abstractions do not allow whole subformulae to be ignored, rather some detail of the formula, for example the arguments to predicates, is dropped globally. Thus ABSTRIPS considers the operators as having numbers of preconditions, while gazing considers the operators a having increasingly detailed preconditions.

Plaisted [Plaisted 80] applied abstraction to theorem proving, and showed that it is possible to define abstraction mappings to preserve the property of proof. In gazing this idea has been extended further by using specific abstraction mappings which allow guidance of the abstract proof. These abstraction mappings record the concepts that appear in the rewrite rules of the theory, and the effect that applying such a rule has on the concepts. This leads to the possibility of, first of all, guiding the search for the abstract proof, and secondly patching the abstract proof elegantly if it fails when applied in the full space. Gazing also uses a more liberal notion of abstraction mapping, allowing the system to ignore the connectives of the formula in the abstract space.

The graph theory prover, GRAPH, of Cvetkovic and Pevac uses a representation of the theory which is, with minor modification, the same as the gaze graph. This graph is used to guide the search for a proof by determining lemmas and definitions which it is appropriate to use in the proof. GRAPH has a notion of complexity of a sequent which corresponds to the

length of paths in the graph which join the hypothesis to the conclusion. After exploring some alternatives, GRAPH chooses the course of action which maximally decreases the complexity of the sequent. This technique involves expending a great deal of computational resources in determining the appropriate rule to apply, in particular requiring the application of more than one of the alternative rewrite rules. This would be acceptable if the heuristic were more reliable, but the heuristic suffers from the same drawbacks of peeking, namely that the heuristic embodies only a one-step look-ahead and does not consider function symbols in the complexity measure. The second of these disadvantages is not a problem for GRAPH, since its logic does not admit function symbols, but the first can be a problem. Both factors limit the general usefulness of the technique.

The Separation of Logical and Database Inference

Stickel's work on theory resolution [Stickel 85] is related to gazing and to Bledsoe's inequality reasoning techniques in that it separates inferences relative to a theory from logical steps. Inferences which are purely logical are performed by the resolution component of the inference mechanism, and the theory deductions are carried out by a separate, unspecified, TBox. The gazing technique, in addition to making the distinction between the logical and theory deductions, proposes the use of abstraction spaces for guiding the theory deductions. An implementation of gazing therefore provides the theorem prover with a TBox.

6.4. Conclusions

In this thesis I have described a technique, gazing, which enables the control of the use of rewrite rules in theorem proving (chapter 4). Gazing was described within a natural deduction framework (chapter 1) and was motivated by some problems which were found with the guidance techniques suggested by Bledsoe ([Bledsoe & Tyson 75a, Bledsoe & Tyson 78] and chapter 2). In chapter 5 I justified the use of the technique by showing that it is not expensive to apply, and can behave no worse than an unguided prover. I have implemented VOYEUR, a theorem proving system which can operate in two modes. In one mode the theorem prover emulates RUT (chapter 3), a rational reconstruction of the UT theorem provers (chapter 2). In the other mode, the theorem prover replaces some of the RUT inference rules with the single gazing rule, which applies the gazing technique to the conjecture. A comparison of proofs performed by VOYEUR in the two modes is presented in appendix B.

6.4.1. What Has Been Learned

In this thesis I have described gazing, a technique for controlling the use of rewrite rules within a natural deduction theorem prover. There are two main problems with the peeking and pairs techniques:

- They consider only the *predicates* which appear in the conjecture and rewrite rule.
- They are "short-sighted" in that they perform only one-step look ahead.

Gazing overcomes both of these problems by providing guidance based on all of the concepts in the goal and rule, and by allowing the examination of arbitrary rewriting steps.

Gazing constructs a plan of action (a sequence of rewrite rules which are to be applied) by considering abstractions of the rewrite rules which are available. The abstractions represent the effect of applying the rule by recording the concepts which are altered by the application of the rule. The application of rewrite rules is viewed in our system as a means to exchange the concepts that appear in the current conjecture for a new set of concepts which appear in the new conjecture. The aim of applying rewrite rules is to make the set of concepts appearing in the conclusion of a conjecture the same as that in the hypothesis. This way of viewing the process is called the *common currency model* by Alan Bundy who devised it.

The gazing technique has much in common with the techniques described in [Plaisted 80, Sacerdoti 74] and [Cvetkovic & Pevac 83]. These techniques all use the concept of abstraction in order to plan an appropriate course of action.

The use of the gazing technique has been shown to be *effective* and *inexpensive*. These two facts imply that gazing is a useful tool for theorem proving.

- Gazing is effective in that the construction of the plan can limit the redundant and useless application of rewrite rules as a theorem prover searches for a proof.
- Gazing is inexpensive since search in the abstraction spaces may be carried out efficiently and is much less explosive than that of the problem space.

The efficiency of the search is made possible by the particular abstraction space that have been chosen. Searching for a plan in the abstraction space may be reduced to the problem of searching for paths in a graph, and efficient algorithms exist for this problem. Better yet, the graph is merely the representation of the rewrite rules of the theory in the abstraction space. Once all of these rules are known all paths can be computed and stored in a table. The planning problem then becomes a simple applicability check at prove-time. This applicability check may be performed in linear time. The ability to precompute all paths is implemented as an option in VOYEUR.

The abstraction spaces are much less explosive to search since they contain only abstract representations of the rewrite rules. Two distinct rewrite rules may collapse into one abstract rule in some abstraction space, since in the abstract representation they have the same effects. Of course, the planner must decide which of the rules to eventually use, but this decision may be made on the basis of a different abstraction. The problem of deciding whether a rewrite rule can, and should, be applied may be made much more quickly in the abstraction space. This is because only the relevant features of the rewrite rule are available in the abstraction space.

In experiments with the VOYEUR theorem prover the use of the gazing technique can be seen to lead to shortened proofs. Some of these proofs are presented in appendix B.

6.4.2. What Has Yet To Be Learned

Some questions that are left open by this research into the gazing technique are suggested below.

1. How can the search strategy used in searching the abstraction spaces be tailored (mechanically or by hand) to a particular theory?

2. What features of the theory are important in the design of the search strategy for that theory.

3. How effective is gazing at providing guidance when the number of rewrite rules in the theory gets very large?

4. Can the abstract representation of the theory be put to use in solving other problems in theorem proving and mathematical reasoning?

5. Can gazing be extended to deal with other logical inference techniques (eg, resolution, the connection method, semantic tableaux)?

6. Can gazing be extended to handle recursive definitions?

The first question above is suggested by the observation that *ss1*, the search strategy whose development and testing has been reported here, was devised by examining the proofs and rewrite rules of naive set theory. This theory has certain properties which were used to develop the search strategy. In particular, the theory has similar numbers of functions and predicates, and this has lead to the roughly equal division of labour between the predicate and f/p abstraction spaces. In a theory, for example group theory, where there is only one predicate and many function symbols *ss1* will probably not be so effective, since the number of possible steps in the f/p space will not be constrained by guidance from the predicate space.

The discussion above leads to the second question. If the system is to use different search strategies in different theories, what features of the theories are relevant in making the choice? One feature, as described above, is the relative proliferation of predicates and functions in the theory. A high proportion of one over the other will place too large a burden on one of the abstraction spaces of *ss1* for the technique to result in useful guidance.

The effectiveness of gazing in theories which have very large numbers of rewrite rules is clearly an important question. It seems certain that *ss1* would be inappropriate for such a theory since the abstract representations of rewrite rules would fail to distinguish between many rewrite rules, i.e. many rewrite rules would "collapse" into the same rule in the abstract space. In this case more refined abstraction spaces might be devised to distinguish between such rules. It seems that this extension will be necessary in any later implementation of gazing.

Finally, the extension of gazing to different logical inference systems and to recursive definitions is of interest to determine the generality of the method. The extension to handling recursive definitions seems quite straightforward, although I have not attempted this in the current implementation. Plaisted's work [Plaisted 80] shows that the technique of abstraction mappings can be applied to resolution based provers, but he does not use abstraction mappings to guide the search for the abstract proof, other than to reduce the size of the problem space. It seems clear that gazing can be applied equally to any logical system which does not perform extensive normal-forming.

6.4.3. Summary

In this chapter I have presented:

- Work that is strongly related to that reported in this thesis,
- Possible extensions to the work reported here, and,
- Some overall conclusions from this work.

The work that is related to Gazing falls into three categories. I described this related work in section 6.2, focussing on natural deduction theorem provers which exhibited different methods of controlling database deduction, the uses of abstraction mappings, and the separation of logical and database inference.

Possible extensions to the work presented in this thesis include developing new search strategies which may be used to search for proofs in the abstraction spaces of a theory. This will be necessary since the success of the *ss1* search strategy depends to some extent on the

structure of the theory. In particular, I hope to develop a strategy for planning in theories with one predicate and a large number of functions. *Sɛl* is not appropriate for such a theory since the f/p planner is guided by information from the predicate space planner, and such guidance would not be available in this theory. Identifying features of the theory which can be used determine the choice of strategy for that theory is also an important task.

Finally the conclusions of this thesis may be summed up by saying that gazing can be an *effective and efficient* technique for guiding the choice of rewrite rules to apply to a conjecture in the course of a proof. It can be effective since the guidance that it produces cannot cause the program to apply more redundant or useless rewritings than it would if unguided, and often causes the prover to move more directly to a proof. It can be efficient since the cost of providing the guidance is small. This is because the abstraction spaces may be searched for a plan of action very quickly since they are propositional in nature. The information that is required by the system to produce a plan resides only in the abstract representation of the theory, and thus as soon as the rewrite rules are known this information can, in principle, be computed. If the precomputation is carried out, the retrieval of an appropriate plan during a proof may be carried out only at the expense of a simple applicability check. Thus, while gazing saves much search the expense of carrying out the technique is very slight.

Appendix A
Some Results: Exercises in Set Theory

In the country of the blind
the one-eyed man is king
H. G. WELLS

A.1. Introduction

In this appendix I present some results obtained by using the Voyeur theorem prover* to prove simple theorems in set theory. The conjectures are taken from chapter 1 of [Sigler 66], a book of exercises in naive set theory, designed to accompany [Halmos 60]. [Sigler 66] consists only of definitions and exercises to be attempted by the student. Most of the exercises in the first chapter are very simple and are well suited as conjectures for the Voyeur theorem prover.

In order to assess the effectiveness of gazing, statistics are presented for the proof of each conjecture by Voyeur in RUT mode and in GAZER mode. These two modes can be thought of as two distinct theorem provers and will be referred to as such throughout this appendix. The main advantage of using two modes of the Voyeur prover is that the modes share all of the program that is appropriate to both provers. This factors out advantages that might be gained by either prover being implemented more efficiently than the other.

RUT has access to only the definitions of any theory, as it is unable to guide the use of lemmas. This makes the comparison between RUT and GAZER somewhat unfair, since

a. If the prover needs to use a lemma in the proof then RUT will be unable to use it, and thus a longer proof (or no proof) will be found by RUT than by GAZER, on the other hand,

b. GAZER has to consider the lemmas in constructing its plan for rewriting, and this may cause GAZER to be slowed in its proof attempt.

---

*The Voyeur program is available on request from the author.

**The implementation of GAZER allows the user to choose between the three choices of action (described in chapter 5): (1) "pre-compile" the plans of the theory, (2) produce the plans on the fly but keep them when they have been constructed for later use, or (3) simply compute the plans each time they are needed. For the purposes of the experiments described here, GAZER was used in the third mode. This is slightly unfair to GAZER but either of the other choices would have made it harder to compare RUT with GAZER.

While the results are not presented here, RUT can be used in a mode that allows it to access lemmas in the **Reduce** rules. For none of the conjectures presented in this appendix does this feature improve RUT's ability to prove the conjecture. RUT is able to prove all of the conjectures in either mode, however allowing RUT to use lemmas uniformly degrades RUT's performance by causing it to attempt to apply the lemmas as rewrite rules at each application of the **Reduce** rules.

Chapter 1 of [Sigler 66] divides naturally into three parts. The chapter begins with the presentation of the basic definitions and exercises 1.1 through 1.8. After these exercises the definitions of binary union and intersection are given and then exercises 1.9 through 1.32, concerning these and other concepts. Finally the definition of symmetric difference is stated and exercises 1.33a through 1.33g relating to this concept are presented.

I have similarly divided the presentation of the exercises to the prover into three parts. In the first, the prover has to prove the first set of conjectures (1.1 through 1.10) from the initial axioms. In the second the prover is set the second batch of problems (1.9 through 1.32) and has the previously proved conjectures and all of the definitions to work from. Finally the prover has the conjectures proved in the first and second parts as lemmas, and all of the definitions presented in the chapter for the task of proving exercises 1.33a through 1.33g.

The reader will notice that not all the exercises stated in [Sigler 66] are attempted. Those exercises that are omitted fall into three categories:

- Some of the exercises are concerned with proving that certain objects are sets. I have omitted the axioms for the concept of being a set from this presentation.
- Some exercises are concerned with proving properties of the natural numbers, which are recursively defined as: $0 = \emptyset, n+1 = n \cup \{n\}$. Since neither RUT nor GAZER can handle recursive definitions, these exercises are omitted.
- Some exercises deal with arbitrary sets, e.g. $\{a,b,c,d,e,f\}$. No definition for this appears in chapter 1 of [Sigler 66], and so the reader, as well as the program, cannot solve these problems without additional information.

A.2. The Initial Theory

[Sigler 66] begins with the presentation of the initial axioms of the theory. These axioms are presented in subsection A.2.1. Then some simple propositions about the defined concepts are set as exercises for the student. These conjectures (1.3a through 1.10) are presented in subsection A.2.2 and results for Voyeur's proofs are presented in subsection A.2.3.

A.2.1 Axioms

The initial axioms are given below.

Axiom 1: Definition of $=$
$\forall x,y.[x=y \Leftrightarrow \forall z.z \in x \leftrightarrow z \in y]$

Axiom 2: Definition of $\subseteq$
$\forall x,y.[x \subseteq y \Leftrightarrow \forall z.z \in x \rightarrow z \in y]$

Axiom 3: Definition of $\subset$
$\forall x,y.[x \subset y \Leftrightarrow x \subseteq y \wedge \neg x = y]$

Axiom 4: Definition of generalized intersection
$x \in gen{-}inter(y) \Leftrightarrow \forall z.z \in y \rightarrow x \in z$

Axiom 5: Definition of generalized union
$x \in gen{-}union(y) \Leftrightarrow \exists z.z \in y \wedge x \in z$

Axiom 6: Definition of set difference
$x \in y - z \Leftrightarrow x \in y \wedge \neg x \in z$

Axiom 7: Definition of pair
$x \in \{y,z\} \Leftrightarrow x = y \vee x = z$

Axiom 8: Definition of power set
$x \in 2^y \Leftrightarrow x \subseteq y$

Axiom 9: Definition of $\emptyset$
$x \in \emptyset \Leftrightarrow \bot$

This theory is very small and there are no choice points for the provers, i.e., there is at most one way of rewriting a particular concept to another. This means that, for the purposes of assessing the success of gazing at selecting facts to use in a proof, the theory is not sufficiently rich. Gazing will never have any choices to make. However, the results of the proofs for the conjectures which follow do indicate the extent to which gazing can speed up the task of proving theorems. The reason for the increase in speed is that GAZER does not attempt to rewrite the conjecture using these axioms unless the system is applying the gazing inference rule, and then the prover knows exactly which rules to apply before attempting the rewriting. RUT attempts to apply the function definitions to the conjecture every time the **Reduce** rules are called by the prover, and these appear early in the inference rule ordering. Thus the attempted application of these rules can slow the prover down dramatically. The results in section A.2.3 indicate this phenomenon.

A.2.2 Conjectures

The first conjectures that are set as exercises in [Sigler 66] are quite simple. Many of them are proved by gazing simply applying a default rule such as,

- Introduce both polarities if the hypothesis or conclusion of the conjecture is empty, and,
- Apply *any* rule if the conjecture is already expressed in a common currency.

RUT too, quickly rewrites these conjectures to $\top$.

Exercises 1.1 and 1.2 prove trivial equivalences of logic which are assumed by Voyeur, and so it is not interesting to submit these to the prover.

Lemma 1: Reflexivity of $=$ (1.3a)
$\forall x.[x = x]$

Lemma 2: Symmetry of $=$ (1.3b)
$\forall x,y.[x = y \rightarrow y = x]$

Lemma 3: Transitivity of $=$ (1.3c)
$\forall x,y,z.[x = y \wedge y = z \rightarrow x = z]$

Lemma 4: Reflexivity of $\subseteq$ (1.4a)
$\forall x.[x \subseteq x]$

Lemma 5: AntiSymmetry of $=$ (1.4b)
$\forall x,y.[x \subseteq y \wedge y \subseteq x \rightarrow x = y]$

Lemma 6: Transitivity of $\subseteq$ (1.4c)
$\forall x,y,z.[x \subseteq y \wedge y \subseteq z \rightarrow x \subseteq z]$

Lemma 7: Transitivity of $\subset$ (1.5)
$\forall x,y,z.[x \subset y \wedge y \subset z \rightarrow x \subset z]$

Lemma 8: Cycle of $\subseteq$ implies $=$ (1.6)
$\forall x,y,z.[x \subseteq y \wedge y \subseteq z \wedge z \subseteq x \rightarrow x = y \wedge y = z]$

Lemma 9: (1.7a)
$\forall x,y,z.[x \subseteq z \wedge y \subseteq z \wedge x \subseteq y \rightarrow z - y \subseteq z - x]$

Lemma 10: (1.7b)
$\forall x,z.[x \subseteq z \rightarrow x = z - (z - x)]$

Lemma 11: $\emptyset$ is a Subset of all sets (1.8a)
$\forall x.[\emptyset \subseteq x]$

Lemma 12: The only Subset of $\emptyset$ is $\emptyset$ (1.8b)
$\forall x.[x \subseteq \emptyset \rightarrow x = \emptyset]$

A.2.3 Results

The results for the simple initial conjectures (1.3 - 1.8 in [Sigler 66]) are presented in figure A-1. They indicate the extent to which separating the theory and logical inferences can speed up the proof of some conjectures. As remarked in section A.2.2 since the prover has no choices to make in selecting rewrite rules, and the proofs are very simple, these results indicate how the prover is sped up by virtue of separating inferences in this way.

For each conjecture above the following statistics are given:

- The time taken to perform the proof of the conjecture, and,
- The number of subgoals generated in the course of the proof and the number of subgoals that were necessary in that proof.

The latter is a measure of the amount of work that was done by the prover which was not useful to the proof. A ratio of N/N for any N means that every subgoal suggested by the prover was part of the eventual proof. The number of subgoals in the proof also gives a measure of the complexity of that proof.

These statistics are given for both RUT and GAZER. These statistics were obtained by a version of Voyeur running on a SUN 3/50 workstation implemented in Quintus Prolog version 1.6. Voyeur was run in a non-interactive mode. If the program were run interactively the program could probably be persuaded to find shorter proofs by using the interactive commands to prune the search.

With the exception of lemma 4, GAZER performs better than RUT for this set of conjectures. The reason for the increased time for proving lemma 4 is that RUT simply uses the definition of $\subseteq$ in an application of **Define C** as its first step, while GAZER uses the **Gaze** rule to achieve the same result. The time saved by GAZER because it does not attempt to apply the function definition rewrite rules in the **Reduce** rule is lost in the application of the more expensive **Gaze** rule. Since both provers apply the definition of $\subseteq$, the rest of the proof, which involves reducing the resulting implication to $\top$ is the same for both provers.

Notice that the proofs produced by GAZER and RUT are not always the same, in particular they are sometimes of different lengths. This difference arises only because of the different handling of the rewrite rules. There is a "knock-on" effect of this different handling, which is caused by the logical rules of the prover receiving different sequents to work on. A simple difference in the order in which inferences are made at one point in the proof, can cause the proofs to diverge dramatically. This happens most noticeably when one prover applies a splitting rule, when the other does not. The result is that the prover which makes the split,

Conjecture Number	RUT Time (in seconds)	RUT Subgoals Used/Gen.	GAZER Time (in seconds)	GAZER Subgoals Used/Gen.
1	0.55	6/6	0.40	2/2
2	6.83	7/7	3.72	7/7
3	12.50	7/7	7.97	7/7
4	0.50	3/3	0.38	2/2
5	8.22	7/7	4.47	7/7
6	5.70	4/4	2.97	4/4
7	315.48	44/49	116.28	55/59
8	29.30	14/14	14.95	14/14
9	31.13	10/10	10.17	8/8
10	26.48	17/17	8.75	15/15
11	0.62	3/3	0.53	2/2
12	12.52	20/22	4.53	11/11

Figure A-1: Results for the initial theory

has to do the same work for the two different cases, where if the sequent is provable without splitting then the work can be done in one case.

The results for the proof of lemma 4 characterizes the tension in the choice to use gazing. For simple conjectures in simple theories, the cost of gazing can outweigh the cost of "blind" rewriting. While a plan of appropriate rewriting steps is being crafted by the gazing inference rule, "blind" rewriting can be carrying out many rewriting steps that may quickly lead to a proof. The utility of gazing is only recognized when the conjecture to prove does not quickly fall to the "blind" rewriting method. Gazing does however improve the proof time for all of these conjectures, in many cases by a factor of about 3. This phenomenon is to be expected when devising a search control technique. When resources are expended to control search rather than carry out unguided search, it will be more expensive for those problems which quickly succumb to blind search. However, the ability to guide the search for a solution to hard problems, thus using considerably less resources far outweighs the degraded performance on the simpler problems.

A.3. Extending The Theory

After the conjectures which are proved in the initial theory, the concepts $\cup$ and $\cap$ are defined. The definitions are given below.

Axiom 10: Definition of $\cup$
$x \in y \cup z \Leftrightarrow x \in y \vee x \in z$

Axiom 11: Definition of $\cap$
$x \in y \cap z \Leftrightarrow x \in y \wedge x \in z$

After the presentation of these definitions, conjectures 13 through 33 are proved in an extended theory which consists of the initial axioms, lemmas 1 through 12, and these two definitions. There are three conjectures that neither GAZER nor RUT can prove. These are presented here as lemmas 13, 14 and 16. The reason for failure in all three cases is determinism in the rewriting package. While both provers are able to remake the choice to use a particular inference rule should the original choice fail to lead to a proof, neither prover is able to remake choices made within the application of an inference rule. When rewriting, the provers apply the first sequence of rewrite rules that are applicable, and this choice is fixed even on backtracking. In the case of these two conjectures, a rewrite rule is applied which binds a variable to a term which causes failure in another part of the proof. This choice is not remade, and thus the proof fails.

A.3.1 Conjectures for the Extended Theory

The conjectures to be proved in the extended theory are given in this subsection.

Lemma 13: $\cup$ is a special case of *gen-union*
$\forall x,y.[x \cup y = gen\text{-}union(\{x,y\})$

Lemma 14: $\cap$ is a special case of *gen-inter*
$\forall x,y.[x \cap y = gen\text{-}inter(\{x,y\})$

Lemma 15: $\emptyset$ is an identity for $\cup$ (1.10)
$\forall x.[x \cup \emptyset = x]$

Lemma 16: $\emptyset$ is only identity for $\cup$ (1.11)
$\forall x.[x \cup y = x \rightarrow y = \emptyset]$

Lemma 17: $\emptyset$ is a fixpoint for $\cap$ (1.12)
$\forall x.[x \cap \emptyset = \emptyset]$

Lemma 18: Commutativity of $\cup$ (1.13a)
$\forall x,y.[x \cup y = y \cup x]$

Lemma 19: Commutativity of $\cap$ (1.13b)
$\forall x,y.[x \cap y = y \cap x]$

Lemma 20: Associativity of $\cup$ (1.13c)
$\forall x,y,z.[x \cup (y \cup z) = (x \cup y) \cup z]$

Lemma 21: Associativity of $\cap$ (1.13d)
$\forall x,y,z.[x \cap (y \cap z) = (x \cap y) \cap z]$

Lemma 22: Distributivity of $\cap$ over $\cup$ (1.13e)
$\forall x,y,z.[x \cap (y \cup z) = (x \cap y) \cup (x \cap z)]$

Lemma 23: Distributivity of $\cup$ over $\cap$ (1.13f)
$\forall x,y,z.[x \cup (y \cap z) = (x \cup y) \cap (x \cup z)]$

Lemma 24: Idempotency of $\cup$ (1.13g)
$\forall x.[x \cup x = x]$

Lemma 25: Idempotency of $\cap$ (1.13h)
$\forall x.[x \cap x = x]$

Lemma 26: (1.26a)
$\forall u,x,y.[(x \subseteq u \wedge y \subseteq u) \rightarrow (x \subseteq y) \leftrightarrow x \cap (u - y) = \emptyset]$

Lemma 27: (1.26b)
$\forall u,x,y.[(x \subseteq u \wedge y \subseteq u) \rightarrow (x \subseteq y) \leftrightarrow (u - x) \cup y = u]$

Lemma 28: (1.26c)
$\forall u,x,y.[(x \subseteq u \wedge y \subseteq u) \rightarrow (x \subseteq y) \leftrightarrow x \cap (u - y) \subseteq (u - x)]$

Lemma 29: (1.26d)
$\forall u,x,y.[(x \subseteq u \wedge y \subseteq u) \rightarrow (x \subseteq y) \leftrightarrow x \cap (u - y) \subseteq y]$

Lemma 30: (1.26e)
$\forall u,w,x,y.[(x \subseteq u \wedge y \subseteq u) \rightarrow (x \subseteq y) \leftrightarrow x \cap (u - y) \subseteq w \cap (u - w)]$

Lemma 31: (1.27)
$\forall x,y.[x \cup y = x \leftrightarrow y \subseteq x]$

Lemma 32: (1.28)
$\forall x,y.[x \cap y = x \leftrightarrow x \subseteq y]$

Lemma 33: (1.30)
$\forall x,y.[gen\text{-}inter(x) \cap gen\text{-}inter(y) \subseteq gen\text{-}inter(x \cap y)]$

A.3.2 Results for Conjectures 13 through 33

The results for lemmas 13 through 33 are shown in figure A-2.

Gazing is not uniformly successful in reducing the time taken to produce a proof from that taken by RUT. However for the genuinely hard conjectures gazing gives a significant improvement to the length of time taken to perform the proof. In this theory, the cost of gazing is more than it previously was, since there are many more choices to be made in the abstraction spaces. If the eventual plan to be used is simply to apply rewrite rules that RUT would have used anyway, then RUT has a computational advantage over GAZER. This shows in the results for the conjectures where GAZER takes much longer to prove the conjecture than RUT does, (conjectures 15, 17, 20, 21, 24, 25 and 32) . In some cases it takes GAZER slightly more than twice as long to complete the proof of the conjecture than RUT (15 20, 21 and 25. On the other hand, when gazing pays off, the improvement is dramatic, consider, for example, lemma 23, where RUT takes 20 times as long as GAZER to produce the proof.

Conjecture Number	RUT Time in seconds	RUT Subgoals Used/Gen.	GAZER Time in seconds	GAZER Subgoals Used/Gen.
13	-	-	-	-
14	-	-	-	-
15	2.27	6/6	6.16	8/8
16	-	-	-	-
17	2.12	6/6	3.17	5/5
18	64.93	24/24	8.87	12/12
19	3.70	6/6	9.30	12/12
20	5.90	6/6	15.88	16/16
21	7.40	6/6	18.00	16/16
22	168.55	26/26	32.28	30/30
23	639.20	111/111	32.87	30/30
24	1.97	6/6	5.83	8/8
25	2.28	6/6	5.40	8/8
26	106.28	25/25	44.38	21/21
27	129.25	27/27	88.47	29/29
28	140.16	22/22	46.22	20/20
29	94.13	16/16	48.13	13/13
30	206.15	22/22	81.30	20/20
31	34.38	20/20	39.98	12/12
32	22.77	14/14	40.48	9/9
33	40.10	5/5	11.83	5/5

Figure A-2: Results for the extended theory

A.3.3 The Definition of Symmetric Difference and its Associated Conjectures

Finally in chapter 1 of [Sigler 66], the definition of symmetric difference (axiom 12) is stated. The conjectures 34 through 40 are set for the student.

Again, the theory is extended before these conjectures are presented to Voyeur. The theory for these conjectures has conjectures 1 through 33 added as lemmas.

Axiom 12: Definition of $+$
$x + y == (x - y) \cup (y - x)$

A.3.4 Conjectures for the Final Theory

Here are the conjectures to be proved in this theory:

Lemma 34: $\emptyset$ identity of $+$ (1.33a)
$\forall x.[x + \emptyset = x]$

Lemma 35: Commutativity of $+$ (1.33b)
$\forall x,y.[x + y = y + x]$

Lemma 36: Associativity of $+$ (1.33c)
$\forall x,y,z.[x + (y + z) = (x + y) + z]$

Lemma 37: $\cap$ distributes over $+$ (1.33d)
$\forall x,y,z.[x \cap (y + z) = (x \cap y) + (x \cap z)]$

Lemma 38: $-$ subset of $+$ (1.33e)
$\forall x,y.[x - y \subseteq x + y]$

Lemma 39: $=$ iff $+$ is $\emptyset$ (1.33f)
$\forall x,y.[x = y \leftrightarrow x + y = \emptyset]$

Lemma 40: Cancellation of $+$ (1.33g)
$\forall x,y,z.[x + z = y + z \rightarrow x = y]$

A.3.5 Results for Conjectures 34 through 40

The results for conjectures 34 through 40 are presented in figure A-3. As before, gazing is not successful in improving the performance for all of the conjectures, (conjectures 34, 38 and 39, take longer to prove in GAZER) but for the harder conjectures the improvement is obtained and is quite dramatic. Consider lemma 36 for example, where RUT takes more than ten times as long to prove the conjecture than GAZER does.

There is one conjecture from this set that neither prover can prove, namely 40. In this case failure is caused by the provers' inability to introduce tautologies as hypotheses. Once the defined concepts in the conjecture have been eliminated the conjecture is as below:

$$
\begin{array}{l}
[(w \in a \wedge \neg\, w \in c) \vee (w \in c \wedge \neg\, w \in a)] \rightarrow \\
\quad [(w \in b \wedge \neg\, w \in c) \vee (w \in c \wedge \neg\, w \in b)] \wedge \\
[(w \in b \wedge \neg\, w \in c) \vee (w \in c \wedge \neg\, w \in b)] \rightarrow \\
\quad [(w \in a \wedge \neg\, w \in c) \vee (w \in c \wedge \neg\, w \in a)] \\
\qquad\qquad \vdash^{\dagger} \\
(s \in b \rightarrow s \in a) \wedge (s \in a \rightarrow s \in b)
\end{array}
\tag{8}
$$

To complete the proof it is necessary to reason by cases. Under first the assumption that $s \in c$ and later that $\neg\, a \in c$, the conjecture can be proved. Once we have this, the conjecture is proved by the law of excluded middle. The prover fails since it cannot "invent" this argument for itself. The next implementation of VOYEUR will allow the user to have the prover add additional hypotheses, after first proving them valid. However, even then the provers will not be able to handle this conjecture alone.

	RUT		GAZER	
Conjecture Number	Time in seconds	Subgoals Used/Gen.	Time in seconds	Subgoals Used/Gen.
34	13.28	7/7	45.10	8/8
35	415.78	125/125	14.43	10/10
36	9 108.18	841/841	910.38	282/282
37	858.31	127/127	340.35	100/100
38	3.52	4/4	34.68	14/14
39	90.10	44/44	1 132.93	39/39
40	-	-	-	-

Figure A-3: Results for conjectures 34 through 40

Again we can see that for some conjectures the improvement due to the use of gazing is dramatic. On the other hand, for some conjectures the use of gazing increases the time taken to complete the proof.

A.4. Proving What is Already Known

As a final experiment in this theory, I again submitted to both GAZER and RUT each of the lemmas 1 through 40. This time though, the theory in which the provers were working contained each of those lemmas as well as the definitions of the theory. Thus the new theory contains 40 lemmas in addition to the axioms of the theory. The idea behind this experiment is to determine how effectively the theorem provers are able to utilize the knowledge that is contained in the database of rewrite rules. It is to be hoped that any theorem prover would

be able to find the trivial proof of any conjecture which is identical to some fact that it is supposed to "know". The idea here is not to ensure that the prover can construct trivial proofs of conjectures exactly the same as those which it already has access to, but rather to test whether the prover could prove a conjecture which involves as a subgoal of the proof, some conjecture which matches what is already known. A good example of this arises, for example, in the proof of the transitivity of $\subset$ (lemma 7). One subgoal that arises in the course of this proof is to prove the transitivity of $\subseteq$ (lemma 6). If the prover has already proved lemma 6, and had this conjecture added to its database, than it should prove this subgoal trivially. This ability is not, in general, available to provers. RUT, for example cannot utilize any of the lemmas of the theory because it has no techniques for controlling their use. If RUT is operated in a mode where it does have access to the lemmas in an unguided fashion (that is, it will apply them whenever it can) its performance is degraded considerably because it spends a large amount of time attempting to apply lemmas when it is impossible so to do.

The results for this experiment are presented in figures A-4 and A-5. GAZER is able to make quite good use of the database as evidenced by the increased speed with which the proofs are found. However for come conjectures (for example, 26 through 30) the time taken to prove the lemma is increased. This is because this theory is much more complex than the previous theories, and therefore the gazing inference rule is much more expensive to apply. This increase in complexity is overcome for other conjectures by the ability to access the appropriate fact needed to prove the conjecture but this isn't possible in these cases. The lemmas 26 through 30 are all of the form $(P \wedge Q) \rightarrow (R \leftrightarrow S)$. The real force of these lemmas when used as rewrite rules should be as an exchange between R to S whenever you can prove P and Q. This is called conditional rewriting, and is a notion that GAZER lacks. GAZER sees this as a rewrite rule with P and Q on one side, and R and S on the other. Further when the proof of the lemma is attempted R and S are split up, and so the rewrite rule is not seen to apply. In general, GAZER finds it easier to recognize the appropriateness of simple rewrite rules and fails when they have more complex structure.

A.5. Conclusion

In this appendix I have exhibited the results obtained by using the Voyeur theorem prover to prove a number of conjectures in set theory. The conjectures were taken from a book of exercises, [Sigler 66], which is designed to be used by students learning the theory. The conjectures were presented to the theorem prover in both RUT and GAZER modes, and statistics gathered which indicate the degree to which the use of gazing is successful in shortening the proofs produced by the prover.

Conjecture Number	Time in seconds	Subgoals Used/Gen.	Old Time in seconds	Old Subgoals Used/Gen.	New/Old (time)
1	0.10	2/2	0.40	2/2	0.25
2	0.63	4/4	3.72	7/7	0.17
3	0.95	3/3	7.97	7/7	0.12
4	0.07	2/2	0.38	2/2	0.18
5	12.95	5/5	4.47	7/7	2.90
6	3.97	4/4	2.97	4/4	1.34
7	34.27	20/20	116.28	55/59	0.29
8	37.95	14/14	14.95	14/14	2.54
9	7.23	7/7	10.17	8/8	0.71
10	13.27	3/3	8.75	15/15	1.52
11	0.10	2/2	0.53	2/2	0.19
12	9.32	8/8	4.53	11/11	2.06
13	-	-	-	-	-
14	-	-	-	-	-
15	7.10	8/8	6.16	8/8	1.15
16	-	-	-	-	-
17	3.53	4/4	3.17	5/5	1.11
18	5.58	8/8	8.87	12/12	0.63
19	5.57	8/8	9.30	12/12	0.60
20	7.63	8/8	15.88	16/16	0.55
21	7.53	8/8	18.00	16/16	0.42
22	42.25	30/30	32.28	30/30	1.32
23	41.93	30/30	32.87	30/30	1.28
24	8.28	8/8	5.83	8/8	1.42
25	8.42	8/8	5.40	8/8	1.56
26	74.83	20/20	44.38	21/21	1.69
27	228.72	29/29	88.47	29/29	2.59
28	152.47	19/19	46.22	20/20	3.30
29	167.20	13/13	48.13	13/13	3.47
30	204.87	19/19	81.30	20/20	2.52
31	16.47	7/7	39.98	12/12	0.41
32	14.77	7/7	40.48	9/9	0.36
33	0.23	2/2	11.83	5/5	0.02

Figure A-4: Results for GAZER in full theory (part 1)

Conjecture Number	Time in seconds	Subgoals Used/Gen.	Old Time in seconds	Old Subgoals Used/Gen.	New/Old (time)
34	10.67	8/8	45.10	8/8	0.24
35	5.53	8/8	14.43	10/10	0.85
36	7.63	8/8	910.38	282/282	0.01
37	288.77	96/96	340.35	100/100	0.38
38	0.18	2/2	34.68	14/14	0.01
39	1 132.93	39/39	160.93	11/11	7.04

Figure A-5: Results for GAZER in full theory (part 2)

For simple conjectures it was found that gazing has only a small effect. This is as we would expect since gazing is quite expensive to apply. For simple conjectures in simple theories, a fast, "blind" approach will often be successful, since the amount of work that may be performed redundantly or incorrectly is quite small.

For more complex theories the use of gazing can reap considerable benefit, since the chance of the prover utilizing "blind" search carrying out redundant work is much larger. This is seen most dramatically when considering conjecture 36 above.

Some of the proofs of the conjectures in this chapter are presented in appendix B.

Appendix B
Some Example Proofs

B.1. Introduction

In this appendix I present some proofs created by the Voyeur theorem prover. I have elected to present the proofs of three of the conjectures mentioned in appendix A, one from each of the three groups of conjectures in that appendix. For each conjecture I show the proof produced by GAZER and that produced by RUT. The proofs are shown here as they are produced by Voyeur with two exceptions,

- Wherever possible I have simplified the proofs by replacing skolem terms introduced by the expansion of a definition, by a new constant. For example, in figure B-1, I have replaced the term $sig1.sf1(\emptyset,sf1)$ with the constant $t1$ throughout the proof. This is merely a notational convenience. The skolem function $sig1.sf1$ is obtained by skolemizing the definition of $=$, and is used to represent an arbitrary term which depends on the two sets which are claimed to be equal.

- I have broken the lines of output in convenient places, while Voyeur produces a single line of output for each line of the proof.

Proofs are presented in the form described in chapter 5. In this format a proof is a sequence of sequents. Each sequent is either an axiom from the logic, or follows from sequents earlier in the sequence by a rule of logic. Thus in the representation each proof consists of a number of lines each of which consists of three parts: the *line label*, the *statement* and the *justification*.

- The line label appears in parentheses to the left of the line, it serves only to name the line for reference by the justifications.

- The statement is the sequent that is asserted to be true by the line, and,

- The justification is a reference to the line(s) which show that the statement is indeed true. The justification appears to the right of the statement, and may be empty. If it is empty the statement is true by an axiom of logic.

This presentation of the proof makes it appear that the proof was performed by forward inference, but as described in chapter 1, the UT provers, RUT and GAZER all produce proofs by a mixture of forward and backward inference.

In addition to the sequents that appear in the proof, the sequents that are suggested by the prover but which it subsequently fails to prove are also shown. These sequents represent the "blind alleys" that the prover investigates uselessly. These sequents are distinguished from those which are successfully proved by the typeface in which their numbers are printed. The numbers of those sequents that are in the proof are shown in **bold** face, while those which are not in the proof are shown in normal face.

B.2. The proof of conjecture 12

The first proofs that I present are of conjecture 12 from appendix A. The proofs are shown in figures B-1 and B-2.

Conjecture: $(\forall x.((x \subseteq \emptyset) \leftrightarrow (x = \emptyset)))$

(6) $\bot \vdash \bot$ ()

(5) $(t1 \in sf1) \wedge (\neg(t1 \in sf1)) \vdash \bot$ **(6)**

(4) $(\neg(t1 \in sf1)) \vdash (\neg(t1 \in sf1))$ **(5)**

(3) $(sf1 \subseteq \emptyset) \vdash (sf1 = \emptyset)$ **(4)**

(1) $\top \vdash ((sf1 \subseteq \emptyset) \rightarrow (sf1 = \emptyset))$ **(3)**

(10) $\bot \vdash \bot$ ()

(9) $(t2 \in sf1) \wedge (\neg(t2 \in sf1)) \vdash \bot$ **(10)**

(8) $(\neg(t2 \in sf1)) \vdash (\neg(t2 \in sf1))$ **(9)**

(7) $(sf1 = \emptyset) \vdash (sf1 \subseteq \emptyset)$ **(8)**

(2) $\top \vdash ((sf1 = \emptyset) \rightarrow (sf1 \subseteq \emptyset))$ **(7)**

(0) $\top \vdash (((sf1 \subseteq \emptyset) \rightarrow (sf1 = \emptyset)) \wedge ((sf1 = \emptyset) \rightarrow (sf1 \subseteq \emptyset)))$ **(1,2)**

Note: $t1 = sig1.sf1(\emptyset, sf1)$
$t2 = sig1.sf2(\emptyset, sf1)$

Figure B-1: The proof of conjecture 12, produced by GAZER

The proof produced by GAZER is 11 lines long, while that produced by RUT has 20 lines. The main reason for the increase in length when RUT is used is that RUT does not immediately recognize that both the predicates $=$ and $\subseteq$ must be rewritten to $\in$ before the conjecture can be proved. In the proof produced by GAZER there are two uses of the gazing inference rule. These produce line 4 from line 3 and line 8 from line 7 respectively. Considering the step from line 7 to line 8, GAZER has recognized that, to introduce the same

predicate the definition of $\subseteq$ must be applied to the conclusion, and the definition of $=$ must be applied to the hypothesis. Since there are no functions to be considered, these steps comprise the complete plan. Once the plan has been executed, GAZER calls the reduce rules on the resulting sequent, and this causes the definition of $\emptyset$ to be used and the sequent simplified to an obvious truth*.

In the RUT proof things proceed much more slowly. Step 15 is obtained from step 14 by the expansion of the definition of $\subseteq$, then this formula is reduced to obtain line 16. At this point RUT goes off down a blind alley which is shown in the proof as line 17. RUT has no inference rule which may be applied and the proof of this goal fails**. RUT backtracks and applies **Peek** to line 16, causing the expansion of the predicate $=$. The resulting sequent then simplifies, and is easily proved.

B.3. The proof of conjecture 32

Next I consider the proofs of conjecture 32 from the second group of conjectures in appendix A. The proof produced by GAZER is in figure B-3, and that produced by RUT is in figure B-4.

Again the proof produced by GAZER is shorter than that produced by RUT, but notice that it took GAZER more time to produce the shorter proof. Both provers are forced to backtrack in the course of the proof.

The proof produced by GAZER is quite straightforward. After skolemization, there are two implications to prove. Promoting the hypothesis of the implication gives a goal to which the gazing inference rule is applied. The hypothesis of line 6 has the predicate $\subseteq$, while the conclusion of that goal has predicate $=$ and a function symbol $\cap$. The gazing inference rule suggests that the definition of $\subseteq$ be applied to the hypothesis, and the definitions of $=$ and $\cap$ to the conclusion. After simplification, the goal is proved by promotion and forward chaining. The proof of the opposite implication is dual.

The RUT proof contains more steps because the concepts in the proof are handled more warily. This is not, in general, the case for RUT as when many functions appear in a goal to prove, they are often all eliminated simultaneously by the **Reduce** rule. In this case, there is only one function, and this can only be eliminated after the expansion of the definition of the

---

*Both GAZER and RUT find this obvious truth a little tricky to prove, since **Flip C** appears in the rule base before **Match**. The conclusion is un-negated and moved to the hypothesis, and then a contradiction deduced.

**There is a similar blind alley in the proof of the other implication: goal 8 also fails.

Conjecture: $(\forall x.((x \subseteq \emptyset) \leftrightarrow (x = \emptyset)))$

(12) $\bot \vdash \bot$ ()

(11) $(t1 \in sf1) \wedge (\neg(t1 \in sf1)) \vdash \bot$ **(12)**

(10) $(\neg(t1 \in sf1)) \vdash (\neg(t1 \in sf1))$ **(11)**

(9) $((t1 \in sf1) \rightarrow (t1 \in \emptyset)) \vdash (\neg(t1 \in sf1))$ **(10)**

(8) $(sf1 \subseteq \emptyset) \wedge (t1 \in sf1) \vdash \bot$

(7) $(sf1 \subseteq \emptyset) \vdash (\neg(t1 \in sf1))$ **(9)**

(5) $(sf1 \subseteq \emptyset) \vdash ((t1 \in sf1) \rightarrow (t1 \in \emptyset))$ **(7)**

(13) $(sf1 \subseteq \emptyset) \vdash \top$ ()

(6) $(sf1 \subseteq \emptyset) \vdash ((t1 \in \emptyset) \rightarrow (t1 \in sf1))$ **(13)**

(4) $(sf1 \subseteq \emptyset) \vdash (((t1 \in sf1) \rightarrow (t1 \in \emptyset)) \wedge ((t1 \in \emptyset) \rightarrow (t1 \in sf1)))$ **(5,6)**

(3) $(sf1 \subseteq \emptyset) \vdash (sf1 = \emptyset)$ **(4)**

(1) $\top \vdash ((sf1 \subseteq \emptyset) \rightarrow (sf1 = \emptyset))$ **(3)**

(21) $\bot \vdash \bot$ ()

(20) $(t2 \in sf1) \wedge (\neg(t2 \in sf1)) \vdash \bot$ **(21)**

(19) $(\neg(t2 \in sf1)) \vdash (\neg(t2 \in sf1))$ **(20)**

(18) $((t2 \in sf1) \rightarrow (t2 \in \emptyset)) \wedge ((t2 \in \emptyset) \rightarrow (t2 \in sf1)) \vdash (\neg(t2 \in sf1))$ **(19)**

(17) $(sf1 = \emptyset) \wedge (t2 \in sf1) \vdash \bot$

(16) $(sf1 = \emptyset) \vdash (\neg(t2 \in sf1))$ **(18)**

(15) $(sf1 = \emptyset) \vdash ((t2 \in sf1) \rightarrow (t2 \in \emptyset))$ **(16)**

(14) $(sf1 = \emptyset) \vdash (sf1 \subseteq \emptyset)$ **(15)**

(2) $\top \vdash ((sf1 = \emptyset) \rightarrow (sf1 \subseteq \emptyset))$ **(14)**

(0) $\top \vdash (((sf1 \subseteq \emptyset) \rightarrow (sf1 = \emptyset)) \wedge ((sf1 = \emptyset) \rightarrow (sf1 \subseteq \emptyset)))$ **(1,2)**

Note: $t1 = sig1.sf1(\emptyset, sf1)$
$t2 = sig1.sf2(\emptyset, sf1)$

Figure B-2: The proofs of conjecture 12, produced by RUT

predicate =. The RUT proof begins with the same steps as GAZER. The two implications are proved separately and the antecedents promoted to explicit hypotheses. At line 7, RUT

Conjecture: $(\forall x.(\forall y.(((x \cap y) = x) \leftrightarrow (x \subseteq y))))$

(5) $(t1 \in sf1) \wedge (t1 \in sf2) \wedge ((t1 \in sf1) \rightarrow (t1 \in sf2)) \vdash (t1 \in sf2)$ ()

(4) $((t1 \in sf1) \rightarrow (t1 \in sf2)) \vdash ((t1 \in sf1) \rightarrow (t1 \in sf2))$ **(5)**

(3) $((sf1 \cap sf2) = sf1) \vdash (sf1 \subseteq sf2)$ **(4)**

(1) $\top \vdash (((sf1 \cap sf2) = sf1) \rightarrow (sf1 \subseteq sf2))$ **(3)**

(8) $(t2 \in sf1) \wedge (t2 \in sf2) \wedge ((var167 \in sf1) \rightarrow (var167 \in sf2)) \vdash (t2 \in sf2)$ ()

(7) $((var167 \in sf1) \rightarrow (var167 \in sf2)) \vdash ((t2 \in sf1) \rightarrow (t2 \in sf2))$ **(8)**

(6) $(sf1 \subseteq sf2) \vdash ((sf1 \cap sf2) = sf1)$ **(7)**

(2) $\top \vdash ((sf1 \subseteq sf2) \rightarrow ((sf1 \cap sf2) = sf1))$ **(6)**

(0) $\top \vdash ((((sf1 \cap sf2) = sf1) \rightarrow (sf1 \subseteq sf2)) \wedge$
$((sf1 \subseteq sf2) \rightarrow ((sf1 \cap sf2) = sf1)))$ **(1,2)**

Note: $t1 = sig1.sf2(sf2, sf1)$
$t2 = sig1.sf2((sf1 \cap sf2), sf1)$

Figure B-3: The proof of conjecture 32, produced by GAZER

expands the definition of the predicate $=$, which results in another conjunction in the conclusion. GAZER does not see this conjunction, as one of the conjuncts may be quickly simplified to $\top$. RUT also performs this simplification, but the reduction appears explicitly in the proof, and is performed after the splitting of the introduced conjunction (between lines 9 and 11).

Once the definition of $=$ has been expanded, and the resulting conjunction split. RUT then reduces the conclusion by eliminating the function $\cap$. The resulting subgoal (line 12) is easily proved by promotion and peek forward chaining. Note that it is the capability to peek forward chain that enables RUT to prove this goal without explicitly expanding the definition of $\subseteq$ in the hypothesis.

B.4. The proof of conjecture 35

In this section I give the proofs of conjecture 35 from the previous appendix. The proof produced by GAZER is only 10 lines long, while RUT produces a proof of 125 lines. For reasons of space, I do not show the whole of the RUT proof here, but only the first few steps.

The proof produced by GAZER relies on the previously proved lemma that $\cup$ is commutative (conjecture 18). The interesting steps are in the proof lines 4-6 and 7-9. These groups are

Conjecture: $(\forall x.(\forall y.(((x \cap y) = x) \leftrightarrow (x \subseteq y))))$

(6) $(t1 \in sf2) \wedge (t1 \in sf1) \wedge ((sf1 \cap sf2) = sf1) \vdash (t1 \in sf2)$ ()

(5) $(t1 \in sf1) \wedge (t1 \in (sf1 \cap sf2)) \wedge ((sf1 \cap sf2) = sf1) \vdash (t1 \in sf2)$ **(6)**

(4) $((sf1 \cap sf2) = sf1) \vdash ((t1 \in sf1) \rightarrow (t1 \in sf2))$ **(5)**

(3) $((sf1 \cap sf2) = sf1) \vdash (sf1 \subseteq sf2)$ **(4)**

(1) $\top \vdash (((sf1 \cap sf2) = sf1) \rightarrow (sf1 \subseteq sf2))$ **(3)**

(11) $(sf1 \subseteq sf2) \vdash \top$ ()

(9) $(sf1 \subseteq sf2) \vdash ((t2 \in (sf1 \cap sf2)) \rightarrow (t2 \in sf1))$ **(11)**

(13) $(t2 \in sf1) \wedge (t2 \in sf2) \wedge (sf1 \subseteq sf2) \vdash (t2 \in sf2)$ ()

(12) $(sf1 \subseteq sf2) \vdash ((t2 \in sf1) \rightarrow (t2 \in sf2))$ **(13)**

(10) $(sf1 \subseteq sf2) \vdash ((t2 \in sf1) \rightarrow (t2 \in (sf1 \cap sf2)))$ **(12)**

(8) $(sf1 \subseteq sf2)$
$\vdash (((t2 \in (sf1 \cap sf2)) \rightarrow (t2 \in sf1)) \wedge$
$((t2 \in sf1) \rightarrow (t2 \in (sf1 \cap sf2))))$ **(9,10)**

(7) $(sf1 \subseteq sf2) \vdash ((sf1 \cap sf2) = sf1)$ **(8)**

(2) $\top \vdash ((sf1 \subseteq sf2) \rightarrow ((sf1 \cap sf2) = sf1))$ **(7)**

(0) $\top \vdash ((((sf1 \cap sf2) = sf1) \rightarrow (sf1 \subseteq sf2)) \wedge$
$((sf1 \subseteq sf2) \rightarrow ((sf1 \cap sf2) = sf1)))$ **(1,2)**

Note $t1 = sig1.sf2(sf2, sf1)$
$t2 = sig1.sf1(sf1, (sf1 \cap sf2))$

Figure B-4: The proof of conjecture 32, produced by RUT

dual, as they are required to prove one half of a biconditional. In line 7 GAZER recognizes that the hypothesis and conclusion are already in a common currency, and so plans to apply some rewrite rule to both sides simultaneously. The only applicable rule is the definition of $+$. This produces line 8, to which no other inference rules apply so gazing is again applied. Here the planner again sees that the hypothesis and conclusion are in a common currency, but this time recognizes that the conjecture would be proved if it could show that $\cup$ were commutative. Since this fact is already known, it applies the commutativity rule to one of the formulae (the hypothesis) and then the goal is proved, in line 9 by matching. Note that in line 7 gazing does recognize that the goal could be proved if $+$ were known to be commutative, but the prover does not already know this fact.

The main reason for the length of the RUT proof compared to that of the proof produced by

Conjecture: $(\forall x.(\forall y.((x + y) = (y + x))))$

(6) $(t1 \in ((sf1 - sf2) \cup (sf2 - sf1))) \vdash (t1 \in ((sf1 - sf2) \cup (sf2 - sf1)))$ ()

(5) $(t1 \in ((sf1 - sf2) \cup (sf2 - sf1))) \vdash (t1 \in ((sf2 - sf1) \cup (sf1 - sf2)))$ **(6)**

(4) $(t1 \in (sf1 + sf2)) \vdash (t1 \in (sf2 + sf1))$ **(5)**

(2) $\top \vdash ((t1 \in (sf1 + sf2)) \rightarrow (t1 \in (sf2 + sf1)))$ **(4)**

(9) $(t1 \in ((sf2 - sf1) \cup (sf1 - sf2))) \vdash (t1 \in ((sf2 - sf1) \cup (sf1 - sf2)))$ ()

(8) $(t1 \in ((sf2 - sf1) \cup (sf1 - sf2))) \vdash (t1 \in ((sf1 - sf2) \cup (sf2 - sf1)))$ **(9)**

(7) $(t1 \in (sf2 + sf1)) \vdash (t1 \in (sf1 + sf2))$ **(8)**

(3) $\top \vdash ((t1 \in (sf2 + sf1)) \rightarrow (t1 \in (sf1 + sf2)))$ **(7)**

(1) $\top \vdash (((t1 \in (sf1 + sf2)) \rightarrow (t1 \in (sf2 + sf1))) \wedge$
$((t1 \in (sf2 + sf1)) \rightarrow (t1 \in (sf1 + sf2))))$ **(2,3)**

(0) $\top \vdash ((sf1 + sf2) = (sf2 + sf1))$ **(1)**

Note: $t1 = sig1.sf1((sf2 + sf1),(sf1 + sf2))$

Figure B-5: Proof of conjecture 35, produced by GAZER

.
.
.

(65) $\top \vdash ((((t1 \in sf2) \wedge (\neg (t1 \in sf1))) \vee ((t1 \in sf1) \wedge (\neg (t1 \in sf2)))) \rightarrow$
$((t1 \in sf1) \wedge (\neg (t1 \in sf2)))) \vee ((((t1 \in sf2) \wedge (\neg (t1 \in sf1))) \vee$
$((t1 \in sf1) \wedge (\neg (t1 \in sf2)))) \rightarrow$
$((t1 \in sf2) \wedge (\neg (t1 \in sf1))))$ **(66)**

(4) $\top \vdash ((t1 \in ((sf2 - sf1) \cup (sf1 - sf2))) \rightarrow$
$(t1 \in ((sf1 - sf2) \cup (sf2 - sf1))))$ **(65)**

(2) $\top \vdash (((t1 \in ((sf1 - sf2) \cup (sf2 - sf1))) \rightarrow (t1 \in ((sf2 - sf1) \cup (sf1 - sf2)))) \wedge$
$((t1 \in ((sf2 - sf1) \cup (sf1 - sf2))) \rightarrow$
$(t1 \in ((sf1 - sf2) \cup (sf2 - sf1)))))$ **(3,4)**

(1) $\top \vdash (((sf1 - sf2) \cup (sf2 - sf1)) = ((sf2 - sf1) \cup (sf1 - sf2)))$ **(2)**

(0) $\top \vdash ((sf1 + sf2) = (sf2 + sf1))$ **(1)**

Note: $t1 = sig1.sf1(((sf2 - sf1) \cup (sf1 - sf2)),((sf1 - sf2) \cup (sf2 - sf1)))$

Figure B-6: Partial proof of conjecture 35, produced by RUT

GAZER is that RUT expands the definitions of every function that appears in the goal as soon as it is able. This leads to a very complex formula which must be manipulated in order to

prove the conjecture. The very first thing that RUT does to the goal is to expand the definition of $+$ to produce line 1. This is possible, as the definition of $+$ can be applied whatever the predicate dominating the symbol. Then, to produce line 2, RUT expands the definition of $=$. This gives rise to a conjunction that is split to produce lines 3 and 4. Now that the predicate is $\in$, and the functions $-$ and $\cup$ are present, these are eliminated by **Reduce** using their definitions. The result is the formula in line 65, which it takes the remainder of the proof to show because of the complexity of the formula.

B.5. Conclusion

In this appendix I have presented the proofs of three conjectures produced by the Voyeur theorem prover. For each conjecture I have shown the proof produced by GAZER and that produced by RUT. In each case, the use of the gazing inference rule simplifies the proof produced, although GAZER does not always find a proof as quickly as RUT.

The proofs of the remaining conjectures presented in appendix A are available from the author on request.

References

[Andrews 80] Andrews, P.B.
Transforming Matings into Natural Deduction Proofs.
In Bibel,W. and Kowalski, R. (editors), *Proc. 5th Conference on Automated Deduction*. Springer-Verlag, 1980.

[Andrews 81] Andrews, P.B.
Theorem Proving via General Matings.
Association for Computing Machinery 28(2), April, 1981.

[Bibel 80] Bibel, W.
The Complete Theoretical Basis for the Systematic Proof Method.
Technical Report Bericht ATP-6-XII-80, Institut fur Informatik, TU Munchen, 1980.

[Bibel 81a] Bibel, W.
On Matrices with Connections.
Association for Computing Machinery 28(4), October, 1981.

[Bibel 81b] Bibel, W.
Matings in Matrices.
In *Proc. German Workshop on AI.* Springer Verlag, 1981.

[Bibel 82a] Bibel, W.
Automatic Theorem Proving.
Vieweg Verlag, 1982.

[Bibel 82b] Bibel, W.
A Comparative Study of Some Proof Procedures.
Artificial Intelligence 18:269-293, May, 1982.

[Bledsoe 77a] Bledsoe, W.W.
Non-Resolution Theorem Proving.
Artificial Intelligence 9, 1977.

[Bledsoe 77b] Bledsoe, W.W.
A Maximal Method for Set Variables in Automatic Theorem Proving.
Technical Report ATP-33, University of Texas, February, 1977.

[Bledsoe 83] Bledsoe, W.W.
The UT Interactive Prover.
Memo ATP-17B, Mathematics Department, University of Texas, April, 1983.

[Bledsoe & Tyson 75a]
Bledsoe, W.W. and Tyson, M.
The UT interactive Prover.
Memo ATP-17, Mathematics Department, University of Texas, May, 1975.

[Bledsoe & Tyson 75b]
Bledsoe, W.W. and Tyson, M.
Typing and Proof by Cases in Program Verification.
Technical Report ATP-15, University of Texas, May, 1975.

[Bledsoe & Tyson 78]
Bledsoe, W.W. and Tyson, M.
The UT Interactive Prover.
Memo ATP-17A, Mathematics Department, University of Texas, June, 1978.

[Bledsoe *et al* 79] Bledsoe, W.W., Bruell, P., Shostak, R.
A Prover for General Inequalities.
Technical Report ATP-40A, University of Texas, February, 1979.

[Boyer & Moore 79]
Boyer, R.S. and Moore, J S.
ACM Monograph Series: A Computational Logic.
Academic Press, 1979.

[Brown 78] Brown, F.M.
Towards the Automation of Set Theory and its Logic.
Artificial Intelligence (10), 1978.

[Bundy 73] Bundy, A.
Doing Arithmetic With Diagrams.
In *Proceedings of the Third IJCAI*, pages 56-65. IJCAI, 1973.

[Bundy 83a] Bundy, A.
The Computer Modelling of Mathematical Reasoning.
Academic Press, 1983.

[Bundy 83b] Bundy, A.
Finding a Common Currency - A New Proof Plan.
January, 1983.
Internal Note 159, Department of Artificial Intelligence, University of Edinburgh.

[Chang & Lee 73]
Chang, C-L. and Lee, R. C-T.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, 1973.

[Church 36] Church, A.
An Unsolvable Problem of Elementary Number Theory.
American Journal of Mathematics 58, 1936.

[Cvetkovic & Pevac 83]
Cvetkovic, D. and Pevac, I.
Man-Machine Theorem Proving in Graph Theory.
Technical Report, University of Belgrade, 1983.

[Fikes & Nilsson 71]
Fikes, R.E. and Nilsson, N.J.
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.
Artificial Intelligence 2:189-208, 1971.

[Fikes *et al* 72] Fikes, R.E., Hart, P.E. and Nilsson, N.J.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence 3:251-288, 1972.

[Gelernter 59] Gelernter, H.
Realization of a Geometry Theorem Proving Machine.
In *Proceedings International Conference on Information Processing*, pages 273-282. UNESCO, 1959.

[Godel 31] Godel, K.
On Formally Undecidable Propositions of Principia Mathematica and Related Systems.
Monatshefte fur Mathematik und Physik 38, 1931.

[Halmos 60] Halmos, P.
Naive Set Theory.
Van Nostrand, 1960.

[Kleene 67] Kleene, S.C.
Mathematical Logic.
John Wiley and Sons, Inc., 1967.

[Knuth 73] Knuth, Donald E.
The Art of Computer Programming.
Addison Wesley, 1973.

[Nevins 74] Nevins, A.J.
A Human Oriented Logic for Automatic Theorem Proving.
Journal of the ACM 4:606-621, 1974.

[Nevins 75a] Nevins, A.J.
Plane Geometry Theorem Proving Using Forward Chaining.
Artificial Intelligence 6:1-23, 1975.

[Nevins 75b] Nevins, A.J.
A Relaxation Approach to Splitting in an Automatic Theorem Prover.
Artificial Intelligence 6:25-39, 1975.

[Pastre 77] Pastre, D.
Automatic Theorem Proving in Set Theory.
Technical Report, University of Paris (VI), 1977.

[Plaisted 80] Plaisted, D.A.
Abstraction Mappings in Mechanical Theorem Proving.
In Bibel, W. and Kowalski, R. (editor), *5th CADE*, pages 264-280. CADE, 1980.

[Plaisted 86] Plaisted, D.A.
Abstraction Using Generalization Functions.
In Siekmann, J (editor), *8th CADE*, pages 365-376. CADE, 1986.

[Plummer 84] Plummer, D.
RUT: Reconstructed UT Theorem Prover.
Working Paper 165, Department of Artificial Intelligence, Edinburgh, September, 1984.

[Plummer 85a] Plummer, D.
An Investigation and Rational Reconstruction of the UT Theorem Prover.
Research Paper 256, Dept. of Artificial Intelligence, Edinburgh, May, 1985.

[Plummer 85b] Plummer, D.
Gazing: Using the Structure of the Theory in Theorem Proving.
Working Paper 180, Department of Artificial Intelligence, Edinburgh, May, 1985.

[Plummer & Bundy 84]
Plummer, D. and Bundy, A.
Gazing: Identifying Potentially Useful Inferences.
Working Paper 160, Department of Artificial Intelligence, Edinburgh, Feburary, 1984.

[Quine 69] Quine, W.V.O.
Set Theory and its Logic.
Oxford University Press, 1969.

[Reiter 73] Reiter, R.
A Semantically Guided Deductive System for Automatic Theorem Proving.
In *Proceedings of the 3rd IJCAI*, pages 41-46. International Joint Conference on Artificial Intelligence, 1973.

[Reiter 76] Reiter, R.
A Semantically Guided Deductive System for Automatic Theorem Proving.
IEEE Transactions on Computers :328-334, April, 1976.

[Robinson 65] Robinson, J.A.
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM 12, 1965.

[Sacerdoti 74] Sacerdoti, E.D.
Planning in a hierarchy of abstraction spaces.
Artificial Intelligence 5:115-135, 1974.

[Sacerdoti 77] Sacerdoti, E.D.
Artificial Intelligence Series: A Structure for Plans and Behavior.
Elsevier Computer Science Library, 1977.

[Schmidt 83] Schmidt, D.
Natural Deduction Theorem Proving in Set Theory.
Technical Report, Computer Sciences Dept., University of Edinburgh, July, 1983.

[Sigler 66] Sigler, L.E.
Van Nostrand Mathematical Studies: Exercises in Set Theory.
Van Nostrand, 1966.

[Stickel 85] Stickel, M.E.
Automated Deduction by Theory Resolution.
In *Proceedings of the AAAI-85 National Conference on AI*, pages 1181-1186. AAAI, 1985.

[Suppes 57] Suppes, S.
The University Series in Undergraduate Mathematics: Introduction to Logic.
D. Van Nostrand Company, Inc., 1957.

[Tennant 78] Tennant, N.
Natural Logic.
Edinburgh University Press, Edinburgh, 1978.

[Wallen 86] **Wallen, L.A.**
Generating Connection Calculi from Tableaux and Sequent Based Proof Systems.
In Cohn, A.G. and Thomas, J.R. (editors), *Artificial Intelligence and its Applications*. John Wiley and Sons Limited, 1986.

Index of Definitions