THEOREM PROVING :
RESOLUTION, NON-RESOLUTION AND BY ABSTRACTION

Dissertation submitted to the Jawaharlal
Nehru University in partial fulfilment
for the award of the degree of
MASTER OF PHILOSOPHY

PRIYADARSHI TRIPATHY

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067
1982

C E R T I F I C A T E

This dissertation entitled 'Theorem Proving: Resolution, Non-resolution and By Abstraction' embodies work carried out at the School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi- 110 067.

This work is original and has not been submitted in part or full for any other degree or diploma of any university.

Priyadarshi Tripathy
(PRIYADARSHI TRIPATHY)
STUDENT

N.P. Mukherjee
PROF. N.P. MUKHERJEE
DEAN

R. Sadananda.
DR. R. SADANANDA
SUPERVISOR

ACKNOWLEDGEMENTS

I express my deep sense of gratitude to Dr.R. Sadananda for his unstinting guidance, constant encouragement and inspiration, constructive criticisms and critical comments.

I would also like to express my sincere thanks to Prof. N.P. Mukherjee, Dean, School of Computer & Systems Sciences, Jawaharlal Nehru University, for giving all facilities for producing this dissertation.

I feel it important to record my thanks to the Library staff for their co-operation and help.

I also thank the staff members and my colleagues of the School and specially to my class mate Sh.S.K. Sinha for his encouragements during my years in Delhi.

My special thanks are also due to Mr. K. Chand for his hardship in typing the dissertation.

The financial aid from C.S.I.R., India, in the form of J.R.F., is thankfully acknowledged.

Priyadarshi Tripathy

(PRIYADARSHI TRIPATHY)

CONTENTS

CHAPTER I

Introduction

Intelligence [49] is a meaningless concept in and of itself. It requires a frame of reference, a specification of a domain of thought and action, in order to make it meaningful. Now the question arises "are we intelligent enough to understand intelligence?" One approach to answer this question is "Artificial Intelligence", the field of computer science that studies how machine can be made to act intelligently. Artificial Intelligence is the ability of the machine to do things that people would say it requires intelligence. Artificial Intelligence research is an attempt to discover and describe aspect of human intelligence that can be simulated by machines. At present there are machines that can do many things. For example:

(a) Play games of strategy (e.g., Chess, Checkers) and (in Checkers) learn to play better than people.
(b) Learn to recognise visual or auditory patterns.
(c) Find proof for mathematical theorems.
(d) Solve certain, well-formulated kind of problems.
(e) Process information expressed in human language.

The extent to which machine can do these things independently is still limited.

Now the question arises "can a machine think?" [46]

According to Feigenbaum [19]

> "No - if one defines thinking as an activity peculiarly and exclusively human. Any such behaviour in machine, therefore, would have to be called thinking like behaviour.
>
> No - if one postulates that there is something in the essence of thinking which is inscrutable, mysterious, mystical.
>
> Yes - if one admits that the question is to be answered by experience and observation, comparing the behaviour of the computer with that of human beings to which the term "thinking" is generally applied".

Next the question arises what is meant by human intelligence and what is the difference between human and Artificial Intelligence? According to S. Watanabe [47] :

> "Artificial Intelligence functions in terms

> of abstract symbol and human intelligence functions in terms of paradigmatic symbol. A paradigmatic symbol is a particular object, on the mental representation thereof, capable of evoking a whole yet flexible field-objects. It is not an object qua object but object qua agent of elicitation. Thinking in terms of paradigmatic symbols needs no additional rules of interpretation, while thinking in terms of abstract symbol requires an adhoc rule of interpretation to have any relation with actual behavioural operation. Paradigmatic symbols have their place in the life oriented internal value structure, while abstract symbol has no relation with value. Abstract symbols cannot make inductive inference work without human work".

Evidence concerning human intelligence can be obtained from major sources:

(a) History

(b) Introspection

(c) Social Sciences

(d) Biological Sciences.

Next we will discuss what are the limits of Artificial Intelligence research. According to Arner [1] :

> "It is still irrelevant whether or not there may exist some upper-bound above which machine cannot go in this continuam. Even if such a boundary exist, there is no evidence that it is located close to the position occupied by today's machine".

According to S. Watanabe [48] :

> ".....If somebody asks the reviewer whether he believes that computers will be unable to perform certain intelligent behaviour of which human are capable, the answer will be 'yes', provided appropriate definition are given to the terms computer and intelligent.....".

In 1979 H.L. Dreyfus published a book what computers cannot do. In this book he discussed the limit of Artificial Intelligence.

Dreyfus's two major arguments against an optimistic view on Artificial Intelligence are - (a) failure of Artificial Intelligence during the period 1957-1967 to achieve the initial

premises, and during the period 1972-1977 facing the problem of knowledge representation (b) wrong assumptions underlying the optimistic view of Artificial Intelligence.

He explains the reason "why in the face of increasing difficulties, the workers in Artificial Intelligence show such untroubled confidence". [18] . He mentions four basic assumptions (Biological assumption, Psychological assumption, Epistemological assumption and Ontological assumption) underlying various attempts towards Artificial Intelligence and refutes them one by one. He offers an alternative view base on "phenomenology". Then he mentions three major aspects of all intelligent behaviour (The role of body in organising and unifying our experience of objects, the role of the 'situation' in providing a background against which behaviour can be orderly without being rule-like, and the role of human purpose and needs that are recognised as relevant and accessible) that are revealed by the phenomenological view point.

Let us come to the classification of Artificial Intelligence. Sir James Lighthill (1973) classified Artificial Intelligence activities into three categories [11] :

Category A

Standing for advanced automation, in which category

he placed optical character recognition, automatic theorem proving, inference and chemical structure from mass spectometry and other data, machine translation, product sign and assembly, problem solving, decision making etc.

Category B.

Building robots - mimicking some special function that are highly developed in man; eye-hand co-ordinate, use of natural language, 'common sense' problem solving within some limited universe of discourse such as games and puzzals.

Category C

Computer based central nervous system (CNS) research including various parts of CNS, scene analysis, memory, biological basis of learning, psycholinguistics etc.

He regards progress in A and C as slow but definite, but B has no trumphs major or minor to show for it. Further he holds that work in A and C is more defensible from the view points of its own stated objectives, but thinks that activity in B will trail off to nothing due to lack of success.

Here we are only concerned with automatic theorem proving. Automatic theorem proving is an important subject in Artificial Intelligence. It has long man's ambition to find a general decision procedure to prove theorems. This desire clearly dates back to Leibniz (1646-1716); it was revived by Peano around the turn of the century and by Hilbert's school in 1920s. A very important theorem was proved by Herbrand in 1930; he proposed a mechanical method to prove theorems. Unfortunately, his method was very difficult to apply, since it was extremely time consuming to carry out by hand. With the invention of digital computers logicians regained interest in mechanical theorem proving. In 1960, Herbrand's procedure was implemented by Gilmore on a digital computer, followed shortly by a more efficient procedure proposed by Davis and Putnam.

A major break through in automatic theorem proving was made by J.A. Robinson in 1965; he developed a single inference rule, the resolution principle, which was shown to be highly efficient and very easily implemented on computers. Since then, many improvements of the resolution principle have been made. Automatic theorem proving has been applied to many areas, such as program analysis, program synthesis, deductive question-

answering systems, problem-solving system and robot technology. The number of application keeps growing.

In our dissertation we are concerned with automatic theorem proving. There are two approaches to automated theorem proving those are proof finding and consequence finding. A proof finding program attempts to find a proof for a certain given theorem. A consequence finding program is given some axioms and then tries to deduce consequences from the axioms and to select 'interesting' consequences.

We have discussed resolution principle and also automatic theorem proving which includes non-resolution theorem proving in Chapter II.

In Chapter III we have generalised the limit heuristic and some other concepts of Bledsoe et al. [7] so that it enables to give the proof of limit theorems by using resolution and non-resolution principle.

In Chapter IV we have discussed about completely non-clausal theorem proving [29] and applied abstraction mapping to it.

In conclusion we give comment on this work.

CHAPTER II

AUTOMATIC THEOREM PROVING

Automatic theorem proving was born in the early 1930s with the work of Herbrand. In that approach, instead of proving a formula valid, he proved that negation of the formula was inconsistent, but this approach did not get much interest until high speed digital computers were developed. Earlier work by Newell, Simon, Shaw and Gelernter in the middle and late 1950s emphasized the heuristic approach, but the weight soon shifted to various syntactic methods culminating in a large effort on resolution type system in the last half of 1960s. It was about 1970 when considerable interest was revived in heuristic methods and the use of human supplied, domain dependent, knowledge.

First we will duscuss those efforts in automatic theorem proving, during the past few years emphasized on resolution, and then we will go to the techniques in automatic theorem proving other than resolution.

2.1 Resolution Theorem Proving

Herbrand method required to generate a very large number of ground clauses, it is abandoned in favour of two

alternative approaches. One of these approaches is Robinson's resolution principle [41] . The other is to use the idea proposed by Prawitz [39,40] . For the last decade, the resolution principle has got most of the attention of researchers, and has been playing a dominant role in the field. Compared to resolution, Prawitz's idea is relatively unexplored.

P.C. Gilmore (1960), H. Wang (1965) as well as M. Davis and H. Putnam (1960), are among those who programmed a computer to find proofs in first order predicate calculus.

Essentially what Davis and Putnam did were the following ideas:

(a) A formula of the first order logic can be transformed into a prenex normal form where the matrix contains no quantifiers and the prefix is a sequence of quantifiers.

(b) The matrix, since it does not contain quantifiers, can be transferred into a conjuctive normal form.

(c) Without effecting the inconsistency property, the existential quantifiers in the prefix can be eliminated by skolem function.

After these programs had been written, J.A. Robinson developed an inference rule which he calls resolution principle. Roughly speaking, the resolution principle draws the most general possible conclusion from two given statements; the conclusion and the two statements generally contains variables. We will now give a process to infer a new clause from two other clauses (called parent clauses). Let $\{L_i\}$ and $\{M_i\}$ be the parent clauses and assume that variables occuring in $\{M_i\}$ donot occur in $\{L_i\}$. Let $\{l_i\}$ and $\{m_i\}$ are two subsets of $\{L_i\}$ and $\{M_i\}$ respectively such that a most general unifier exists for the set $\{\sim l_i\} \cup \{m_i\}$. Then we say that the two clauses $\{L_i\}$ and $\{M_i\}$ resolve and the new clause

$$[\{L_i\} - \{l_i\}]_\lambda \cup [\{M_i\} - \{m_i\}]_\lambda \quad \text{is the}$$

resolvant of two clause.

The resolvent is an inferred clause, and the process of forming a resolvent from two "Parent" clause is called a resolution. For further study on resolution principle see [15, 20, 25, 27, 33, 43] .

J.A. Robinson was the first to prove that the resolution principle is effective and sound, and for proof finding, is complete. R. Lee (1967) was the first to prove

that resolution is complete for consequence finding. That the resolution principle is effective means that one can write a computer program, which, in a finite number of steps, will find the factors of any clause and the resolvents of any two clauses. That the resolution principle is sound means that a clause logically implies each of its factors and that two clauses, taken together, logically imply each of their resolvents. The resolution principle is complete for proof finding in the sense of the following theorem, first proved by J.A. Robinson [41] :

> If a finite set of clauses unsatisfiable, then a contradiction can be found in a finite number of applications after resolution principle.

The resolution principle is complete for consequence finding in the sense of the following theorem, first proved by R. Lee [41] :

> If a clause C is a consequence of a finite nonempty set of clauses, then a clause T can be found in a finite number of application of the resolution principle such that C is an immediate consequence of T alone.

Several researchers have strengthened these theorems by showing that certain restricted forms of the resolution principle are still complete. Several basic strategies based on resolution techniques have appeared. Of these, the set of support [51] , hyperresolution [42] , unit resolution [12,15] and paramodulation [50] strategies seems to be most notable. For these techniques see Chang and Lee [15] , Loveland [27] and Nilson's review article [34] .

All of these strategies are essentially algorithms for sharing the unsatisfiability of a statement S expressed in the first order predicate calculus. S is normally formed as the denial that the theorem to be proved follows from a given set of axioms. In most cases the Herbrand universe of S is infinite. For the algorithm mentioned above, this fact is a major cause of failure to prove theorems of even moderate complexity. Ross A. Overbeck [35] proposes an algorithm which has been used successfully to prove theorems of moderate difficulty. The basic technique is to generate from a given statement S a sequence of statements $S_0, S_1, S_2, \ldots\ldots\ldots$ with the following properties:

(a) If there exists an i such that S_i is unsatisfiable, then S is unsatisfiable.

(b) If S is unsatisfiable, then there exist an i, such that S_i is unsatisfiable.

(c) For all i the Herbrand universe of S_i is finite. Hence, for each i the satisfiability of S_i is decidable.

One can then generate successive elements of the sequence S_i testing each element for satisfiability. By applying this algorithm he proves the following theorem:

Let G be a group such that $x^{-1} = x^2$ for all $x \in G$. If h is defined as $h(x,y) = xyx^{-1}y^{-1}$, then for all $x,y \in G$, $h(h(x,y),y) = e$(the identity).

In 1978 Malcolm C. Harrison and Norman Rubin [21] generalised resolution principle of theorem proving which permits the hierarchical organisation within the theorem prover. This generalization corresponds to an extension of the usual unification procedure. This generalisation permits us to use members of a specified set U of clauses as rules of inference rather than clauses to be inferred form. He describe two types of resolution U-g-resolution and U-e-g-resolution. We will discuss these two types of generalized resolution.

Consider two clause G and H of the form

A U $\{L(x_1, x_2, \ldots\ldots x_l)\}$ and B U $\{M(y_1, y_2, \ldots\ldots y_m)\}$.

Suppose clause C of the form D U $\{\overline{L}(w_1, \ldots\ldots w_l)\}$U $\{\overline{M}(z_1, \ldots z_m)\}$ can be inferred from a set of clause U, and there is a most general simultaneous unifier[1] λ of the pairs (x_i, w_i), $i=1,2,\ldots\ldots l$ and (y_j, z_j), $j=1,2,\ldots\ldots m$. Then the clause

$$[A\lambda - L(x_1, \ldots x_l)\lambda] \quad U \; D\lambda \; U \quad [B\lambda - M(y_1, \ldots y_l)\lambda]$$

is a binary U-generalized-resolvent of G and H. We will refer to clause C as the unifying clause and to the L and M literals of C as the linking literals. Furthermore, if clause C' of the form $\overline{L}(w_1, \ldots w_l)$ can be inferred from U, and there is an mgsu λ of the pair (x_i, w_i), $i=1, \ldots l$ then the clause

$A\lambda - \{L(x_1, \ldots\ldots x_l)\lambda\}$ is a Unary U-g-resolvent of G

U-g-resolution is somewhat similar to Dixon's

---

1. We define a simultaneous unifier (su) of a set of pairs (x_i, y_i), $i=1,2,\ldots n$ to be a substitution λ which satisfies $x_i \lambda = y_i \lambda$, $i=1, \ldots n$. If there exist an su for a set of pairs, the set is called simultaneously unifiable (which we also signify by su). We use mgu to stand for most general unifier. If the pairs (x_i, y_i) are su, then most general simultaneous unifier (mgsu) in any such that for any su μ there exists a σ such that $\lambda\sigma = \mu$.

Z-resolution [17] . However Z-resolution attempts to unify two literals using a subset Z of clauses, but constrains the proof of equivalence to involve no more than two resolution. Furthermore, the clauses in Z must contain exactly two literals each, and are constrained so that no variable occurs more than once in each literal, and resolution between two clauses of Z gives a tautology.

Consider two clauses G and H of the form $A \cup \{L(x_1,\ldots x_1)\}$ and $B \cup \{\bar{L}(y_1,\ldots y_1)\}$. Suppose clause $D_1 \cup \{w_1 = z_1\}$ can be inferred from the set U of clauses such that there is an mgsu λ of the pairs (x_1,w_1) and (y_1,z_1), then the clause

$$[A\lambda - L(x_1,..x_1)\lambda\}] \cup \bigcup_1 D_1\lambda \cup [B\lambda - \{\bar{L}(y_1,..y_1)\lambda\}]$$

is a U-equality-generalised resolvent of G and $\bar{H}$.

The most similar approach to U-e-g-resolution reported in the literature in Morri's E-resolution [28] . The main difference appears to be that E-resolution uses any clause to prove terms equal, using paramodulation into one until the other is obtained. In U-e-g-resolution only clauses in U need to be used, and there is no particular commitment to paramodulation.

Some of the most interesting and potential useful idea in resolution based, automated theorem proving was appeared were related to input and unit refutation. In an input refutation of a set S of clauses, one parent of every resolvent is a resolvent is a clause in S, where as in a unit refutation, one parent is a unit i.e., a clause with but one literal. Chang [12,15] has shown that a set S has a unit refutation (by resolution and factoring) iff it has an input refutation. Later [22] it was shown that an unsatisfiable set S in which each clause has at most one positive literal i.e., a Horn set, has a unit and hence an input refutation. Furthermore Kuehner [26] has shown that for Horn set, unit and input refutation can be further restricted, to become the so-called SPU-refutation(selective positive unit) and SNL-refutation (selective negative linear), respectively. In an SPU-refutation each clause is ordered with the positive literal, if there is one, on the right, and resolution is permitted only when one parent is a positive unit and only one the leftmost literal of each clause. In as SNL-refutation each clause is similarly ordered, resolution is permitted only if one parent is an input clause and the other parent is the result of the previous resolution, the non input parent must be negative, and in this parent it is only the leftmost literal which can be resolved

upon. Furthermore, in the resolvent, the negative literals of the input parent if any, take the place of the literal resolved upon in the negative parent. It can also be shown that no factoring is necessary in these refutation.

There are many theorems, whose clause formulation will not be a Horn set. In 1976 G.E. Peterson [37] extends the previous result in a natural way to any unsatisfiable set of clauses.

In 1981 David A. Plaisted [38] defines a class of mapping called abstraction. These function map a set S of clauses onto a possibly simpler set T of clauses also resolution proofs from S map onto possibly simpler resolution proofs from T. In order to search for a proof of a clause C from S, it suffices to search for a proof from T and attempt to invert the abstraction mapping to obtain a proof of C from S. First he presents an incomplete theorem proving strategy based on abstractions. This strategy can guide the search for a proof of a particular consequence of a set of clauses, as well as guiding the search for a proof that a set of clauses is inconsistent. After that, inference rules related to resolution are discussed in which 'm-clauses', which are multisets of literals are introduced. That is, with each

literal in the m-clause, a count is kept of 'how many times it occurs' in the m-clause. A version of resolution called m-resolution is defined. In addition, m-abstractions are defined. The advantage of m-abstraction is that they preserve much more the structure of a proof that do ordinary abstraction. As a consequence, there are simple, complete theorem proving strategies based on m-abstraction. Also, there are strategies that use more than one m-abstraction, at the same time. All the strategies that are based on m-abstraction are complete. Bounded m-clauses are discussed next. They are m-clauses in which less information about the number of occurrences of a literal in a clause is kept. Abstraction and complete theorem proving strategy based on bounded m-clauses are presented.

Recently Neil.V.Murray [29] considers a version of the first-order predicate calculus in which statements are not required to be in the clause form as required by standard resolution theorem provers. Only requirement is that the well-formed formulas (wff) to be quantifier free and have distinct variables and all variables are implicitly universally quantified. We have discussed this in great detail in the Chapter IV, where we apply abstraction mapping to it.

Next we will discuss the idea proposed by Prawitz.

Given a set of clauses, Prawitz's idea is that, instead of generating the ground instances of clauses of S in some arbitrarily defined order, one should find by calculations the values that, when substituted for variables in S, give an unsatisfiable set of ground instances. Essentially, Prawitz's idea is based upon the observation that a set of clause is unsatisfiable iff there is a set of M of copies (variants) of clauses in S and a ground substitution θ such that $M\theta$ is truth-functionally unsatisfiable. θ is known as a solution of M. This observation is actually Herbrand's theorem in different form. Many methods [2,13,14,40] have been proposed to find such M and θ. In [14] the approach is as follows:

(a) First, find a connection graph for a set S of clauses;
(b) Change the connection graph to a directed graph;
(c) From the directed graph, obtain a set of rewriting rules;
(d) Use the rewriting rules to generate a refutation plan;
(e) Finally, use a unification algorithm to check whether the plan is acceptable or not.

The main difference between this approach and the others is that in this case first to generate a plan, and then perform unification at the last step of a proof.

2.2 Non-Resolution Theorem Proving

In this section we will mention those efforts in automatic theorem proving, during the past few years, which have emphasized techniques other than resolution [5] and briefly discuss them. These includes:

(a) Knowledge Base
(b) Reductions (rewrite rules)
(c) Algebric Simplification
(d) Built-in Inequalities
(e) Natural System
(f) Forward chaining
(g) Overdirector
(h) Types
(i) Advice
(j) Procedures
(k) Analogy

2.2.1 Knowledge Base

We store information in a knowledge base (or data base), processed that information to obtain other information, and interrogate the data base when necessary to answer questions. The central idea here is that facts are stored about "objects" rather than predicate. For example the hypothesis open (A_0)

would be stored with "open" as a property of "A_0" rather than with "A_0" as a property of open. The graph provers of Bundy[10], Ballantyne and Benett [3] , and theorem provers of Bledsoe [4] use such a database.

2.2.2 Reductions

A reduction is a rewrite rule,

$$A \longrightarrow B$$

For example, the rule

$$t \in (A \cap B) \longrightarrow t \in A \wedge t \in B$$

required to change all subformulas of the form $t \in (A \cap B)$ into the form $t \in A \wedge t \in B$. Such rules are semantic; their inclusion, and their use is not based upon their syntactic structure but on their meaning.

Some reduction rules are given below:

IN	OUT
$t \in (A \cap B)$	$t \in A \wedge t \in B$
$t \in (A \cup B)$	$t \in A \vee t \in B$
$t \in x{:}P(x)$	$P(t)$
$t \in \text{subsets}(A)$	$t \subseteq A$
.	.
.	.

There are four types of reduction

(a) Reduction (Rewrite Rules)

(b) Conditional Reduction

(c) Controlled Definition Instantiation

(d) Complete set of Reductions

The principal workers in this area are - Ballantyne and Bledsoe [4,6] , Bledsoe et al. [7] , Nevins [30] , Slagle [44] and Pastre [36] .

2.2.3 Algebraic Simplification

There is a strong need to avoid adding field axioms for the real numbers as hypothesis to a theorem being proved, because this greatly slows proofs. The associative and commutative axioms for + and . are especially troublesome, so several efforts have been made to "Build these in". Some references to this work are Slagle and Norton [45] and Bledsoe et al. [7] .

2.2.4 Built-in Inequalities

Again we must avoid the explicit use of such axioms as the transitivity axiom

$$(x \leq y \wedge y \leq z \Rightarrow x \leq z)$$

Bledsoe et al. [7] employ "interval types" for dealing with certain inequalities, and Slagle and Norton [45] have built in axiom for handling total and partial ordering.

2.2.5 Natural System

It is most important concept of non resolution principle. Suppose we are given a goal G and a hypothesis H and we wish to show that G follows from H

or more general to find a substitution θ for which $H\theta \Longrightarrow G\theta$ is a proportionally valid formula. A set of rules is given for manupulating H and G to obtain the desired θ . For example,

$(P(a) \wedge (P(x) \Longrightarrow Q(x)) \Longrightarrow Q(a))$ has a solution

$\theta \equiv a/x$.

Now what are the advantages of the natural system? Bledsoe [5] feels that the natural systems are:

(a) Easier for human use

(b) Easier for machine use of knowledge.

Human use

1. Bring to bear knowledge from pure mathematics in the same form used there.

2. Recognize situations where such knowledge can be used.
3. Professional mathematician will want to participate.
4. Easier to design, augment, work upon.
5. Essential for man-machine interaction.

Machine use.

1. Automatically limits the search. Does not start all proofs of the theorem. (Syntactic search strategy).
2. A natural vehicle to hang heuristic, knowledge, semantic. (Semantic search strategies).
3. Easier to combine Procedures with deduction.
4. Contextual database problem. (One data base would be needed for each clause).
5. New language (PLANNER, QA4). Ease the implimentation.

We quote Arthur Nevins [30] regarding the advantage of natural systems:

"A point worthy of stress is that a deductive system is not 'simpler' merely because it employs fewer rules

of inference. A more meaningful measure of simplicity is the ease with which heuristic consideration can be absorbed into the system".

2.2.6 Forward Chaining

Forward chaining, which is the act of generating new facts from the old one i.e. one hypothesis is applied to another to obtain an additional hypothesis

Hypothesis conclusion

$$P(A) \wedge (P(x) \Longrightarrow Q(x)) \Longrightarrow C$$

$$\Downarrow$$

$$Q(A)$$

(a) Ground Forward Chaining

This procedure is performed on formulas in the hypothesis of a theorem and thereby creates new hypothesis. It is invoked when something new is added to the hypothesis, such as at the beginning of the proof or when a new hypothesis is "promoted".

Example:

$P(x_0), Q(y_0)$ and $(P(x) \wedge Q(y)) \Longrightarrow R(x,y))$ yield

$R(x_0, y_0)$

(b) Procedural Forward Chaining

In this case a procedure is invoked which manipulates items in the database (or in hypothesis) to produce new items.

The early programs of Newell, Simon and Shaw, used forward chaining. Nevins [31] ,Ballantyne [4] and Pastre [36] also used forward chaining.

Brown [9] describe theorem prover for elementary number theory using no chaining rules, forward or backward, but instead interaction between equations is based upon the use of many truth-value preserving transformation.

2.2.7 Overdirector

Every prover has a control routine which direct the search tree. See Newel, Simon and Shaw's [32] control structure is shown in Fig.1. Overdirector have the flexibility to switch from one line of attack to another, and back again, as the proof proceed, thus providing a parallel search capability This of course require a (controlled) back-up mechanism, unrestricted back-up is intolerable. A contextual database, which can be consulted by the overdirector to help it whether and how much to backup or what other line of attack to take, is an indispensable part of the prover in our mind.

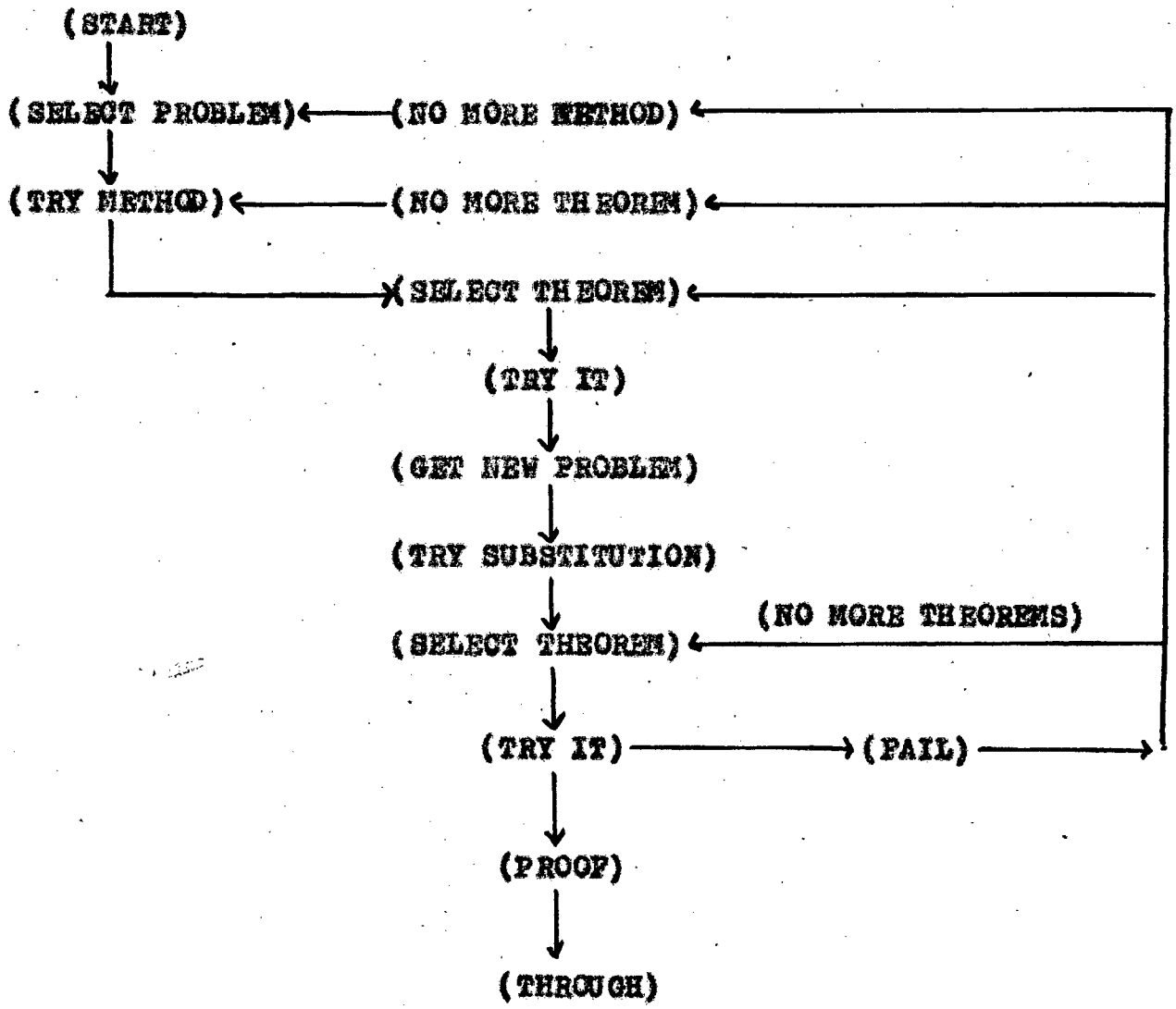

(START)
(SELECT PROBLEM)
(NO MORE METHOD)
(TRY METHOD)
(NO MORE THEOREM)
(SELECT THEOREM)
(TRY IT)
(GET NEW PROBLEM)
(TRY SUBSTITUTION)
(NO MORE THEOREMS)
(SELECT THEOREM)
(TRY IT)
(FAIL)
(PROOF)
(THROUGH)

Fig. 1.

2.2.8 Types

The concept of typing plays very important role in mathematics and computer science. Using a letter e for the identity element of a group, lower case letter x,y,z for members of the group and capital letters G,H for groups and subgroup is helpful to human in proving theorems. Similarly typing is helpful to human in proving theorems. Some examples of types are given below:

e	identity in a group
x,y,z	member of a group
G,H	groups, subgroups
P,Q	Predicate
z	Complex
ϵ,δ	Infinitesimals
r,s,t	standard real
w	Infinite large integer

2.2.9 Advices

One of the most powerful things a human can do to aid the prover is to provide "advice" for the use of a theorem or a lemma. To explain this we will give an example [5] :

An Advice Lemma

GOAL VERIFY

($\lvert A\rvert \leq \epsilon$)	(A=B+C)	1
	and	
	($\lvert B\rvert < \epsilon_1$)	2
	and	
	($\lvert C\rvert < \epsilon_2$)	3
(or other advice)	and	
	($\epsilon_1 + \epsilon_2 \leq \epsilon$)	4
	(A=B.C)	1'
	and	
	($\lvert B\rvert < \epsilon_1$)	2'
	and	
	($\lvert C\rvert < \epsilon_2$)	3'
	and	
	($\epsilon_1 + \epsilon_2 \leq \epsilon$)	4'
	or	
	(other advice)	

The proof of an example using advice lemma

Theorem

$$|a| < E^{\alpha} - 1 \wedge |b| < E^{\beta} - 1 \wedge 1+c = (a+1)(b+1) \Rightarrow |c| < E^{\alpha+\beta} - 1$$

Proof

GOAL: $|c| \leq E^{\alpha+\beta}-1$

Parts 1-4 of the advice lemma are used to concert this subgoals (i) - (iv) below:

(i) $c = B+C$

To solve this, convert the hypothesis

$(1+c)=(a+1)(b+1)$ to $c=a.b+a+b$ and substitute

$a.b/B$, $a+c/C$

(ii) $|a.b| \leq \epsilon_1$ use part 1'-4' of the advice lemma

(a) $a.b=B.C$ $a/B, b/C$

(b) $|a| \leq \epsilon_{11}$ $E^{\alpha}-1/\epsilon_{11}$

(c) $|b| \leq \epsilon_{12}$ $E^{\beta}-1/\epsilon_{12}$

(d) $(E^{\alpha}-1)(E^{\beta}-1) \leq \epsilon_1$ $() \cdot ()/\epsilon_1$

(iii) $|a+b| < \epsilon_2$ use part 1-4 of the advice lemma (again)

(a) $|a| < \epsilon_{21}$ $\quad E^{\alpha} - 1 / \epsilon_{21}$

(b) $|b| < \epsilon_{22}$ $\quad E - 1 / \epsilon_{22}$

(c) $(E^{\alpha}-1)+(E^{\beta}-1) \leq \epsilon_2$ $\quad E^{\alpha} - 1 + E^{\beta} - 1 / \epsilon_2$

(iv) $(E^{\alpha}-1).(E^{\beta}-1)+((E^{\alpha}-1)+(E^{\beta}-1)) \leq E^{\alpha+\beta}-1$ use simplification

$E^{\alpha+\beta}-1 \leq E^{\alpha+\beta}-1$ TRUE.

2.2.10 Procedure

The main concept of the procedure is to follow a plan rather than search and calculate an answer or prove a formula. Examples of procedures are: Induction, Built-in partial and total ordering, inequality, associativity etc.

2.2.11 Analogy

In proving a theorem T from a database S, a theorem prover might generate the set of clauses indicated by the area labeled S_1 (See Fig.2). Given the larger database S' (which include S as a subset), the theorem prover will usually generate the much larger set of clauses S_2 before it obtains a proof for T. In general, we want the theorem prover to have the database S' available, since there is no prior information as to which theorem it will be required to prove.

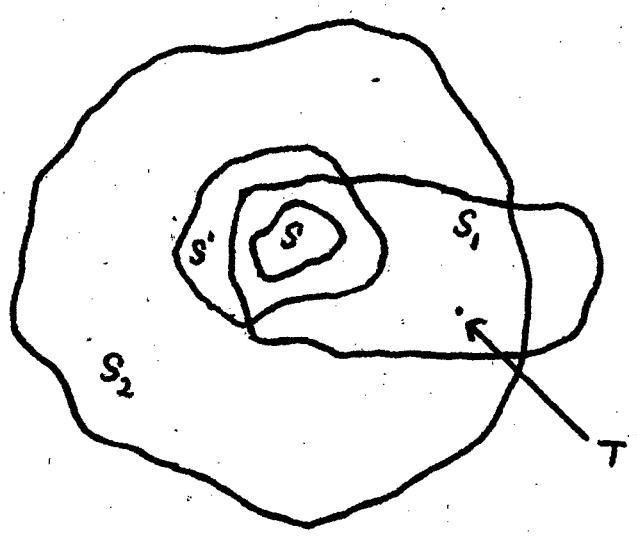


Fig. 2

But we would like to have some program that could often select, for any given theorem T, a database S from which T could be easily derived. We could attempt either to modify the theorem prover itself or to write a new program that would select database for the theorem prover. Because of the undecidability of the predicate calculus, this probability cannot be completely solved. However Kling [23] provided a partial solution to it in the form of an analogy generator that given some help, is often capable of making a good selection for the database. Kling's analogy generating program is called ZORBA - 1.

His paper showed how a proof in a group theory could be used to help to obtain a similar proof in a ring theory.

CHAPTER III

PROOFS OF LIMIT THEOREMS

3.1 Types And Inequalities

Bledsoe et al. [7] used membership types whereby the type A is assigned to x whenever it is known that $(x \in A)$.

Let $\langle a\ b\rangle$ denote the open interval from a to b. $R = \langle -\infty\ \infty\rangle$, $P = \langle 0\ \infty\rangle$, $N = \langle -\infty\ 0\rangle$.

In trying to prove

$(0 < x \Longrightarrow Q(x))$

assign the type P (or $\langle 0\ \infty\rangle$) to x and then try to prove Q(x).

For example, suppose that we are to prove

$0 < b \Longrightarrow$ SOME $x(0 < x \wedge x < b)$.....(1)

one valid approach is to solve for x in

$(0 < b \Longrightarrow 0 < x)$.....(2)

and then try to verify

$(0 < b \Longrightarrow x < b)$.....(3)

for that same x. By using maching we would get as a solution of (2) the substitution [b/x] and required $(0 < b \Rightarrow b < b)$ in (3) which is impossible. Of course (1) is unprovable without further hypothesis (or axioms) but it can be handled by the use of types (which implicitly assumes certain axioms). Our approach in proving (1) is to assign type $\langle 0\ \infty \rangle$ to b, and then try to prove

$$\text{SOME } x\ (0 < x \wedge x < b) \qquad (4)$$

we first solve

$$(0 < x) \qquad (5)$$

by assigning type $\langle 0\ \infty \rangle$ to x and then solve

$$(x < b) \qquad (6)$$

by assigning the type $\langle 0\ \ b \rangle$ to x. The resulting type of x, $\langle 0\ \ b \rangle$ was derived as the intersection of its initial type $\langle 0\ \infty \rangle$ gotten from (5), and the interval $\langle -\infty\ b \rangle$, which would have been the type gotten from (6) alone. Since this intersection is not empty it is assigned as the resulting type of x. Even though the variable x had already been 'solved for' in (5) (typed), it remains a variable in the solution of (6) and therefore would be 'solved for' again (retyped).

Types are used by SOLVE< and SOLVE> and SET-TYPES which are described below:

3.2 SOLVE<

This is for solving linear inequalities (SOLVE <AB) and chooses a variable from A or from B and attempt to solve the inequality (A<B) in terms of that variable. If this fails it then chooses another variable and tries again. Since the terms and variables of A and B may be typed, this must take into consideration such types and reset the type of the variable when the solution is obtained. In fact the answer is completely given by the new types. If it can be shown that A is less than B, then it will return the answer 'T' whether or not A and B have any variables.

In actual theorem proving process, SOLVE<, is applied to formula that have been converted to quantifier free form by the introduction of skolem expressions. Precautions are taken by SOLVE to insure that it does not solve for a variable x in terms of a skolem expression in which x occurs.

For example, consider the false statement

SOME x ALL y (y<x)

The skolem form of this is

(y x)<x.

The result of a call to (SOLVE< (y x)x) is NIL, since x occurs in skolem expression (y x).

On the other hand, the theorem

SOME x ALL y SOME z(y < x+z)

which has skolem form

(y x) < x+z

can be proved by a call to (SOLVE< (y x)(x+z)) which correctly assigns type ⟨(y x) - x ∞⟩ to z.

EXAMPLES:

	INPUT A	INPUT B	New type of x given by(SOLVE AB)
1.	x	1	⟨-∞ 1⟩
2.	x	1	⟨0 1⟩
	Type x is ⟨0 ∞⟩		
3.	0	1	(the value T is returned)
4.	x.a+c (-x+d)		$\langle -\infty \ \frac{d}{1+a} - \frac{c}{1+a} \rangle$
	Type a is ⟨0 ∞⟩		

3.3 SOLVE >

Which is same as above, the only difference is in place of ' <' we have written ' >'

3.4 SOLVE =

This is for solving linear equations. Given two arithmatic expression A and B, it selects a variable x from A or B and tries to solve the equation (A=B) in terms of x. If it succeeds, with answer y, it return the substitution [y/x] otherwise it selects another variable and tries again, returning NIL if all fail.

3.5 SET-TYPE

This is used to assigns types to certain skolem expressions. If a formula of the form $(A \in B)$ is in a conjuctive position of E (i.e. E can be expressed as $((A \in B) \wedge D$ for some D), and if A is a skolem expression which donot occur in B, then (SET-TYPE) assign the type B to A and returns D, the formula gotten by removing $(A \in B)$ from E. If A already has type C, then SET-TYPE assigns the intersection $(B \cap C)$ as the type of A, if $(B \cap C)$ is non-empty. If $(B \cap C)$ is not empty, but cannot be given specifically then the formula (intersection BC) is given as the type of A.

In a similar way, it assigns type to skolem expression which satisfy certain inequalities. For example, if E is

$$(A < 0 \wedge (B < 1 \vee C))$$

then (SET TYPE) assign type $\langle -\infty\ 0\rangle$ to A and returns $(B < 1 \vee C)$, and if E is $(A < B \wedge C)$ then (SET TYPE B) assign type $(-\infty\ B\rangle$ to A and type $\langle A\ \infty\rangle$ to B and return C. Similarly (SET-TYPE $(A \neq 0)$) can be made to assign type (Union $\langle -\infty\ 0\rangle\ \langle 0\ \infty\rangle$) to A.

3.6 SIMPLIFY

This is an alegbraic simplification concept which converts algebraic expression into a canonical form, sorts its terms, and cancels complementary terms of the form $(a+(-a))$ and $(a\ (\frac{1}{a}))$. It is used to manipulate algebraic expression.

EXAMPLES:

INPUT	OUTPUT
(a.(b+c+d))	(a.b+a.c+a.d)
$(a.\frac{1}{b}.\frac{1}{a})$	$\frac{1}{b}$
(|a+c-a| -d)	(|c| -d)

3.7 LIMIT HEURISTIC

The limit heuristic rule we have defined below and those concepts we have described are used to prove limit theorems.

(a) when trying to use a hypothesis of the form

$$|A| < B' \qquad \text{(and other hypothesis)}$$

to establish conclusion of the form $|B| < E$, first try to find a substitution σ which will allow B_σ to be expressed as non-trivial combination of A_σ, $(B=K.A+L)_\sigma$ then try to establish the new combination.

A. $(|K| < M)_\sigma$ for some M

B. $(|A| < \frac{E}{2M})_\sigma$

C. $(|L| < \frac{E}{2})_\sigma$

Such a procedure is valid because if we choose and prove A, B and C, then we would have

$$\begin{aligned} |B|_\sigma &= |K.A+L|_\sigma \\ &\leq (K.A+L)_\sigma \\ &< M.\frac{E}{2M} + \frac{E}{2} \\ &= E \end{aligned}$$

(b) In the similar fashion as above if we are trying to use a hypothesis of the form $|A| < B'$ to establish a conclusion of the form $|B| > E$, find a substitution σ which will allow B_σ to be expressed as $(B=L-K.A)_\sigma$, then try to establish the

three new conclusion

A. $(|K| < M)_\sigma$ for some M

B. $(|A| < \frac{E}{M})_\sigma$

C. $(|L| > 2E)_\sigma$

This procedure is valid because

$$|B|_\sigma = |L-K.A|_\sigma$$

$$\geq (L - K . A)_\sigma$$

$$> 2E - \frac{ME}{M}$$

$$= E$$

The above discussionbased upon the inequality

$$|x| - |y| \leq |x - y|$$

and

$$|x + y| \leq |x| + |y|$$

and uses the fact that

$$\frac{1}{2} + \frac{1}{2} = 1, \quad M.\frac{1}{M} = 1 \quad \text{for} \quad M \geq 0 \quad \text{etc.}$$

3.8 EXTRACT

Now the question is how we will get the value of K and L. The limit heuristic uses EXTRACT described below:

If there is a substitution σ for which B_σ can be expressed as a non-trivial combination of A_σ, $(B=K.A+L)_\sigma$ or $(B=L-K.A)_\sigma$ then EXTRACT RETURNS (KL σ) where σ is the most general such substitution, otherwise NIL is returned.

EXAMPLES:

1. (EXTRACT A(K.A+L)) = (KL T)

2. (EXTRACT A(L-K.A)) = (KL T)

3. (EXTRACT $(f(x)-L_1)(f(x_0)+g(x_0)-(L_1+L_2))$)

   = (1 $(g(x_0)-L_2)$ x_0/x)

4. (EXTRACT $(f(x)-L_1)(\frac{1}{f(x)}-\frac{1}{L_1})$)

   = ($-\frac{1}{f(x).L_1}$ 0 T)

A more precise description of EXTRACT is as follows:

Suppose there is a substitution σ and an expression x such that A_σ and B_σ are polynomials in x, and B is linear in x. Then there are expression a, c, b and d such that x doesnot

occur in c,a or d and A_σ and B_σ can be reexpressed as

$$A_\sigma = a.x+c$$

$$B_\sigma = b.x+d$$

and (EXTRACT AB) returns the value

$(\frac{b}{a}(d-\frac{b.c}{a})\sigma)$ If no such σ and x exist then EXTRACT return NIL.

3.9 Resolution

In this section we show how the limit heuristic and the theory of types explained above can be used in resolution. This is done by giving some additional rules for resolution. These are:

3.9.1 SET-TYPE RULE

For each unit clause of the form $(x \in A)$, where x is a skolem expression which does not occur in a, assign the type A to x. Also for each unit clause of the form $(x < a)$, where x is a skolem expression which does not occur in a, assign the type $\langle -\infty\ a\rangle$ to x. Similarly for unit clauses of the form $(b < x)$ assign type $\langle b\ \infty\rangle$ to x. In each of these cases, remove the unit clause. If x already had a type B and we are trying to assign it a new type A, then assign the type $(A \cap B)$ if it is

non-empty; if $(A \cap B)$ is empty add the empty clause (i.e., the proof is finished), if it can't be determined whether $(A \cap B)$ is empty, leave the original type as it is and donot remove the unit clause. This set type rule need only be applied at the beginning and after new unit clause is generated.

3.9.2 MEMBER RULE

For a clause of the form $DV(x \notin A)$..

(1) If x has type A then add D to the list of clauses.

(2) If x is a variable and x does not occur in A, then assign the type A to x and add D to the list of clauses.

3.9.3 TRANSITIVE RULE

When attempting to resolve two clauses of the form $((a<b) \vee A)$ and $((a' \not< c) \vee B)$, where $a_\sigma = a'_\sigma$ for some substitution σ if (SOLVE $<$ bc) yields σ', then add the resolvent $(A \vee B)_{\sigma\circ\sigma'}$ to the list of clauses.

In the similar fashion if two clauses of the form $((a>b) \vee A)$ and $((a'>b) \vee B)$ where $a_\sigma = a'_\sigma$ for some substitution σ, (SOLVE bc) yields σ' then add the resolved $(A \vee B)_{\sigma\circ\sigma'}$ to the list of clauses.

3.9.4 SOLVE > RULE

For a clause of the form DV(A > B) if (SOLVE > AB) yields the value σ then add D_σ to the list of clauses.

3.9.5 SOLVE < RULE

For a clause of the form DV(A < B) if (SOLVE < AB) yields the value σ then add D_σ to the list of clauses.

3.9.6 SUBSTRACTION RULE

When attempting to resolve two clauses of the form $((a=b) \vee A)$ and $((c \neq d) \vee B)$, if SOLVE = (a-c) (b-d) yield σ, then add $(A \vee B)_\sigma$ to the list of clauses.

3.9.7 SOLVE = RULE

For a clause of the form DV(A ≠ B) if (SOLVE = AB) yields the value σ then add D_σ to the list of clauses.

3.10 Limit Heuristic Rule

(a) When attempting to resolve two clauses of the form

1. $((\ |A| < E') \vee C_1)$

2. $((\sim|B| < E) \vee C_2)$

try to find a substitution σ which will allow B to be expressed as a non-trivial combination of A.

$(B = K.A+L)_\sigma$

and if this is possible for some substitution σ, then add the following new "resolvent" clause to the clause list

3. $((\sim|K| < M) \vee (\sim|A| < \frac{E}{2T}) \vee (|L| < \frac{E}{2}) \vee C_1 \vee C_2)$

where M is a new variable with type $\langle 0\ \infty \rangle$

(b) In the similar fashion to resolve two clauses of the form

1. $((|A| < E') \vee C_1$

2. $((\sim|B| > E) \vee C_2)$ or $(\sim|\frac{1}{B}| < \frac{1}{E}) \vee C_2$

try to find out a substitution σ which will allow B to be expressed as a non trivial combination of A,

$(B = L \cdot K \cdot A)_\sigma$

and if this is possible for some substitution σ then add the following "resolvent" clause to the clause list

3. $(|K| \lhd M) \vee (|A| \lhd \frac{E}{M}) \vee (|L| \rhd 2E)$

where M is a new variable with type $\langle 0\ \infty \rangle$

EXAMPLE 1

Given the clauses

1. $|f(x) - L| < E_1$
2. $|f(x)| \rhd E$

where E_1, E, x are variables and E_1, E has type $\langle 0\ \infty\rangle$

Rule 3.10(b) is used on clause (1) and (2)

EXTRACT ((f(x)-L)(f(x)=EXTRACT((f(x)-L)(f(x)-L+L)

yield $[-1 \quad L]$; and the following resolvent is produced

3. $(\ |1| \prec M) \vee (\ |f(x)-L| \prec \frac{E}{M})\ (|L| \succ 2E)$

where M is a new variable of type $\langle 0\ \infty\rangle$

4. $(\ |f(x)-L| \prec \frac{E}{M}) \vee (\ |L| \succ 2E)$ 3 SOLVE $<$ M $\langle 1\ \infty\rangle$

5. $|f(x)-L| \prec \frac{E}{M}$ 4 SOLVE $>$ E^1 $\langle 0\ \frac{|L|}{2}\rangle$

6. $\square$ 5,1 Rule 3.9.3 E_1 $\langle 0\ \frac{E}{M}\rangle$

---

1. Initially we have assume that E is of type $\langle 0\ \infty\rangle$. From (4) E is of type $\langle -\infty\ \frac{|L|}{2}\rangle$. So we take the intersection of $\langle 0\ \infty\rangle$ and $\langle -\infty\ \frac{|L|}{2}\rangle$ to obtain $\langle 0\ \frac{|L|}{2}\rangle$ (retyped).

EXAMPLE 2

Given the clauses

1. $|f(x)-L| < E_1$

2. $|\frac{1}{f(x)} - \frac{1}{L}| < E$

where E_1, E, x are variable and E, E_1 each has type $\langle 0\ \infty \rangle$

Rule 3.10(a) is used on clause (1) and (2)

EXTRACT $((f(x)-L)(\frac{1}{f(x)} - \frac{1}{L}))$=EXTRACT$((f(x)-L)((-1)\frac{f(x)-L}{f(x).L} + 0))$

yield $[-\frac{1}{f(x).L} \quad 0]$; and the following resolvent is produced.

3. $(|\frac{1}{f(x).L}| < M) \quad (|f(x) - L| < \frac{E}{2M})$

where M is a new variable of type $\langle 0\ \infty \rangle$

4. $|\frac{1}{f(x).L}| < M$ 3,1 Rule 3.9.3 $E_1 \langle 0 \quad \frac{E}{2M} \rangle$

Again limit heuristic rule 3.10(b) is used on (4) and (1)

EXTRACT $((f(x)-L)(f(x)L))$=EXTRACT$((f(x)-L)(f(x)+L-L^2+L^2))$

=EXTRACT $(f(x)-L)(L(f(x)-L)+L^2)$

yield $[-L \quad L^2]$; and the following resolvent is produced.

5. $(\ |L|\prec M')\vee(\ |f(x)-L|\prec \frac{1}{MM'})\ \ (\ |L^2|\succ \frac{2}{M})$

where M' is a new variable of type $\langle 0\ \infty\rangle$

6. $(\ |f(x)-L|\prec \frac{1}{MM'})\ \ (|L^2|\succ \frac{2}{M})$ 5 SOLVE $<$ $M'\ \langle L\ \infty\rangle$

7. $|f(x)-L|\prec \frac{1}{MM'}$ 6 SOLVE $>$ $M\ \langle \frac{2}{L^2}\ \infty\rangle$

8. $\square$ 7,1 Rule 3.9.3 E_1 intersection of $\langle 0\ \frac{E}{2M}\rangle\langle 0\ \frac{1}{MM'}\rangle$

In the similar fashion we can prove the following theorems:

(a) If $\lim_{x\Rightarrow a} f(x)=L_1$; $\lim_{x\Rightarrow a} g(x)=L_2$; then $\lim_{x\Rightarrow a} \frac{f(x)}{g(x)}=\frac{L_1}{L_2}$

(b) $\lim_{x\Rightarrow a} \frac{1}{x^2}=\frac{1}{a^2}$

(c) If a function f is derivable at a point, then $\frac{1}{f}$ is also derivable at that point.

3.11 The Implication Method

In this section we have replaced resolution by an 'implication method'. IMPLY examines the connectives in the formulas, given as arguments to it, and creates one or two subgoals. These subgoals usually call to IMPLY with new arguments which are closely related to but simpler than the original arguments.

The subroutine IMPLY has two arguments:

E (the current formula under examination).

R (a reserve)

usually E is of the form

(H ⇒ C)

The result of a call to IMPLY is either a substitution or NIL. The later indicates failure to establish the subgoal. IMPLY attempts to find and return the most general substitution σ such that (R ⇒ E) is true. If σ is the empty substitution then IMPLY returns T.

Table 1 gives rules describing some of the operation of IMPLY. These rules are applied in the order of their occurrence

in the table; if one fails, the next is tried; if all fail, IMPLY returns NIL. IMPLY returns the value given by the first rule which does not give NIL. We use the shorter notation [E,R] for (IMPLY ER).

Before a formula E is sent to IMPLY it is first converted to a quantifier free form, but without converting it first to prenex normal form. The quantifier free is achieved by using skolem functions then a call is made to (IMPLY E NIL).

TABLE 1

Some of the Rules Defining IMPLY

	INPUT	OUTPUT
1.	[H $\Rightarrow$ C,R] If H $\equiv$C, then If there is a substitution σ which unifies H and C (i.e. $H_\sigma \equiv C_\sigma$) then	T σ
2.	[A $\wedge$B, R] 2.1 [A, R] yields σ_1 if and 2.2 [B_{σ_1}, R] yields σ_2	then ($\sigma_1 \circ \sigma_2$)

3. $[A \vee B, R]$

   if $[A, R]$ yields σ_1 then σ_1

   if $[B, R]$ yields σ_2 then σ_2

4. $[(A \Rightarrow B) \Rightarrow C, R]$

   4.1 $[B \Rightarrow C, R]$ yields σ_1

   if

   4.2 $[R \Rightarrow A_{\sigma_1}, NIL]$ yields σ_2

   end — then $(\sigma_1 \circ \sigma_2)$

   This rule is commonly known as backwards chaining.

5. $[H \Rightarrow (A \Rightarrow B), R]$ $[H \wedge A \Rightarrow B, R]$

6. $[A \vee B \Rightarrow C, R]$

   6.1 $[A \Rightarrow C, R]$ yields σ_1

   if

   6.2 $[B_{\sigma_1} \Rightarrow C, R]$ yields σ_2

   end — then $(\sigma_1 \circ \sigma_2)$

7. $[H \Rightarrow A \vee B, R]$

   if $[H \Rightarrow A, R]$ yields σ_1 then σ_1

   if $[H \Rightarrow B, R]$ yields σ_2 then σ_2

8. $[H \Rightarrow A \wedge B, R]$

   8.1 $[H \Rightarrow A, R]$ yields σ_1

   if

   8.2 $[H \Rightarrow B_{\sigma_1}, R]$ yields σ_2

   end — then $(\sigma_1 \circ \sigma_2)$

9.(a) $[A \wedge B \Rightarrow C, R]$

if $[A \Rightarrow C, R \wedge B]$ yields σ_1 then σ_1

if $[B \Rightarrow C, R \wedge A]$ yields σ_2 then σ_2

(b) $[A \wedge B \Rightarrow C, R]$

if $[A \Rightarrow C, R \wedge A \wedge B]$ yields σ_1 then σ_1

if $[B \Rightarrow C, R \wedge A \wedge B]$ yields σ_2 then σ_2

10. $[H \Rightarrow \sim A \vee B, R]$ $[H \wedge A \Rightarrow B, R]$

11. $[\sim A \wedge B \Rightarrow C, R]$ $[B \Rightarrow A \vee C, R]$

12. $[\sim H \Rightarrow C, R]$ $[R \Rightarrow C \vee H, NIL]$

13. $[H \Rightarrow \sim C, R]$ $[H \wedge C \Rightarrow NIL, R]$

14. $[A = B \Rightarrow C, R]$ $[R' \Rightarrow C', NIL]$

where R' and C' are gotten by replacing B by A in R and in C.

15. $[H \Rightarrow A=B, R]$ (SOLVE ⇒ AB)

(i.e. if there is a substitution σ which unifies A and B, then return σ).

3.12 Examples of computer proofs

In order to use the limit heuristic described in

section 3.7, we must add the following rule to Table 1.

16. $[|A| < E' \Rightarrow |B| < E, R]$
if

16.0 (EXTRACT AB) is $(KL\ \sigma)$(i.e., $(B=K.A+L)_\sigma$),
and if,

16.1 $[R \Rightarrow (|K| < M)_\sigma, NIL$ yield σ_1 for some variable M,
and if,

16.2 $[|A| < E' \Rightarrow |A| < \frac{E}{2M}, R]_{\sigma_0\sigma_1}$ yield σ_2,
and if

16.3 $[R \Rightarrow (|L| < \frac{E}{2})_{(\sigma_0\sigma_1\sigma_2)}, NIL$ yield σ_3,
then return the value $(\sigma_0\sigma_1\sigma_2\sigma_3)$

17. $[|A| < E' \Rightarrow |B| > E, R]$
if

17.0 (EXTRACT AB) is $(KL\ \sigma)$(i.e., $(B=L-KA)_\sigma$)
and if

17.1 $[R \Rightarrow |K| < M)_\sigma, NIL]$ yield σ_1 for some variable M,
and if

17.2 $[|A| < E' \Rightarrow |A| < \frac{E}{M}, R]_{\sigma_0\sigma_1}$ yield σ_2,
and if

17.3 $R \Rightarrow (|L| > 2E)_{(\sigma_0\sigma_{10}\sigma_2)}$, NIL yields σ_3
then return the value $(\sigma_0\sigma_{10}\sigma_{20}\sigma_3)$

Also we need two additional rules for solving inequalities, one rule for types, and one for equations.

18.1 $[H \Rightarrow a<b, R]$ (SOLVE $<$ ab)

18.2 $[H \Rightarrow a>b, R]$ (SOLVE $>$ ab)

19.1 $[a<b \Rightarrow a'<c, R]$ $[(b<c)_\sigma \vee (b=c)_\sigma, R]$

19.2 $[a>b \Rightarrow a'>c, R]$ $[(b>c)_\sigma \vee (b=c)_\sigma, R]$

If there is a substitution σ for which $(a=a')_\sigma$

20. $[H \Rightarrow A \in B, R]$

If A has type B then return T

21. $[a=b \Rightarrow c=d, R]$ (SOLVE=(a-b)(c-d))

These six rules are placed at the beginning of the Table 1 (Section 3.11), in the order 18,19,20,21,16,17.

Also, a provision is made for assigning a type to an expression A when it appears in the form $(A \in B)$ or $(A<B)$ in the hypothesis of the theorem being proved. This is accomplished

when IMPLY is proving a subgoal of the form [H ⇒ G, R] by replacing H by (SET-TYPE H). Such calls to SET-TYPE need only be made in Rules 5,10,13 and before the first call to IMPLY i.e., when new material is added to H.

In what follows, R denotes the real number, P denotes the positives, and FRR denotes the functions on R to R. We use (limit faL) to denote $\lim_{x \Rightarrow a} f(x) = L$.

The standard definition of Limit is:

(Limit faL) ⇔

$$(a \in R) \wedge (L \in R) \wedge (f \in FRR) \wedge ALL \in (0 < \epsilon \Rightarrow SOME\ \delta (0 < \delta\ ALL\ x$$

$$(x \in R \wedge x \neq a \wedge |x-a| < \delta \Rightarrow |f(x)-L| < \epsilon\ )))$$

EXAMPLE 1

If $\lim_{x \Rightarrow a} f(x) = l \neq 0$, $\exists k > 0$ and $\delta > 0$ such that $0 < |x-a| < \delta$ then $|f(x)| > k$.

The definition of Limit is used to obtain[1]

---

1. We have elliminate the quantifiers and introduced Skolem expressions. Also we have used x in place of (xD), L in place of (L), f(x) in place of (f)(xD) and so on.

$$\alpha\{a \in R \wedge L \in R \wedge f \in FRR \wedge (0 < E_1 \Rightarrow (0 < D_1 \wedge (x_1 \in R \wedge x_1 \neq a \wedge |x_1 - a| < D_1 \Rightarrow |f(x_1) - L| < E_1))))$$

$$\gamma\{= (0 < k \Rightarrow (0 < D \wedge (x \in R \wedge x \neq a \wedge |x - a| < D \Rightarrow |f(x)| > k)))$$

We will start description of the computer procedure in finding its proof. A call is made

(IMPLY($\alpha \Rightarrow \gamma$) NIL)

SET-TYPE is applied to , assigning type R to a,L and type FRR to f and the subformula $(a \in R)$, $(L \in R)$, $(f \in FRR)$ are removed from .

Rule 5 is applied, converting the formula to

$$(\alpha \wedge 0 < k \Rightarrow 0 < D \wedge (x \in R \wedge x \neq a \wedge |x - a| < D \Rightarrow |f(x)| > k))$$

SET-TYPE is applied to the hypothesis; k is assigned type $\langle 0\ \infty \rangle$ and $0 < k$ is removed.

Rule 8 calls IMPLY on the two formula

$(\alpha \Rightarrow 0 < D)$

and

$(\alpha \Rightarrow (x \in R \wedge x \neq a \wedge |x - a| < D \Rightarrow |f(x)| > k))$

The first call is satisfied by the Rule 18.1, which uses SOLVE to assign type $\langle 0\ \infty\rangle$ to D. The second results is an application of Rule 5, so the current subgoal is

$$(\alpha \wedge x \in R \wedge x \neq a \wedge |x-a| < D \Rightarrow |f(x)| > k)$$

SET-TYPE is applied to the hypothesis : x is assigned type R and $(x \in R)$ is removed.

By Rule 9(a), the reserve R is set to

$$x \neq a \wedge |x-a| < D$$

and

$(\alpha \Rightarrow |f(x)| > k)$ becomes current goal.

Rule 4 (backward chaining) is now applied. That is, the program will try first to establish the conclusion $|f(x)| > k$ from α . This is subgoal (1). When this subgoal is establish, the program will try to satisfy the hypothesis of α ,namely subgoal (2) below:

(1) $(0 < D_1 \wedge (x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1 \Rightarrow |f(x_1)-L| < E_1 \Rightarrow |f(x)| > k)$

By Rule 9(a) the program will first try to prove

$$(0 < D_1 \Rightarrow |f(x)| > k)$$

SET-TYPE assigns type $\langle 0\ \infty\rangle$ to D_1 and

$(x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D \Rightarrow |f(x_1)-L| < E_1) \Rightarrow |f(x)| > k$

becomes the current goal.

Again the program will "chain backward" using Rule 4. The current subgoal becomes (11) and the hypothesis:

$(x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1)$ is satisfied later at (12)

(11) $\quad |f(x_1)-L| < E_1 \Rightarrow |f(x)| > k$

Now the program will try to apply Rule 17, the limit heuristic. First (EXTRACT($f(x_1)-L$)($f(x)$)) is computed to be $(1\ \ 1\ \ \sigma)$ where $\sigma = [x/x_1]$

Because the result of the call to extract is not NIL, Rule 17 is applicable. The program will try to establish the three subgoals (111), (112) and (113) in accordance with Rules 17.1, 17.2 and 17.3

The current subgoal is

(111) $(x \neq a \wedge |x-a| < D \Rightarrow |1\ 1| < M)$ (where M is a new variable of the type $\langle 0\ \infty\rangle$)
is established by Rule 18.1 SOLVE types M as $\langle 1\ \infty\rangle$.
The subgoal is establish and the program will try to prove.

(112) $|f(x_1)-L| < E_1 \Rightarrow |f(x)-L| < \frac{k}{M}$

By Rule 19.1, the program will try to establish

$(E_1 < \frac{k}{M}) \vee (E_1 = \frac{k}{M})$

The first half of the disjunction is satisfied by a call to (SOLVE $< E_1$ $\frac{k}{M}$) giving type $\langle -\infty \frac{k}{M} \rangle$ to E_1.

This subgoal is establich and the program will try to establish

(113) $(x \neq a \wedge |x-a| < D \Rightarrow |L| > 2k)$

Rule 18.2 applied; (SOLVE $>|L|$ $2k$) is called resulting in the type $\langle -\infty \frac{|L|}{2} \rangle$ to k. Hence k is retyped as $\langle 0 \quad \frac{|L|}{2} \rangle$.

Now the program will return to

(12) $(x \neq a \wedge |x-a| < D \Rightarrow x_1 \in R \wedge x \neq a \wedge |x-a| < D_1)$

where $\sigma = [x/x_1]$

that is

$(x \neq a \wedge |x-a| < D \Rightarrow x \in R \wedge x \neq a \wedge |x-a| < D_1)$

This subcall is established by several subcalls. The conclusion ($x \in R$) follows since x has type R. ($x \neq a$) occurs in the hypothesis and finally

$$|x-a| < D \Rightarrow |x-a| < D_1$$

is established through Rules 19, 18 and a call to SOLVE <. As result the type D is changed to $\langle 0 \quad D_1 \rangle$.

Finally the subgoal

(2) $x \neq a \wedge |x-a| < D \Rightarrow 0 < E_1$ is established by Rule 18 and a call to (SOLVE<0 E_1) which retypes E_1 as $\langle 0 \quad \frac{k}{M} \rangle$. E_1 previously had a type $\langle -\infty \frac{k}{M} \rangle$

Q.E.D.

The proof is complete. We list here the final types assigned to the variables

E_1	$\langle 0 \quad \frac{k}{M} \rangle$
k	$\langle 0 \quad \frac{\lvert k \rvert}{2} \rangle$
M	$\langle 1 \quad \infty \rangle$
D	$\langle 0 \quad D_1 \rangle$
D_1	$\langle 0 \quad \infty \rangle$

EXAMPLE 2

(Limit of a quotient)

$$\text{Limit}(f \in L) = \text{Limit}\ (\tfrac{1}{f})\,a\,(\tfrac{1}{L})$$

The definition of Limit is used to obtain

$$((a \in R \wedge L \in R \wedge f \in FRR \wedge ALL\,E_1(0 < E_1 \Rightarrow SOME\ D_1(0 < D_1 \wedge ALL\ x_1$$

$$(x_1 \in R \wedge x_1 \neq a \wedge |x_1 - a| < D_1 \Rightarrow |f(x_1) - L| < E_1)))$$

$$\Rightarrow (a \in R \wedge L \in R \wedge \tfrac{1}{f} \in FRR \wedge ALLE(0 < E \Rightarrow SOME\ D\ (0 < D \wedge ALL\ x$$

$$(x \in R \wedge x \neq a \wedge |x - a| < D \Rightarrow |\tfrac{1}{f(x)} - \tfrac{1}{L}| < E\ ))))$$

The first three parts of the conclusion ($a \in R$), ($L \in R$) and ($\frac{1}{f} \in FRR$) can be proved byusing the hypothesis of the theorem and the inverse property of R and FRR.

The remainder of the theorem is prepared for IMPLY by eliminating the quantifiers and introducing skolem expression.

We write the above equation as:

$$\alpha \left\{ (a \in R \wedge L \in R \wedge f \in FRR\ \ (0 < E_1 \Rightarrow (0 < D_1 \wedge (x_1 \in R \wedge x_1 \neq a \wedge |x_1 - a| < D_1 \Rightarrow |f(x_1) - L| < E_1\ )))) \right.$$

(11)

$$\gamma \left\{ \Rightarrow\ (0 < E \Rightarrow (0 < D \wedge (x \in R \wedge x \neq a \wedge |x - a| < D \Rightarrow |\tfrac{1}{f(x)} - \tfrac{1}{L}| < E))) \right.$$

Now we will resume the description of the computer's procedure in finding its proof. A call is made to

(IMPLY ($\alpha \Rightarrow \gamma$) NIL)

where α, γ are given in (ii) SET-TYPE is applied to α, assigning type R to a,L and type FRR to f, and the subformula ($a \in R$), ($L \in R$), ($f \in FRR$) are remove from α.

Rule 5 is applied, converting the formula to

$$((\alpha \wedge 0 < E \Rightarrow 0 < D \wedge (x \in R \wedge x \neq a \wedge |x-a| < D \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E))$$

SET-TYPE is applied to the hypothesis; E is assigned type $\langle 0\ \infty \rangle$ and $(0 < E)$ is removed.

Rule 8 calls IMPLY on the two formula

$(\alpha \Rightarrow 0 < D)$

and

$(\alpha \Rightarrow (x \in R \wedge x \neq a \wedge |x-a| < D \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E))$

The first call is satisfied by Rule 18.1 which is SOLVE to assign type $\langle 0\ \infty \rangle$ to D. The second result is an application of Rule 5, so the current subgoal is

$$(\alpha \wedge x \in R \wedge x \neq a \wedge |x-a| < D \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E)$$

SET-TYPE is applied to the hypothesis. x is assigned type R and ($x \in R$) is removed.

By Rule 9(b) reserve R is set to be

$\alpha \wedge x \neq a \wedge x=a$ D.

and

($\alpha \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E$) becomes the current goal.

Rule 4 (backward chaining) is now applied. That is the program will try first to establish the conclusion $|\frac{1}{f(x)} - \frac{1}{L}| < E$ from α. This is subgoal (1), then the program will try to satisfy the hypothesis of α, namely subgoal (2) below:

(1) $(0 < D_1 \wedge (x_1 \in R \wedge x_1 \neq a \wedge |x_1 - a| < D_1 \Rightarrow$

$|f(x_1) - L| < E_1) \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E)$.

By Rule 9(b) the program will first try to prove

$0 < D_1 \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E$

SET-TYPE assign type $\langle 0 \infty \rangle$ to D_1 and

$(x_1 \in R \wedge x_1 \neq a \wedge |x_1 - a| < D_1 \Rightarrow |f(x_1) - L| < E_1)$

$\Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E$

becomes current goal (from now on we shall not mention those subgoals which are tried but not established)

Again the program will "chain backward" using Rule 4. The current subgoal becomes (11) and the hypothesis

$(x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D)$ is satisfied later at (12).

(11) $\quad (|f(x_1) - L| < E_1 \Rightarrow |\frac{1}{f(x)} - \frac{1}{L}| < E)$

Now the program will try to apply Rule 16, the limit heuristic. First EXTRACT $(f(x_1)-L)(\frac{1}{f(x)} - \frac{1}{L})$ is computed to be $(- \frac{1}{Lf(x)} \quad 0 \quad \sigma)$ where $\sigma = [x/x_1]$.

As '0' is the returned as the value of L from EXTRACT the two subgoals (111) (112) are tried in accordance with Rules 16.1, 16.2. The current subgoal becomes

(111) $\quad \alpha \wedge x \neq a \wedge |x-a| < D \Rightarrow |-\frac{1}{Lf(x)}| < M$

where M is a new variable which is assigned type $\langle 0 \infty \rangle$

By rule 9(a), the reserve R is set to $(x \neq a \wedge |x-a| < D)$ and $\alpha = |\frac{1}{Lf(x)}| < M$ becomes the current subgoal. Rule 4 is applied. The current subgoal becomes (111 1) and the hypothesis of α satisfied later at (111 2)

(111 1) $(0<D_1 \wedge (x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1 \Rightarrow |f(x_1)-L| < E_1)$

$\Rightarrow |\frac{1}{Lf(x)}| < M)$

By Rule 9(a) the program tries

$(x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1 \Rightarrow |f(x_1)-L| < E_1) \Rightarrow |\frac{1}{Lf(x)}| < M$

and D_1 is assigned type $\langle 0\ \infty \rangle$

Another application of Rule 4 setup two subgoals (111 11), (111 12)

(111 11) $|f(x_1)-L| < E_1 \Rightarrow |\frac{1}{Lf(x)}| < M$ which can be written in the form of

$$|f(x_1)-L| < E_1 \Rightarrow |Lf(x)| > \frac{1}{M}$$

Since EXTRACT $((f(x_1)-L)(Lf(x))$

$=$ EXTRACT $(f(x_1)-L)(L(f(x)-L)+L^2)$ yields

$[L \quad L^2\ \sigma]$ where $\sigma = [x/x_1]$

The Limit heuristic 5.1(b) is applicable to (111 11). Because the result of the call to EXTRACT is not NIL, Rule 17 is applicable. The program will try to establish the three subgoals (111 111), (111 112) and (111 113) in accordance

with Rule 17.1, 17.2 and 17.3. The current subgoal becomes:

(111 111) $x \neq a \wedge |x-a| < D \Rightarrow |L| < M'$

where M' is a new variable which is assigned type $\langle 0 \ \infty \rangle$

Rule 18.1 is applied; (SOLVE $<|L|$ M') is called, resulting the type $\langle L \ \infty \rangle$ for M'. M' is retyped as $\langle 0 \ \ L \rangle$

Now the program will returns to the subgoal

(111 112) $|f(x_1)-L| < E_1 \Rightarrow |f(x_1)-L| < \frac{1}{MM'}$

By Rule 19.1 the program will try to establish

$(E_1 < \frac{1}{MM'}) \vee (E_1 = \frac{1}{MM'})$

The first half of the disjunction is satisfied by a call to (SOLVE $< E_1 \ \frac{1}{MM'}$) giving type $\langle -\infty \ \frac{1}{MM'} \rangle$ to E_1.

Now the program will return to the subgoal

(111 113) $x \neq a \wedge |x-a| < D \Rightarrow |L^2| > \frac{2}{M}$

Rule 18.2 is applied: (SOLVE $>|L^2|$ $\frac{2}{M}$) is called, resulting in the type $\langle \frac{2}{L^2} \ \infty \rangle$. But N is of the type $\langle 0 \ \infty \rangle$.

Hence it is retyped as $\langle 0 \quad \frac{2}{|L^2|} \rangle$

Now the program will return to the subgoal (111 12)

(111 12) $(x \neq a \wedge |x-a| < D \Rightarrow x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1)$

where $\sigma = [x/x_1]$

i.e. $x \neq a \wedge |x - a| < D \Rightarrow x \in R \wedge x \neq a \wedge |x - a| < D_1$

This subgoal is established by several subcalls. The conclusion ($x \in R$) follows since x has type R, ($x \neq a$) occurs in the hypothesis and finally ($|x-a| < D \Rightarrow |x-a| < D_1$) is established through 18.1 and 19.1 and a call to SOLVE . As a result, the type of D is changed to $\langle 0 \quad D_1 \rangle$.

(111 2) $x \neq a \wedge |x-a| < D \Rightarrow 0 < E_1$

is established through 18. SOLVE types E_1 as $\langle 0 \quad \frac{1}{MM_1} \rangle$ Recall that E_1 was given type $\langle -\infty \quad \frac{1}{MM_1} \rangle$ at (111 112).

Thus both subgoal of (111) have been established and the program will return to the second subgoal of the first use of limit heuristic.

(112) $|f(x_1)-L| < E_1 \Rightarrow |f(x_1)-L| < \frac{E}{2M}$

This subgoal is quickly established using Rule 18.1 and 19.1 and (SOLVE $< E_1 \quad \frac{E}{2M}$). E_1 is retyped as (intersection $\langle 0 \quad \frac{1}{MM'} \rangle \quad \langle -\infty \quad \frac{E}{2M} \rangle$. Recall that E_2 had been given type $\langle 0 \quad \frac{1}{MM'} \rangle$ to establish (111 2). Since program does not know which of $\frac{1}{MM'}$ and $\frac{E}{2M}$ is the smaller, the intersection is given as the answer, after it has checked that the intersection is non-empty.

All of the subgoal of the first application of limit heuristic at (11) have been established, giving as an answer to (11) the substitution $\sigma = [x/x_1]$

Now the program will try to satisfy

(12) $(\alpha \wedge x \neq a \wedge |x-a| < D \Rightarrow x_1 \in R \wedge x_1 \neq a \wedge |x_1-a| < D_1)$

The substitution $[x/x_1]$ establish the first two parts of the conclusion. To prove the third part, the program tries

$(|x-a| < D \Rightarrow |x-a| < D_1)$

which results in the retyping of D as (intersection $\langle 0 \quad D_1 \rangle \quad \langle -\infty \quad D_1 \rangle$. Recall that D previously had type $\langle 0 \quad D_1 \rangle$.

Finally the subgoal

(2) $(\alpha \wedge x \neq a \wedge |x-a| < D \Rightarrow 0 < E_1)$

is establish by Rule 18.1 on a call to SOLVE $< 0\ E_1$ which types E_1 as $\langle 0\ \infty \rangle$ which has no contribution towards the previously typed of E_1.

Q.E.D.

The proof is complete. We list here the final assigned to the variables:

E_1	(intersection $\langle 0\ \frac{1}{MM'} \rangle$ $\langle -\infty\ \frac{E}{2M} \rangle$
M	$\langle 0\ \frac{2}{\lvert L'' \rvert} \rangle$
D	$\langle 0\ D_1 \rangle$
M'	$\langle 0\ L \rangle$

These proofs may be seen long and drawn out but there are essentially the steps a human prover would have to follow in finding and exhibiting a proof.

CHAPTER IV

ABSTRACTION MAPPING ON NON-CLAUSAL THEOREM PROVING

4.1 The Formalism

Throughout our discussion, in this chapter, we assume that commutative properties of $\vee$, $\wedge$ and $\Leftrightarrow$. 'T' and 'F' are atoms and denote truth and falsewood, respectively. Any n-ary predicate symbol followed by n terms is an atom. Well-formed-formula are all and only those finite expressions defined by the following two rules:

(a) Any atom is a well-formed-formula.

(b) If A and B are well-formed-formulas, then so are

$$\sim A, (A \vee B), (A \wedge B), (A \Rightarrow B), (A \Leftrightarrow B).$$

4.2 Polarity

Let A,B,w be well-formed-formulas. We say that w is positive in A iff at least one occurrence of w in A is positive. w is negative in A iff at least one occurrence of w in A is negative. The one occurrence of w in w is positive. We determine the polarity of the consitituents of a well-formed-formulla as follows:

(i) If w is positive (negative) in A, then w is negative (positive) in

$$\sim A \quad \text{and} \quad (A \Rightarrow B).$$

(ii) If w is positive (negative) in A, then w is positive (negative) in

$(A \vee B)$, $(A \wedge B)$ and $(B \Rightarrow A)$

(iii) If w occurs in A, then w is positive and negative in

$(A \Leftrightarrow B)$.

Let A, w and a be well-formed formulas.

A $\{a/w\}$ denotes the well-formed-formula obtained by replacing every occurrence of w in A by a. We may have replacements with more then one component, e.g. $\{A_1/w_1, \ldots .. A_n/w_n\}$. We interpret this as a simultaneous replacement noting that both the A's and w's are well-formed-formulas and for $1 \leq i, j \leq n$, $i \neq j$, w_i does not occur in w_j.

4.3 Reduction

We denote 'reduces to' by $\longrightarrow$ and 'does not reduce to' by $\not\longrightarrow$. Let A,B,C,w,a be well-formed-formullas. We have the following reduction rules:

$\sim T \longrightarrow F$,	$\sim F \longrightarrow T$
$(T \wedge w) \longrightarrow w$,	$(F \wedge w) \longrightarrow F$
$(T \vee w) \longrightarrow T$,	$(F \vee w) \longrightarrow w$
$(T \Rightarrow w) \longrightarrow w$,	$(F \Rightarrow w) \longrightarrow T$
$(w \Rightarrow T) \longrightarrow T$,	$(w \Rightarrow F) \longrightarrow \sim w$
$(T \Leftrightarrow w) \longrightarrow w$,	$(F \Leftrightarrow w) \longrightarrow \sim w$

If well-formed-formulla is in reduced form whenever no reduction rule can be applied to it. In addition to $\longrightarrow$ and $\not\longrightarrow$, we will need the following meta-linguistic relations:

(i) We use $A =: B$ ($A \not=: B$) to denote that A and B have (do not have) the same truth-value under all assignments of truth-values to their atoms.

(ii) We use $A \equiv B$ ($A \not\equiv B$) to denote that A and B are (are not) Syntactically identical.

The above reduction rules are essentially those truth-functional simplifications that one would expect when dealing with the connectives in our formalism. In the next four lemma we relate the process of reduction with the truth value and syntactic form of the well-formed-formula involved.

Lemma 4.3.1

Every well-formed-formula W has a reduced form 'W' such that either $W' \equiv T$, or $W' \equiv F$ or T and F do not occur in W'.

Lemma 4.3.2

If all the atoms of a well-formed-formula W are T or F, then the reduced form of W is either T or F.

Lemma 4.3.3

If A reduced to B, then $A =: B$ and $A \neq B$.

Lemma 4.3.4

Let A be any well-formed formula and $\{A_1, A_2 \ldots\ldots A_n\}$ be the set of all the distinct atoms in A, let a be any assignment of truth-values to $\{A_1, A_2 \ldots\ldots\ldots A_n\}$.

If $A =: F(T)$ then

$$A\ \{a(A_1)/A_1, \ldots\ldots\ldots\ldots a(A_n)/A_n\} \longrightarrow F(T)$$

where $a(A_i)$ is the truth-value assigned to A_i by a.

4.4 Reduction, polarity and Replacement

In this section we mention theorems which relate the replacement of sub-well-formed-formula by a truth-value with the resulting value of the entire well-formed-formula and the polarity of the sub-well-formed formula. These theorems are required for completeness [29] .

Theorem 4.4.1

Given well-formed-formula A where $T \neq: A \neq: F$

If $A\ \{F/w\} =: F(T)$ then w is positive(negative) in A.

If $A\ \{T/w\} =: F(T)$ then w is negative(positive) in A.

Theorem 4.4.1'

Given well formed formula A where $T \not\longleftrightarrow A \not\longleftrightarrow F$.
If A $\{F/w\} \longrightarrow F(T)$, then w is positive(negative) in A.
If A $\{T/w\} \longrightarrow F(T)$, then w is negative(positive) in A.

Theorem 4.4.2

Given well-formed-formula A,B and atom w where $T \neq w \neq F$.
Suppose that w occurs in B and $A \longrightarrow B$.
If w is positive(negative) in B, then w is positive (negative) in A.

Theorem 4.4.3

If $A=:F(T)$, $A \not\longleftrightarrow F(T)$, and $A \neq F(T)$, then some atom a is both positive and negative in A.

4.5 The Proof Procedure

We now give a brief description of the Proof procedure. In essence it is the non-clausal analogue to breadth-first binary resolution. The inference rule is known as NC-resolution.

Given well-formed-formulae U_i and U_j, we can determine if they contain atoms $L_1,.....L_n$ such that $\{L_1,.....L_n\}$

can be unified by mgu M i.e. $\{L_1 \ldots\ldots L_n\}M=L$. We may also determine whether L occurs in U_iM and U_jM with opposite polarity. Now suppose L is positive in U_iM and negative in U_jM. The following well-formed-formula is an NC-resolvent of U_i and U_j:

$$U_iM\ \{F/L\} \vee U_jM\ \{T/L\}$$

If L is negative in U_iM and positive in U_jM, then we have the following dual NC-resolvent:

$$U_iM\ \{T/L\} \vee U_jM\ \{F/L\}$$

Each of these NC-resolvents is a logical consequence of U_i and U_j. Clearly U_i and U_j may have only finitely many NC-resolvents. We may therefore compute the set of all NC-resolvent of pairs of well-formed-formulas in some given set U of well-formed-formulas, and add the NC-resolvents to the original set. The interation of this process finitely many times will yield a contradiction for all unsatisfiable sets U.

As an example, consider the two well-formed-formulas:
$P(x) \Leftrightarrow R(a)$ and $P(b) \Rightarrow (Q(a,b) \vee \sim P(y))$

We may take $\{L_1, L_2, L_3\} = \{P(x), P(b), P(y)\}$.

P(b) is positive and negative in $P(b) \Leftrightarrow R(a)$, but is only negative in $P(b) \Rightarrow (Q(a,b) \vee \sim P(b))$. We may infer the NC-resolvent:

$$P(b) \Leftrightarrow R(a) \{P/P(b)\} \vee P(b) \Rightarrow (Q(a,b) \vee \sim P(b)) \{T/P(b)\}$$

which reduces to

$$\sim R(a) \vee Q(a,b).$$

4.6 Ordinary Abstraction

In this section we will discuss abstraction mapping [38] on non-clausal theorem proving.

Definition: An abstraction is an association of a set f(NC) of non-clauses with each non-clause NC such that f has the following properties:

(i) If non-clause NC3 is a NC-resolvent of NC1 and NC2 and $ND3 \in f(NC3)$ then there exist $ND1 \in f(NC1)$ and $ND2 \in f(NC2)$ such that NC-resolvent of ND1 and ND2 subsumes ND3.

(ii) $f(NIL) = \{NIL\}$ (NIL is empty clause)

(iii) If NC1 subsumes NC2, then for every abstraction ND2 of

NC2 there is an abstraction ND1 of NC1 such that ND1 subsumes ND2.

If f is a mapping of these properties then we will call f an abstraction mapping of non-clauses. Also if $ND \in f(NC)$ we call ND is an abstraction of NC. Abstraction usually also satisfies the property that each ND in f(NC) is a tautology if NC is.

The following result gives us a fairly general method of constructing abstraction:

Theorem 4.6.1

Suppose Φ is a mapping from literals to literals. Let us extend Φ to a mapping from non-clauses to non-clauses by

$\Phi(NC) = \{\Phi(L): L \in NC\}$. Suppose Φ satisfies the following properties:

(1) $\Phi(\sim L) = \sim\Phi(L)$. That is preserves complements.

(2) $\Phi(F)=F$.

(3) $\Phi(T)=T$.

(4) If NC and ND are non-clauses and ND is an instance

of NC, then Φ(ND) is an instance of Φ(NC).

That is Φ preserve instances.

Then Φ is an abstraction mapping. To be precise, f is an abstraction mapping where f(NC) = $\{\Phi(NC)\}$.

Proof: All property are easy to verify except property (i).

We do this as follows:

Suppose NC3 is a NC-resolvent of NC1 and NC2. Then there exist A1, A2 such that A1 $\subset$ NC1, A2 $\subset$ NC2 and there exist substitution $\alpha 1$ and $\alpha 2$ such that A1$\alpha 1$ = $\{L\}$ and A2$\alpha 2$ = $\{L\}$, for some literal L. Let $\alpha 1$, $\alpha 2$ be most general such substitution and L is positive in NC1$\alpha 1$ and negative in NC2$\alpha 2$. Suppose

$$NC3 = ((NC1\ \{F/A1\})\alpha 1)R \cup ((NC2\ \{T/A2\})\alpha 2)R$$

where R is the reduction.

We desire to show that Φ(NC3) is subsumed by a resolvent of Φ(NC1) and Φ(NC2). Now

$$\Phi(NC1\alpha 1) = ((\ \Phi(NC1\ \{F/A1\})\alpha 1)\ \ \{\Phi(L)/F\})$$
$$\Phi(NC2\alpha 2) = ((\ \Phi(NC2\ \{T/A2\})\alpha 2)\ \ \{\Phi(L)/T\})$$

Thus $\Phi((NC1\ \{F/A1\})\alpha 1) \cup \Phi((NC2\ \{T/A2\})\alpha 2)$

is a NC-resolvent of $\Phi(NC1 \alpha 1)$ and $\Phi(NC2 \alpha 2)$ or has the proper subset which is NC-resolvent of $\Phi(NC1 \alpha 1)$ and $\Phi(NC2 \alpha 2)$. Hence $\Phi(((NC1\{F/A1\})\alpha 1)R) \cup \Phi(((NC2\{T/A2\})\alpha 2)R)$ is a NC-resolvent of $\Phi(NC1 \alpha 1)$ and $\Phi(NC2 \alpha 2)$ or has the proper subset which is NC-resolvent of $\Phi(NC1 \alpha 1)$ and $\Phi(NC2\ 2)$. Note that

$$\Phi(NC3) = \Phi(((NC1\{F/A1\})\alpha 1)R) \cup \Phi(((NC2\{T/A2\})\alpha 2)R)$$

Hence NC-resolvent of $\Phi(NC1 \alpha 1)$ and $\Phi(NC2 \alpha 2)$ subsumes $\Phi(NC3)$. However $\Phi(NC1 \alpha 1)$ is an instance of $\Phi(NC1)$ and $\Phi(NC2 \alpha 2)$ is an instance of $\Phi(NC2)$ by properties of Φ. Hence by properties of NC-resolution some NC-resolvent of $\Phi(NC1)$ and $\Phi(NC2)$ subsumes $\Phi(NC3)$. In the similar fashion we could prove the theorem by assuming L is negative in $NC1 \alpha 1$ and positive in $NC2 \alpha 2$.

We could prove a similar theorem if we let Φ be a relation between literals, and if we required Φ to have appropriate properties.

The following result is more general:

<u>Theorem 4.6.2</u>

Suppose F is a mapping literals to literals. Suppose

that for all $\Phi \in F$, for all literals L, $\Phi(\sim L) = \sim\Phi(L)$, $\Phi(F)=F$ and $\Phi(T)=T$. If NC is a non-clause, let $\Phi(NC)$ be $\{\Phi(L): L \in NC\}$ as usual. Suppose that if non-clause ND is an instance of non-clause NC, then for all $\Phi 2 \in F$, there exist $\Phi 1 \in F$ such that $\Phi 2(ND)$ is an instance of $\Phi 1(NC)$.

Define f by $f(NC) = \{\Phi(NC): \Phi \in F\}$.

Then f is an abstraction mapping.

If f is an abstraction mapping as in the above theorem, then we say f is defined in terms of literal mapping. Not all abstraction are defined in terms of literal mappings.

4.7 Example of Abstraction

Using these theorems, we can construct many abstractions. We now give some examples of abstraction, all of which can be obtained from the above theorems.

4.7.1 The Propositional Abstraction

If NC is the non-clause $\{L_1 \ldots . L_k\}$ then f(NC) is $\{NC'\}$, where NC' is the non-clause $\{L_1', \ldots . L_k'\}$ and L_i' is defined as follows:

for $1 \leq i \leq k$,

If L_i is of the form $P(t_1.....t_n)$, then L_i' is P.

If L_i is of the form $\sim P(t_1.....t_n)$, then L_i' is $\sim P$.

Thus f(NC) is a non-clause in the propositional calculus.

4.7.2 Renaming Predicate and Function Symbols

For non-clause NC, f(NC) = {NC'} where NC' is the non-clause in which all function and predicate symbol in NC have been renamed in some systematic way. The renaming need not be one-to-one; two distinct predicate or function symbols may be renamed to the same symbol. However, a predicate symbol and a function symbols may not be renamed to the same symbol.

4.7.3 Changing Signs of Literals

Let Q be a set of predicate symbols. If NC is the non-clause $\{L_1.....L_k\}$, then f(NC) is {NC'} where NC' is the non-clause $\{L_1',.....L_k'\}$, L_i' is defined as follows, for $1 \leq i \leq k$;

If L_i is of the form $P(t_1,....t_n)$ and $P \in Q$, then L_i' is $\sim P(t_1,.....t_n)$. If L_i is of the form $\sim P(t_1,....t_n)$ and $P \in Q$, then L_i' is $P(t_1,....t_n)$, otherwise L_i' is L_i.

4.7.4 Deleting Arguments

For non-clause NC, $f(NC) = \{NC'\}$ where NC' is NC with certain arguments of certain function or predicate symbols deleted. For example $g(t_1,\ldots t_n)$ may be replaced by $g(t_2,\ldots,t_n)$ everywhere. Note that the propositional abstraction is a special case of this (All arguments of all predicate symbols are deleted).

4.7.5 Permuting Arguments

For non-clause NC, $f(NC) = \{NC'\}$ where NC' is NC with the order of the arguments of certain function or predicate symbols changed in some systematic way.

4.8 Algebric Properties of Abstraction

Definition: Suppose f_1 and f_2 are abstractions. The composition of f_1 and f_2 denoted by f_2f_1 defined by

$$f_2f_1(NC) = \cup \{f_2(ND) : ND \in f_1(NC)\}$$

Definition: The identity abstraction is a mapping f such that $f(NC) = \{NC\}$ for all non-clauses NC.

Definition: We say abstraction f_1 and f_2 are inverse if $f_1f_2 = f$

and $f_2 f_1 = f$, where f is the identity abstraction. If an abstraction has an inverse, then it really has not thrown away any information about the set of clauses. For example, a 1-1 renaming of predicate symbol is an invertible abstraction.

Theorem 4.8.1

The composition of two abstraction is an abstraction. Definition: If f_1 and f_2 are abstraction, there union f is defined by $f(NC) = f_1(NC) \cup f_2(NC)$ for all non-clause NC.

Theorem 4.8.2

If f_1 and f_2 are abstraction, their union is also an abstraction.

4.9 Abstraction of Resolution Proofs

We now show how abstraction can be used to guide the search for a proof of a non-clause NC from a set S of non-clauses. First we show that if there is a proof of NC from S, then there is an "abstracted proof" of something subsumming an abstraction of NC from abstraction of non-clauses in S. We then describe procedures which, given an abstracted proof, attempt to reconstruct the original proof.

If f is an abstraction mapping and S is a set of

non-clauses then we write f(S) to indicate $\cup\{f(NC):NC \in S\}$.

Theorem 4.9.1

Suppose S is a set of non-clauses and f is an abstraction mapping for S. Suppose NC' is a non-clause derivable from S by NC-resolution and $ND' \in f(NC')$. Then there is a non-clause NB' derivable from f(S) by NC-resolution such that NB' subsumes ND'.

Proof: By induction on the depth of the proof NC'. If NC' S the theorem is true since we can choose NB' to be ND'. Suppose NC' is NC-resolvent of NC1 and NC2, where NC1 and NC2 can be derived from S by proofs of depth less than the depth of the proof of NC'. Suppose ND1 and ND2 are abstraction of NC1 and NC2 respectively, such that some NC resolvent ND of ND1 and ND2 subsumes ND'. The non-clauses ND1 and ND2 must exist, by the definition of abstraction. Applying the theorem inductively there must exist non-clauses NB1 and NB2 derivable (By applying NC-resolution and reduction). From f(S) such that NB1 subsumes ND1 and NB2 subsumes ND2. It follows by the properties of subsumption that either NB1 subsumes ND or NB2 subsumes ND or some NC-resolvent NB of NB1 and NB2 subsumes ND. Hence either NB1 subsumes ND' or NB2 subsumes ND' or some NC-resolvent NB' of NB1 and NB2 subsumes ND'. This completes the proof.

Note that the derivation of NB' from f(S) will have depth not more than the depth of the derivation of NC from S.

Corollary

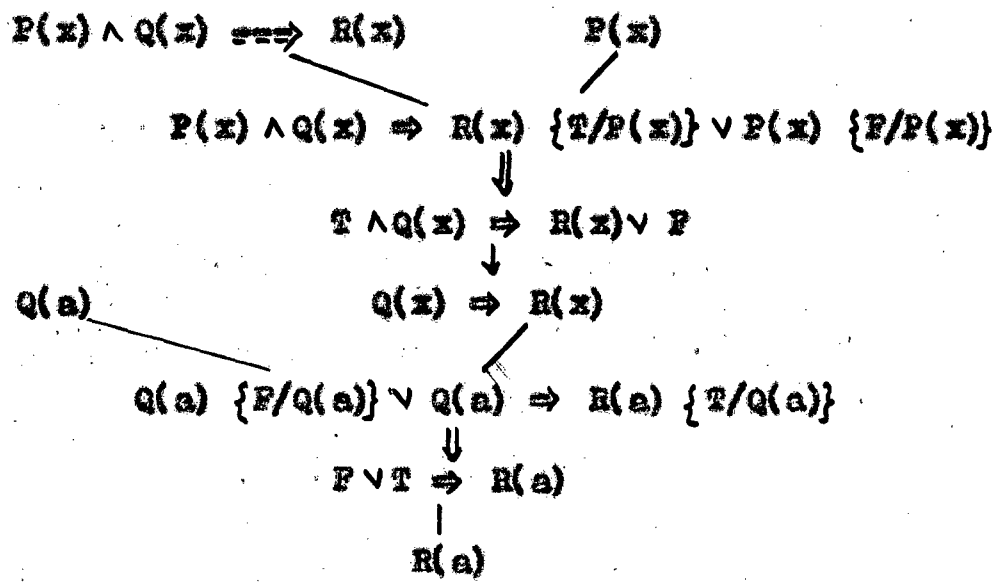
If S is inconsistent so is f(S).

Proof: Take NC' to be NIL. The ND' is also NIL. By properties of abstraction mappings. Since NB' subsumes ND', NB' must NIL, also since NB' is derivable fromf(S), f(S) is inconsistent.

This theorem can be used to show that S is consistent, but its main value for us is in the information that a proof from f(S) can give us about the structure of a possible proof from S. For example see Fig. 3(a), 3(b) and 3(c).

4.10 Terminology Relating to Proofs

We now introduce some terminology which will help to describe and analyse various procedure for using abstracted proofs as a guide in the search for a proof for a non-clause NC from a set S of non-clauses. We consider non-clauses that are variants to be identical. This can be accomplished by choosing variables in non-clause in some canonical way.



Suppose f is the propositional abstraction.

Thus $P(t_1,\ldots t_n)$ is replaced by P etc.. We have the following abstracted proof:

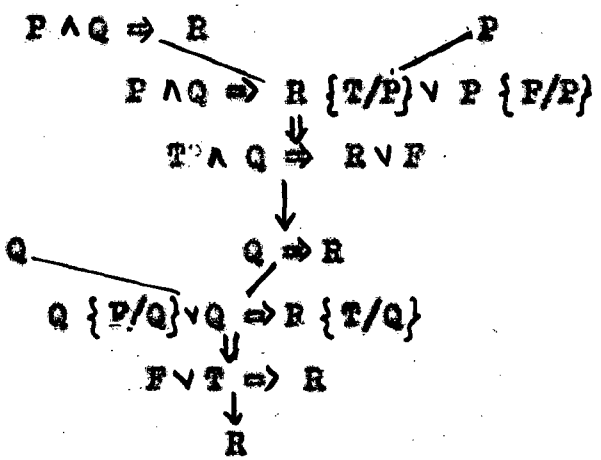
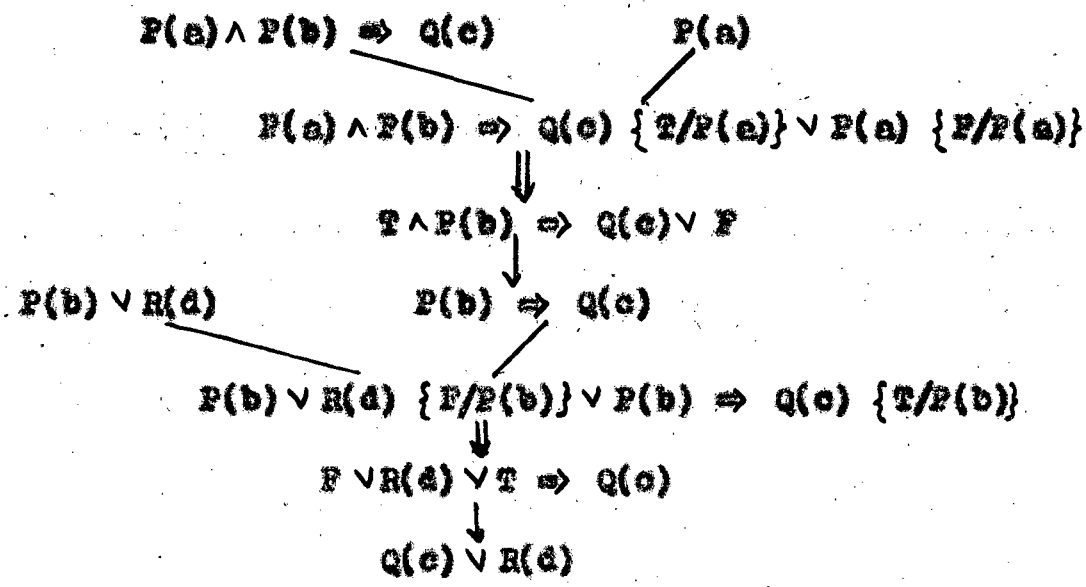
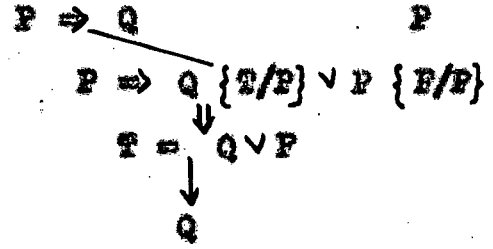


Fig. 3(a).

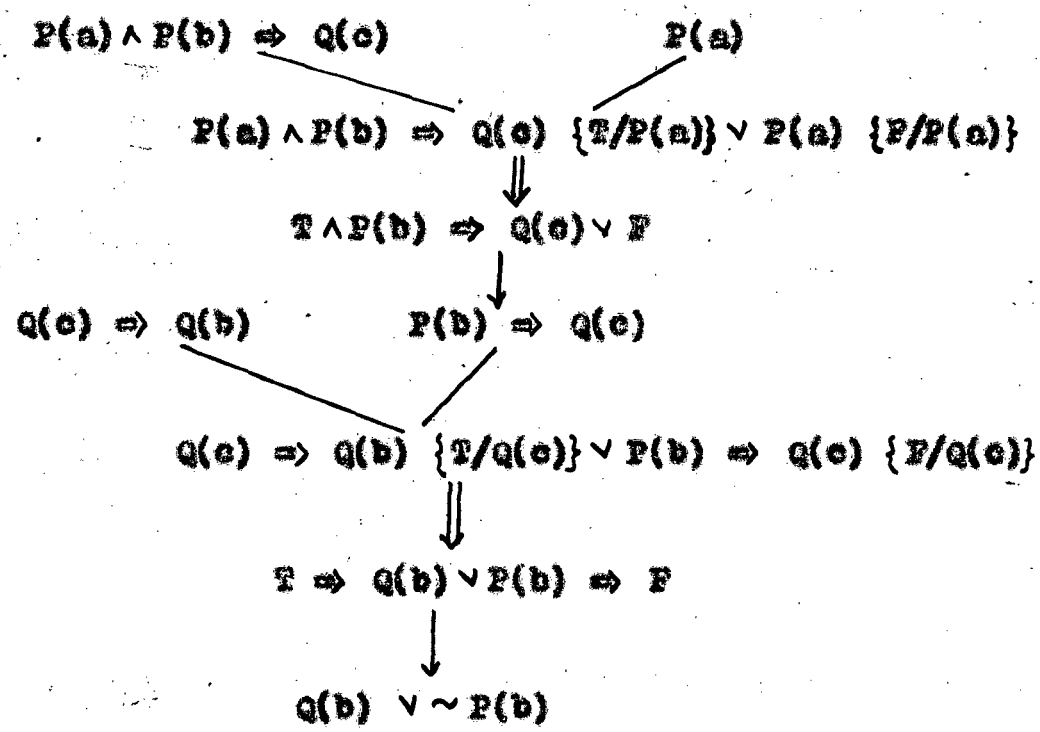


Suppose f is the Propositional abstraction. We have the following abstracted proof:



Note that we lost the literal P from {P,Q}, even though the literal P(b) remains in {P(b), Q(c)} .

Fig. 3(b).



Let f be the propositional abstraction, as before.

We have the following abstracted proof:

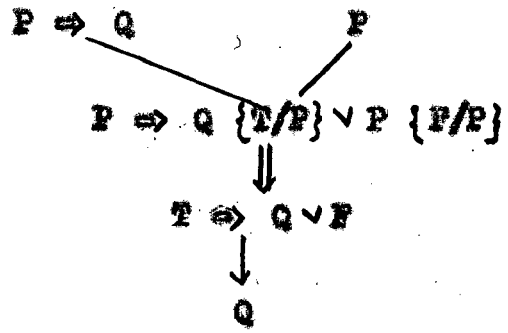


Fig. 3(c).

Although testing if two non-clauses are variant is in general polynomially equivalent to graph isomorphism, in practice this test is not difficult. If variants are not considered to be identical, then there might be many more possible NC-resolvent of two non-clauses, since many NC-resolvent might be variants of each other.

Definition: A NC-resolution proof T is a finite set of nodes together with a set of triples of these nodes. Also, each node N has a label, written label(N), which is non-clause. No two distinct labels of nodes of T may be variant, but the same non-clause may label more than one node of T. If $\langle N1, N2, N3\rangle$ is a triple of nodes of T, then we require label (N3) NC-resolvent of label (N1) and label (N2). The nodes N1 and N2 may be identical. We refer to the set of triples of T by Res(T) and the set of nodes by Nodes(T). Each triple is called a NC-resolution. We require that if $\langle N1, N2, N3\rangle \in$ Res(T), then $\langle N2, N1, N3\rangle \in$ Res(T). A node T that is not the third component of any triple T is called initial node of T. The label of such a node is called an initial non-clause of T. A node that is not the first or second component of any triple of T is called a terminal node of T. The label of such a node is called a terminal non-clause of T. Finally, we require that

there be a function 'depth' mapping from nodes of T into non-negative integers, such that:

(a) depth (N) = 0 for all initial nodes N of T.

(b) depth (N) = 1+min max(depth(N1),depth(N2):

$\langle N1, N2, N3\rangle \in Res(T)$).

We call depth (N) the depth of the node N. Note that a single node by itself, with a label, is a permissible NC-resolution proof. The existence of a depth function insure that no node is used to derive itself. Thus the proof has no 'Loops'. We sometimes refer to the triple $\langle N1, N2, N3\rangle$ of a proof T by $\langle NC1, NC2, NC3\rangle$, where NC1 = Label(N1), NC2 = Label(N2) and NC3 = Label(N3).

Definition: If S is a set of non-clauses, then a NC-resolution proof from S is a NC-resolution proof in which the labels of all initial nodes are non-clauses in S.

This is an example of a NC-resolution proof from $P1, P1 \Rightarrow P2, P2 \Rightarrow P3$. Let the proof T be defined to have nodes N1, N2, N3, N4 and N5. The labels are P1, P1 $\Rightarrow$ P2, P2 $\Rightarrow$ P3, P2 and P3 respectively. We write N:NC to indicate that NC is label of node N. The triples are $\langle N1, N2, N4\rangle$,

⟨N2, N1, N4⟩ , ⟨N4, N3, N5⟩ , ⟨N3, N0, N5⟩ this corresponds to the proof, which is shown in Fig. 4.

Here N1, N2 and N3 are the initial nodes. Also P1, P1 ⇒ P2, P2 ⇒ P3 are initial non-clause. The only terminal node in this example is N5 and P3 is the only terminal non-clause. (There may be more than one terminal node or non-clause in some examples). The depth of N1, N2 and N3 is 0, the depth of N4 is 1 and the depth of N5 is 2.

Definition: If T1 and T2 are NC-resolution proofs, we write T1 T2 indicate that Res (T1) is a subset of Res(T2) and that Node (T1) is a subset of Node(T2). We call T1 a subproof of T2. If all initial nodes of T1 are also initial nodes T2, we call T1 an initial subproof of T2.

Example of subproof and initial sub-proof are shown in Fig.5 and Fig.6 respectively.

Definition: If T is a NC-resolution proof and ⟨N1,N2,N3⟩∈ Res(T) then we call N1 and N2 predecessors of N3, we call N3 successor of N1 and N2.

Definition: The depth of NC-resolution proof T is the maximum depth of any node of T. The depth of NC-resolution ⟨N1,N2,N3⟩

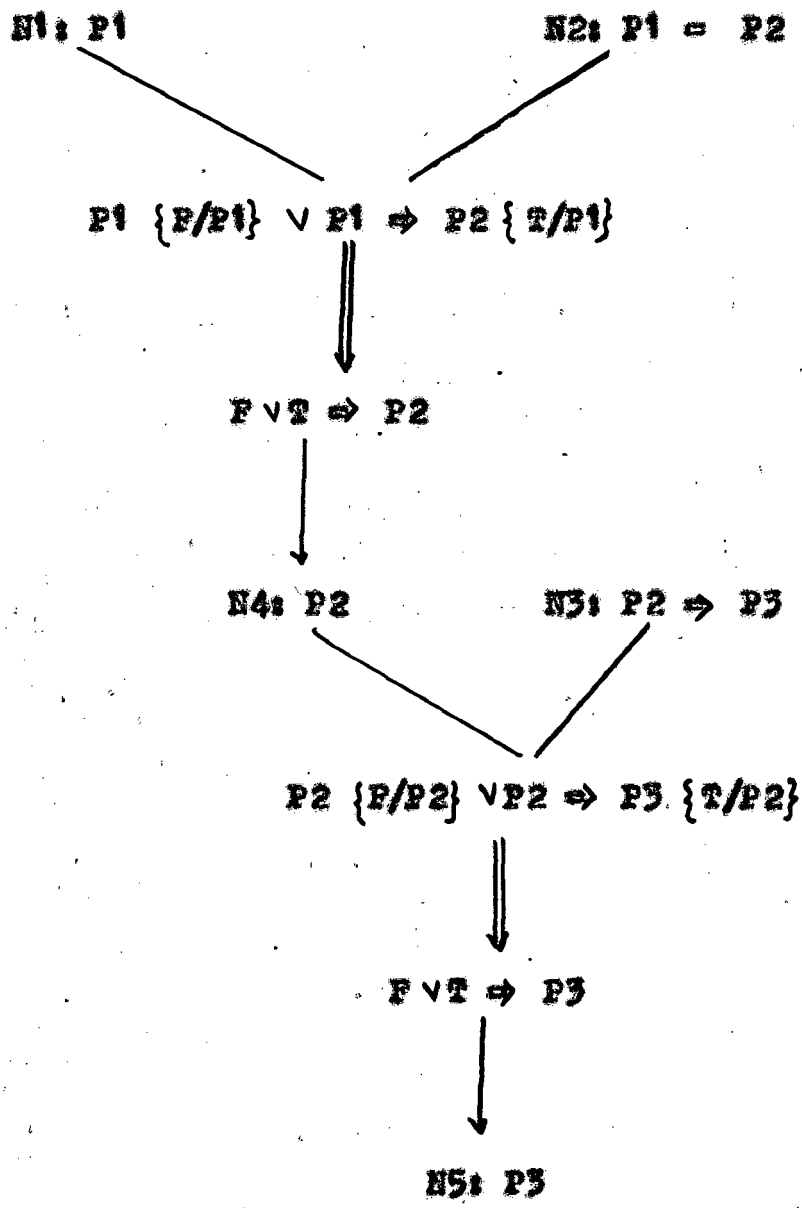

N1: P1
N2: P1 ⇒ P2
P1 {F/P1} ∨ P1 ⇒ P2 {T/P1}
F ∨ T ⇒ P2
N4: P2
N3: P2 ⇒ P3
P2 {F/P2} ∨ P2 ⇒ P3 {T/P2}
F ∨ T ⇒ P3
N5: P3

Fig. 4.

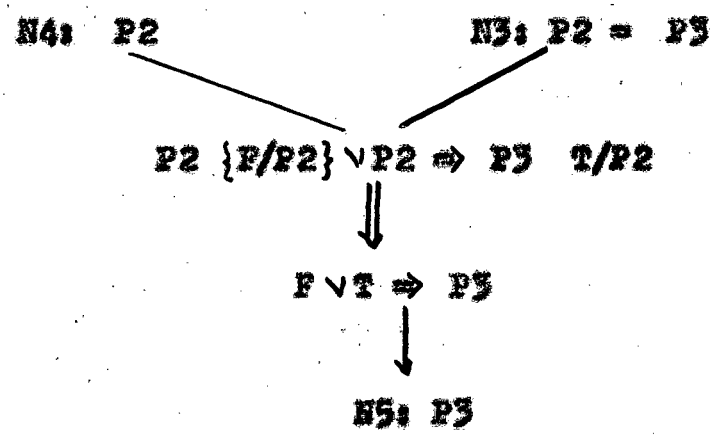


Fig.5. It is a subproof of Fig.4.

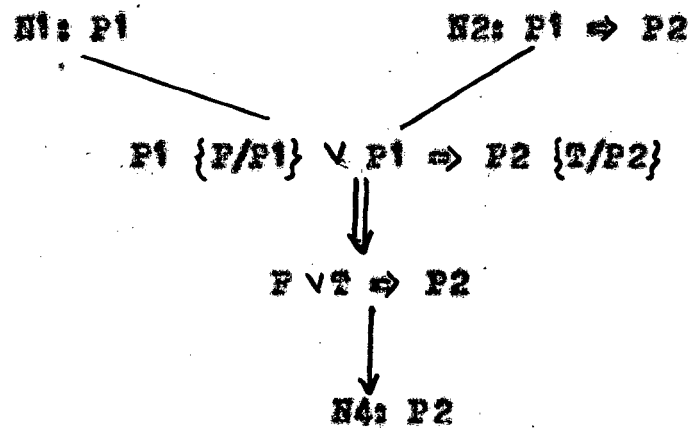


Fig.6. It is an initial subproof of Fig.4.

of T is the depth of N3 in T. If label (N) = NC, we often refer to the depth of NC instead of the depth on N. Note that NC may have more than one depth in T.

If T is a NC resolution proof and non-clause NC is the label of some node of T, then we say that NC appears in T. Speaking informally, we say that NC is an element of T.

Definition: If the terminal non-clause NC of a NC-resolution proof T is unique, then we define Result (T) to be NC. Note that NC may appear at more than one node of T, but NC must appear at the terminal node of T.

Definition: Suppose S is a set of non-clause and NC is a non-clause. A NC-resolution proof of NC from S is a NC-resolution proof T from S such that NC is the label of some node of T.

Definition: Suppose T is a NC-resolution proof from S. Then we say T is a minimal NC-resolution proof from S if

(a) T has exactly one terminal node and

(b) For each non initial node N of T, there are nodes N1, N2 of T such that $\langle N1,N2,N\rangle \in Res(T)$ and $\langle N2,N1,N\rangle \in Res(T)$ and no other triples of T have N as a third component (N1 and N2 may be identical).

Note that if T is a minimal proof from S, then Result(T) is defined. A minimal proof need not be minimal in the usual sense. It could be that the terminal non-clause of T appear at more than one node of T. That is some lemmas may have been derived more than once. We say T is a minimal proof of NC from S is T is a minimal proof from S and Result (T) = C.

Examples of non-terminal proof are shown in Fig.7(a) and 7(b).

Definition: Suppose T and T' are two NC-resolution proofs. Then we say T and T' has same shape if there is a relation ' $\approx$ ' between nodes in T and nodes 'T' such that ' $\approx$ ' has the following properties:

(a) For all nodes N of T there exists a node N' of T' such that N $\approx$ N' and for all nodes N' of T' there exists a node N of T such that N $\approx$ N'.

(b) Suppose $\langle N1, N2, N3 \rangle$ is a NC-resolution of T (That is an element of Res(T)) and $\langle N1', N2', N3' \rangle$ is a resolution of T'. Suppose N3 $\approx$ N3'. Then either N1 $\approx$ N1' and N2 $\approx$ N2' or N1 $\approx$ N2' and N2 N1'. Both may be true iff N1 = N2 or N1' = N2'.

(c) Suppose N is a node of T and N' is a node of T'

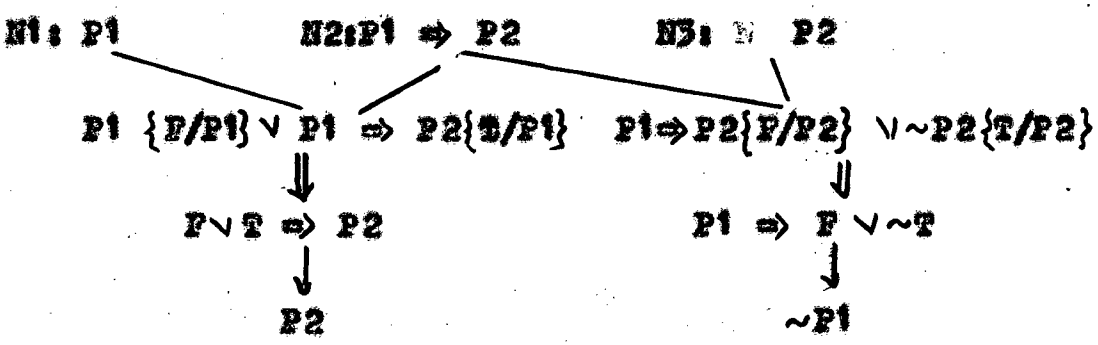


Fig.7(a). This is not minimal because it has two terminal nodes.

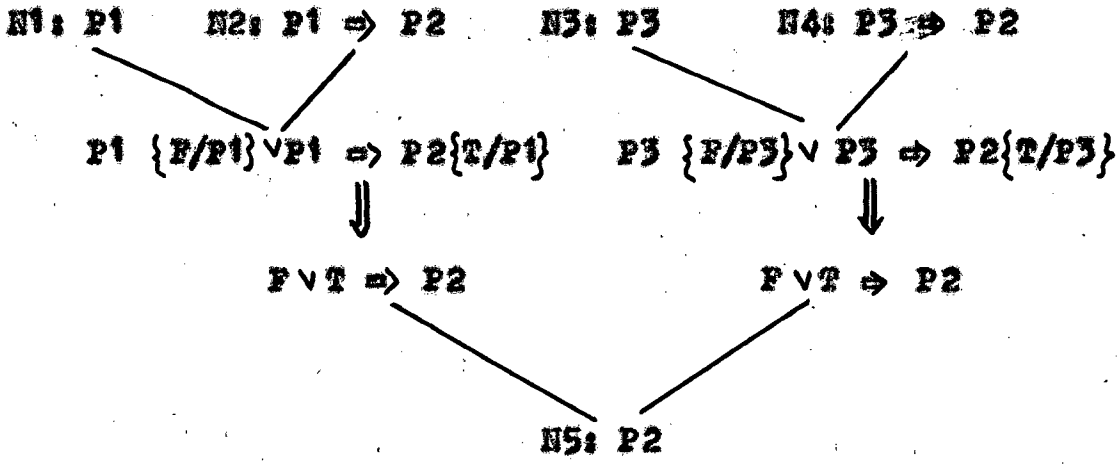


Fig.7(b). This is not minimal because the node N5 is produced by two different NC-resolution.

and $N \approx N'$. Then N is initial in T iff N' is initial in T', and N is terminal in T iff N' is terminal in T'.

(d) The relation ' $\approx$ ' is a 1-1 relation between terminal notes of T and T'.

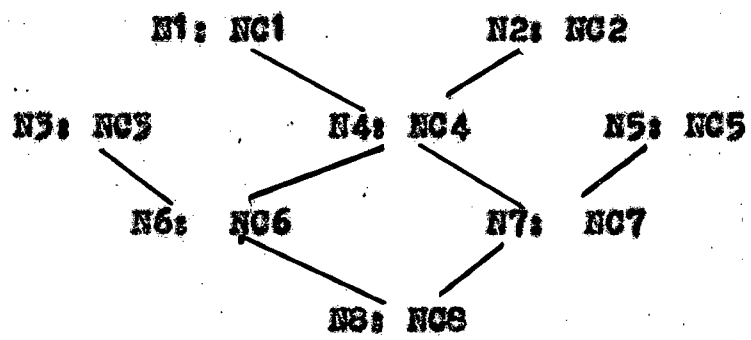
In this case, we call ' $\approx$ ' a shape correspondence between T and T'. Property (1) of shape correspondence is actually a logical consequence of properties (2), (3) and (4). The basic idea of shape correspondence is that if T and T' are expressed sets of NC-resolution proof trees, then there sets of trees will have the same shape (ignoring the labels of the nodes in the trees). We write $T \approx T'$ if ' $\approx$ ' is a shape correspondence between T and T'. If $T \approx T'$ then the depth of T and T' are equal.

Here is an example of shape correspondence between proofs having different 'structure'.

The node related by a shape correspondence as follows (See Fig. 8).

T	T'
N1	N9,N13
N2	N10,N14

T


N1: NC1
N2: NC2
N3: NC3
N4: NC4
N5: NC5
N6: NC6
N7: NC7
N8: NC8

T'

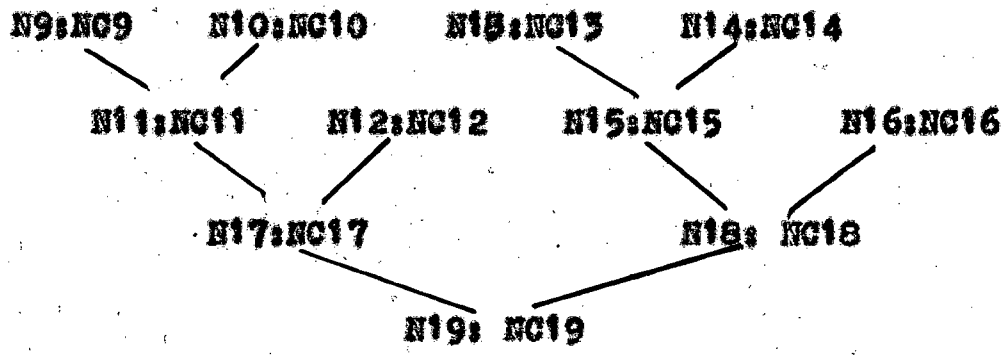

N9:NC9
N10:NC10
N13:NC13
N14:NC14
N11:NC11
N12:NC12
N15:NC15
N16:NC16
N17:NC17
N18: NC18
N19: NC19

Fig. 8.

N3	N12
N4	N11,N15
N5	N16
N6	N17
N7	N18
N8	N19

Suppose $\langle N1,N2,N3\rangle \in Res(T)$ and $\langle N1',N2',N3'\rangle \in Res(T')$. Then we say $\langle N1,N2,N3\rangle \approx \langle N1',N2',N3'\rangle$ if $N1 \approx N1'$, $N2 \approx N2'$ and $N3 \approx N3'$ or if $N1 \approx N2'$, $N2 \approx N1'$ and $N3 \approx N3'$.

If T1, T2 are NC-resolution proofs and ' $\approx$ ' is shape correspondence between T1 and T2, then we say $NC1 \approx NC2$ iff there exist node N1 of T1 and node N2 of T2 such that NC1=label(N1) and NC2=label (N2), and $N1 \approx N2$.

Suppose R is a binary relation on non-clauses. We extend R to a binary relation on resolution proof in the following way:

R(U,U') is true iff U and U' have the same shape, and

there exists a shape correspondence ' $\approx$ ' between U and U' such that NC $\approx$ NC' implies R(NC,NC') for all non-clause NC in U and all non-clause NC' in U'.

Suppose R1 and R2 are binary relation on non-clauses and U and U' are NC-resolution proof then we say (R1:R2)(U,U') if there is a shape correspondence ' $\approx$' between U and U' such that

(a) If N is an initial node of U and N' is the initial node of U' and N $\approx$ N', then R1(label(N), label(N')) is true.

(b) If N is a non initial node of U and N' is a non initial node of U' and N $\approx$ N', then R2(label(U), label(U')) is true.

This allows us to specify a different relation between initial non-clauses than between non initial non-clauses. In the example above R(T,T') would be true if R(NC1, NC9), R(NC1, NC13), R(NC2, NC10), R(NC2, NC14), R(NC5, NC12) etc. were all true. Also (R1:R2) (T,T') would be true if R1(NC2, NC14), R1(NC1, NC13), R1(NC2, NC10) etc. were true and if R2(NC4, NC11), R2(NC4, NC15) etc were true.

Definition: Let R be the identity relation. That is R(NC1, NC2) is true iff NC1 and NC2 are the same non-clause. Then we say proofs U and U' are isomorphic iff R(U,U') is true. Thus isomorphism proofs have the same non-clauses at corresponding nodes, although the structure may differ in an essential ways. Note that if U and U' isomorphic, they have the same depth.

4.11 A Procedure in Abstracted Proof

This is an incomplete theorem proving procedure based on abstraction. Suppose f is an abstraction mapping on a set S of non-clauses. Suppose T is a proof of NC' from S, and ND' $\in$ f(NC'). We know by theorem 4.9.1 that there is a proof U of f(S) of some non-clause NB' subsuming ND'. It will frequently turn out that T and U have the same shape (See Fig.9). Therefore, one way to search for proofs of NC' from S is to search for proofs U of some such non-clause NB' from f(S) and try to find a proof T of NC' from S of the same shape as U. The proofs from f(S) are 'abstracted proof' which help to guide us in the search for proofs from S.

We make this relation between T and U precise, and use it as the basis of a theorem proving program 'ndfind'.

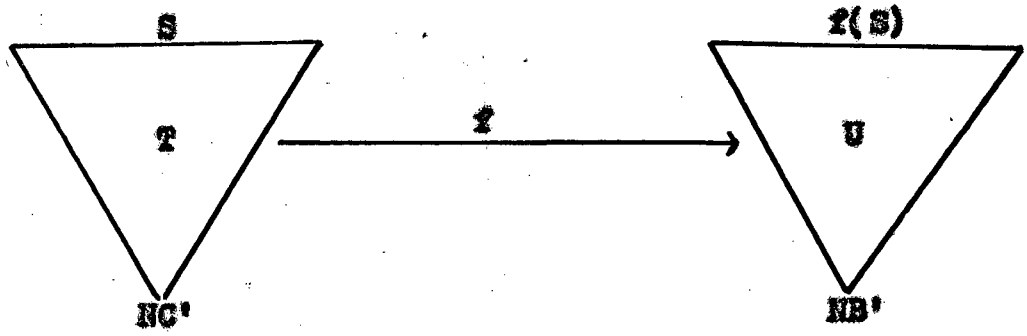


Fig. 9.

Definition: Suppose f is an abstraction mapping. Let R1(NB,NC) be the relation NB $\in$ f(NC) and let R2(NB,NC) be the relation 'NB subsumes an element of f(NC)'. We write $T \longrightarrow_f U$, where f is the propositional abstraction. Fig.3(b) in section 4.9 gives a proof T not having an abstraction U such that $T \longrightarrow_f U$.

The procedure 'ndfind', given an abstracted proof T such that $T \longrightarrow_f U$. Suppose S is a set of non-clauses and U is a proof from f(S) with each node N in U, 'ndfind' keeps a set non-clauses(N) of non-clauses NC having the following properties:

There is an initial, sub proof U1 of U and minimal proof T1 from S such that $T1 \longrightarrow_f U1$ and such that NC=Result(T1) and N is the unique terminal node of U1. Note that NC is derived from S by NC resolution (See Fig.10).

Procedure ndfind (U,S,f):

[assume that for all initial nodes N of U, non-clauses(N) = $\{NC \in S \mid label(N) \in f(NC)\}$ and that non clause (N) = for non initial nodes N of U]

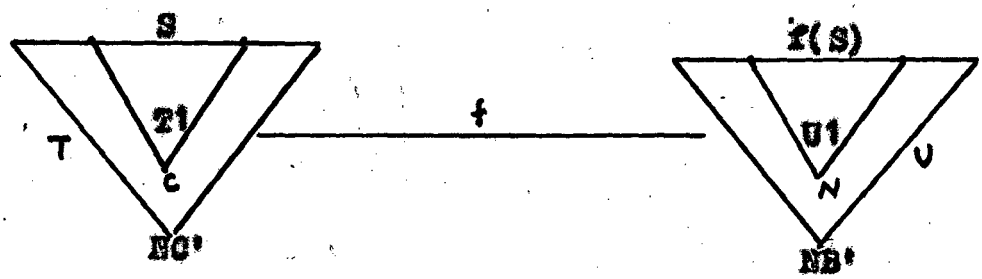


Fig. 10.

Loop

While (there exist nodes N1,N2,N of U and
non-clauses NC1,NC2,NC such that

(1) $\langle N1,N2,N\rangle \in Res(U)$

(2) NC1 $\in$ non-clause(N1) and NC2 non-clauses(N2)

(3) NC is a NC-resolvent of NC1 and NC2

(4) NC $\notin$ non-clauses (N)

(5) Label(N) subsumes an element of f(NC));#

add C to non-clause (N)

repeat

end ndfind;

Suppose we are looking for a proof of non-clause NC' from set S of non-clauses using abstraction f. To do this we choose ND' in f(NC) and execute the following steps for d=1,2,3... until a proof NC' is found.

(1) Let V be the set of depth d NC-resolution proofs from f(S) generated exhaustively.

(2) Let V1 be the initial sub proof of V containing all NC resolutions that contribute to depth d proof of

something subsuming ND'. Note that V1 may have more than one terminal node.

(3) Initialize non-clause (N) for nodes N of V1 as required by 'ndfind'.

(4) ndfind (V1, S, f)

We call this procedure 'proofsearch1'. Note that it is incomplete.

Example 1

Suppose the set S of input non clauses is $P(x) \wedge Q(x) \Rightarrow R(x), P(x), Q(a)$ and if the propositional abstraction. Suppose we are looking for a proof of R(a) from S we generate proofs from f(S) of something subsuming R and then apply 'ndfind'. Note that $f(S) = P \wedge Q \Rightarrow R, P, Q$. Assuming that the abstracted space is searched exhaustively to depth 2 we obtain the abstracted proof U: which is shown in Fig. 11(a).

The nodes N1 and N3 have been drawn twice for clarity. To use 'ndfind' we first set non-clause (N1) to P(x), non-clause (N2) to $P(x) \wedge Q(x) \Rightarrow R(x)$ non-clause (N3) to Q(a) and

non-clause (N4), non-clause (N5) and non-clause (N6) to Φ (the empty set). When 'ndfind' is executed it will add $Q(x) \Rightarrow R(x)$ to non-clauses (N4), $P(a) \Rightarrow R(a)$ to non clause (N5) and $R(a)$ to non-clause (N6) twice.

Next we will show that 'ndfind' is not complete.

Example 2

Suppose the set S of input non-clauses is

$P(a) \wedge P(b) \Rightarrow Q(c)$, $P(b) \vee R(d)$, $P(a)$ and we are using the propositional abstraction. Suppose we are looking for a proof of $Q(c) \vee R(d)$ from S. We obtain the abstracted proof U at depth 1; which is shown in Fig. 11(b).

When 'ndfind' executes, it will generate $P(b) \Rightarrow Q(c)$ at N4 and $P(a) \vee Q(c)$ will not generated. At depth 2, $Q(c) \vee R(d)$ will not be generated by 'ndfind' even though it may be obtained by a simple depth 2 proof from S.

4.12 Logical Consequences

The Algorithm 'proof search 1' can be modified to test if a non clause NC' is a logical consequence of set S of non clauses. This can be done by making use of the following fact [24].

If NC' is a logical consequence of S, then there is a non-clause

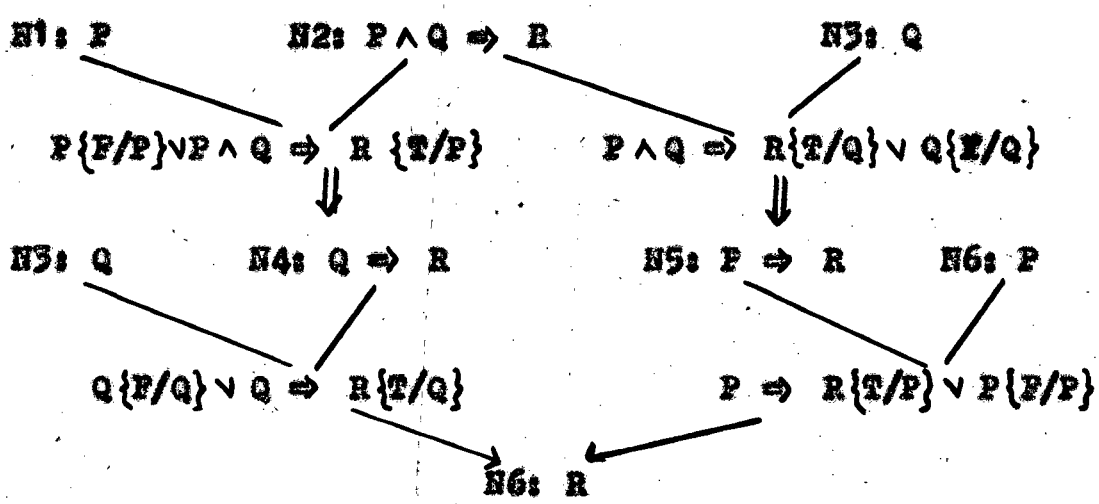


Fig. 11(a).

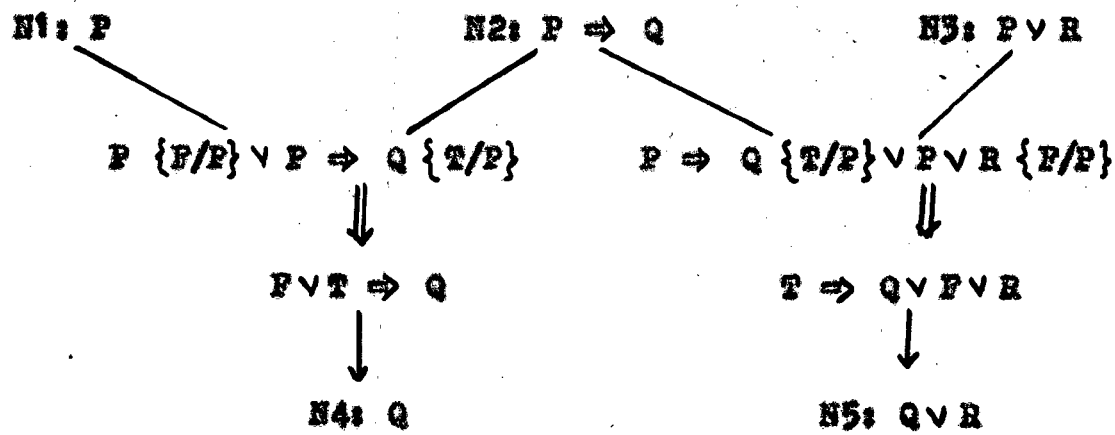


Fig. 11(b).

NC" derivable from S by NC-resolution such that NC" subsumes NC'. Also if ND' is an abstraction of NC', there will be an abstraction ND" of NC" such that ND" subsumes ND' by property (3) of abstraction. Therefore, there is a proof from f(S) of a non-clause NB' subsuming ND". Therefore NB' subsume ND'. Hence if we generate abstracted proofs of non-clauses NB' subsuming ND' and call 'ndfind', we may generate a proof of something subsuming NC'. If such a proof is generated, we know that NC' is a logical consequence of S. This approach is incomplete.

4.13 MNC-Resolution and M-Abstractions

We now extend the concept of abstraction to "multinon-clause" which are multi-sets of literals. That is, we allow a literal to occur more than once in a multinon-clause. The concept of abstraction is adopted to multinon-clause to obtain 'm-abstraction' mapping.

Definition: A multiset M is a set S together with a function g mapping S into the set of positive integers. We refer to S as Set (M) and $x \in S$, g(x) denoted by mult (x, M). By convention, mult (x, M) = 0 if $x \in S$.

Intuitively, a multiset is a set in which elements can

occur more than once. For $x \in S$, mult (x, M) tells 'how many times' x occure in M. We write M as $\{n_1 * x_1, \ldots . n_k * x_k\}$ where mult $= \sum_{j, x_j = x} n_j$, also $1 * x$ is written as x.

We often regard an ordinary set A as a multiset M in which each element of A occurs exactly once, and in which no other elements occur.

The size $\overset{M}{k}$ of a multiset $M = \{n_1 * x_1, \ldots . n_k * x_k\}$ is defined to be $\sum_{i=1}^{k} n_i$.

Definition: If M1 and M2 are multisets, then their union M1 + M2 is defined by

$$\text{mult}(x, M1 \oplus M2) = \text{mult}(x, M1) + \text{mult}(x, M2)$$

Their intersection M1 + M2 is defined by

$$\text{mult}(x, M1 \circledcirc M2) = \text{Min}(\text{mult}(x, M1), \text{mult}(x, M2)).$$

Their difference M1-M2 is defined by

$$\text{mult}(x, M1 - M2) = \max(0, \text{mult}(x, M1) - \text{mult}(x, M2)).$$

Definition: If M1 and M2 are multisets, then we write $M1 \subset M2$ (M1 is a sub-multiset of M2) if for all x, $\text{mult}(x, M1) \leq \text{mult}(x, M2)$).

Definition: If M is a multiset and g is a mapping from Set (M) into a set N, then $g(M) = \oplus_{x \in M} g(x)$. Thus mult(y, g(M)) = $\sum_{x, g(x)=y}$ mult(x,M) and $|g(M)| = M$.

Note that for multisets M1 and M2 $g(M1-M2)=g(M1)-g(M2)$ if $M2 \subset M1$. This is not true of ordinary sets, however.

Definition: A multinon-clause (or mn-clause) is a multiset of literals. That is, with each literal in the mn-clause, a multiplicity is kept, which is positive integer telling how many times the literal occurs in the multinon-clause. We can write a multi non-clause by writing each element the number of times it occurs in the multi non-clause.

Definition: If NC is a multi non-clause and α is a substitution, then $NC\alpha$ is $\{L\alpha : L \in C\}$ where L is counted the right number of times. That is mult(L1, $NC\alpha$) = $\sum_{L \in NC.L=L1}$ mult(L,NC). Thus $|NC\alpha| = |NC|$, and if $NC = \{L_1, L_2 \ldots . L_n\}$ then $NC\alpha = \{L_1\alpha, L_2\alpha \ldots . L_n\alpha\}$.

Definition: Suppose NC1 and NC2 are multi non-clauses. Suppose $A1 \subset NC1$ and $A2 \subset NC2$ (This means that every literal occurs no more times in A1 then in NC1 and similarly for A2). Suppose

there exist substitution $\alpha 1$ and $\alpha 2$ such that for all literal L, Set (A1 $\alpha 1$) = $\{L\}$ and Set (A2 $\alpha 2$) = $\{L\}$. Let $\alpha 1$ and $\alpha 2$ be most general such substitutions. If L is positive in NC1 and negative in NC2 then (NC1$\{F/A1\}$)$\alpha 1$ + (NC2$\{T/A2\}$)$\alpha 2$, is an MNC-resolvent of NC1 and NC2. Similarly if L is negative in NC2 and positive in NC1 then (NC1 $\{T/A1\}$) $\alpha 1$ + (NC2$\{F/A1\}$)$\alpha 2$ is an MNC-resolvent of NC1 and NC2.

Ordinary non-clauses can be viewed as multi non-clauses in which the multiplicity of each literal in the non-clause is 1. We have the following results concerning the relation of MNC resolution to ordinary NC-resolution.

Theorem 4.13.1

Suppose NC3 is an ordinary NC-resolvent of non-clauses NC1 and NC2. Suppose ND1 and ND2 are mn-clauses such that Set(ND1) = NC1 and Set (ND2) = NC2. Then there is an MNC-resolvent ND3 of ND1 and ND2 such that Set (ND3) = NC3.

Theorem 4.13.2

Suppose non-clause NC is derivable from Set S of non-clauses by ordinary NC-resolution. Then there is a mn-clause ND derivable from S by MNC-resolution such that

Set (ND)=NC (In derivation of ND, we consider non-clauses of S to be mn-clauses). Note that if NC=NIL, then ND=NIL also.

Theorem 4.13.3

Suppose mn-clause ND3 is an MNC-resolvent of mn-clauses ND1 and ND2. Then some ordinary NC-resolvent of Set (ND1) and Set (ND2) subsumes Set (ND3).

Theorem 4.13.4

Suppose S is a set of multi non-clauses and mn-clause ND is derivable from S by MNC-resolution., (For thid derivation we consider the non-clause of S to be mn-clauses). Then there is an ordinary non-clause NC-derivable from S by ordinary NC-resolution, such that NC subsumes Set (ND).

Definition: An m-abstraction is a mapping f from multi non-clause to (ordinary) sets of multi non-clauses. Satisfying the following properties:

(i) If NC3 is an MNC resolvent of NC1 and NC2, and ND3 $\in$ f(NC3), then there exist ND1 $\in$ f(NC1) and ND2 $\in$ f(NC2) such that ND3 is an instance of some MNC-resolvent of ND1 and ND2.

(ii) f(NIL)=NIL

The following two results, analogous to theorem 4.6.1 and theorem 4.6.2 for ordinary abstraction, show that m-abstraction are also easy to construct.

Theorem 4.13.5

Suppose Φ is a mapping from literals to literals. Let us extend Φ to a mapping from mn-clause to mn-clauses as follows:

$$\Phi(\{n_1*L_1,\ldots n_k*L_k\}) = \{n_1*(L_1),\ldots,n_k*(L_k)\}\ .$$

Thus $|\Phi(NC)| = |NC|$ but need not be one-to-one.

Suppose satisfied the following properties:

(1) $\Phi(\sim L) = \sim\Phi(L)$ for all Literals

(2) $\Phi(F) = F$

(3) $\Phi(T) = T$

(4) If a mn-clause ND is an instance of the mn-clause NC, then $\Phi(ND)$ is an instance of $\Phi(NC)$.

Then the mapping f defined on mn-clauses by $f(NC) = \{\Phi(NC)\}$ is an m-abstraction mapping.

Proof: Similar to the proof of theorem 4.6.1

Theorem 4.13.6

Suppose F is a set of mappings from literals to literals. For each $\Phi \in F$, extend Φ to a mapping from mn-clause to mn-clause as in the preceding theorem. Suppose that all $\Phi \in F$, $\Phi(\sim L) = \sim\Phi(L)$ for all literals L, $\Phi(F)=F$ and $\Phi(T)=T$. Suppose also that if mn-clause ND is an instance of mn-clause NC, then for all $\Phi 2 \in F$ there exists $\Phi 1 \in F$ such that $\Phi 2(ND)$ is an instance of $\Phi 1(NC)$. Define mapping f on mn-clauses by $f(NC) = \{\Phi(NC) : \Phi \in F\}$. Then f is an m-abstraction mapping(Note that f(NC) is an ordinary set of mn-clauses).

Proof: Similar to the proof of theorem 4.6.2

If f is an m-abstraction as in this theorem, we say f is defined in terms of literal mappings.

Theorem 4.13.7

If f is an m-abstraction defined in terms of literal mappings, then f also satisfies the following property:

For all mn-clauses NC1 and NC2, if NC1 subsumes NC2, then for all $ND2 \in f(NC2)$ there exists $ND1 \in f(NC1)$ such that ND1 subsumes ND2.

(For mn-clauses, we say ND1 subsumes ND2 if there exists substitution such that $ND1\theta \in ND2$, that is for all $L \in ND1\theta$ mult $(L, ND1\theta) \leq$ mult $(L, ND2)$).

4.14 Examples of m-abstractions

These m-abstractions are obtained from the abstraction presented in section 4.7 by counting each literal the right number of times.

(1) The ground m-abstraction

If NC is the mn-clause $\{L_1 \ldots\ldots\ldots\ldots L_k\}$, then $f(NC) = \{(L_1\theta, L_2\theta, \ldots L_k\theta) : L_i\theta$ are ground literals, for $1 \leq i \leq k\}$.

(2) The propositional m-abstraction

If NC is the mn-clause $\{L_1 \ldots\ldots L_k\}$, then f(NC) is ND where ND is the mn-clause $\{L_1' \ldots\ldots L_k'\}$ and L_i' is obtained from L_i by deleting all arguments of the predicate symbol. Thus if L_i is $P(t_1 \ldots\ldots t_n)$ then L_i' is P and if L_i is $\sim P(t_1 \ldots t_n)$, then L_i' is $\sim P$.

Similarly we can define m-abstractions based on renaming of predicate or function symbols, changing the signs of literals, changing the order of the arguments of various

function or predicate symbols, or removing arguments from various function or predicate symbols.

We can define the composition $f_1 f_2$ of m-abstraction f_1 and f_2 and show as before that it is an m-abstraction if f_1 and f_2 are. Also, the union of two m-abstraction is an m-abstraction. Moreover, it is easy to show that if f_1 and f_2, are m-abstraction defined in terms of literal mappings, then the union and composition of f_1 and f_2 are also defined in terms of literal mappings.

4.15 M-abstraction of MNC-resolution proof

We now indicate how m-abstraction can be used to guide the search for a proof of an mn-clause NC from a set S of mn-clauses. We omit some details because the development is analogous to that for ordinary abstraction. The concept of an m-abstraction proof the depth of an m-abstraction proof etc. are defined in a way analogous to the way these concepts were defined for ordinary NC-resolution. We denote the nodes of an MNC-resolution of V by MRes(V). An MNC-resolution of V is a triple $\langle N1, N2, N3\rangle$ of nodes of V such that label (N3) is an MNC-resolvent of the mn-clause label (N1) and label (N2). We require that if $\langle N1, N2, N3\rangle \in$ MRes(V), then $\langle N2, N1, N3\rangle \in$ MRes(V),

as before. Also if M1 and M2 are two relations between mn-clauses and V1 and V2 are two MNC-resolution proofs, we define (M1;M2)(V1,V2) as before. We abreviate (M;M)(V1,V2) as M(V1,V2). If f is an m-abstraction mapping and S is a set of mn-clauses, we write f(S) to denote the set $\bigcup_{NC \in S} f(NC)$. Note that f(S) is an ordinary set of mn-clauses.

Definition: Suppose T and V are MNC-resolution proofs. Suppose f is an m-abstraction mapping. Let M1(NB,NC) be the relation 'NB f(NC)' and let M2(NB,NC) be the relation 'NB has an instance in f(NC)'. Then we write $T \longrightarrow_f U$ if (M1;M2)(U,T) is true. We think of U as an abstraction of the proof T. Note that T and U will have the same shape and depth. Also, if T is a proof from S, then U is a proof from f(S).

Theorem 4.15.1

Suppose T is a minimal MNC-resolution proof of an mn-clause NC' from a set S of mn-clauses. Suppose f is an m-abstraction mapping. Then for every mn-clause ND' in f(NC'), there exists an MNC-resolution proof U from f(S) such that $T \longrightarrow_f U$ and such that Result (U) is defined and has ND' as an instance (See Fig. 12).

This result relates to ordinary NC-resolution in the following way: Suppose non clause NC' is devivable from set S of non-clauses by ordinary NC-resolution. Suppose f is an m-abstraction mapping. Then there exists an mn-clause NC1 such that Set (NC1)=NC', and such that for all ND'$\in$ f(NC1), a non-clause having ND' as instance devivable from f(S) by MNC-resolution. Examples are discussed in Fig.13(a) and 13 (b).

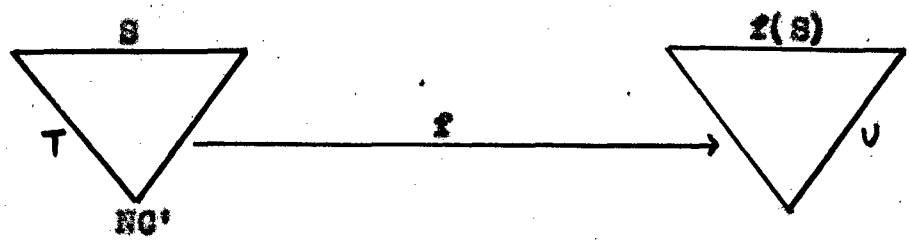
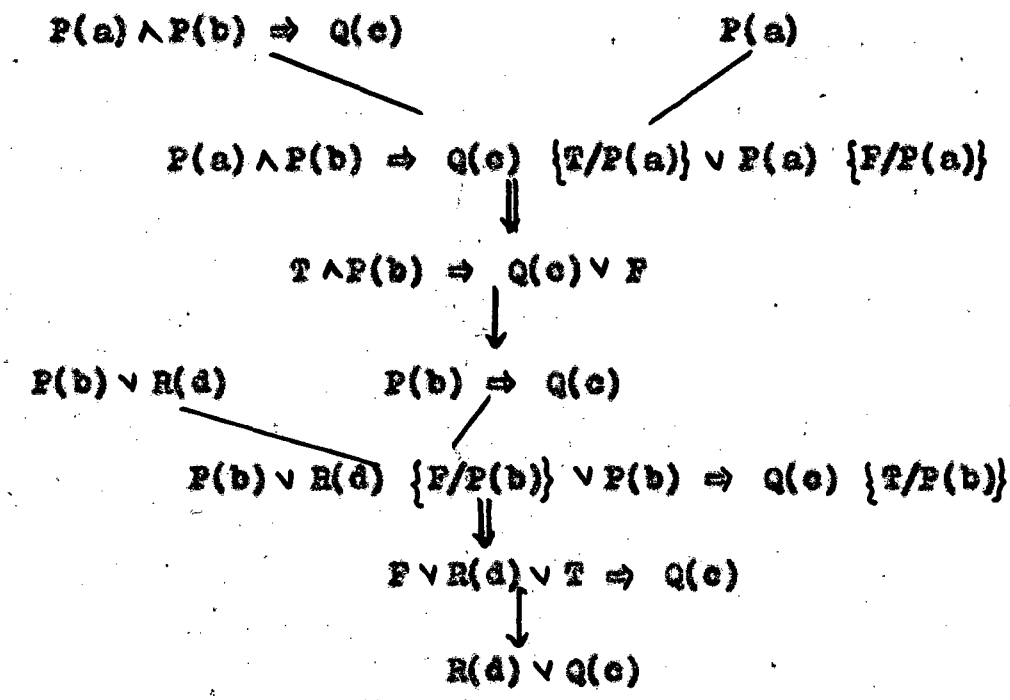


Fig. 12.



If we use the propositional m-abstraction, we obtain the following m-abstracted proof:

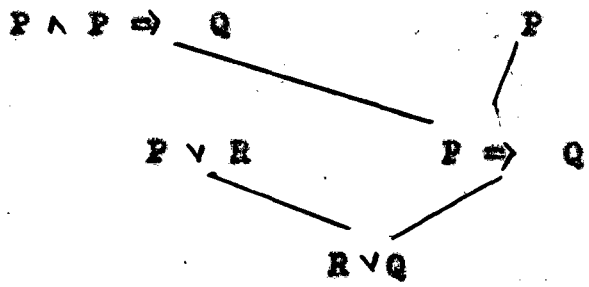
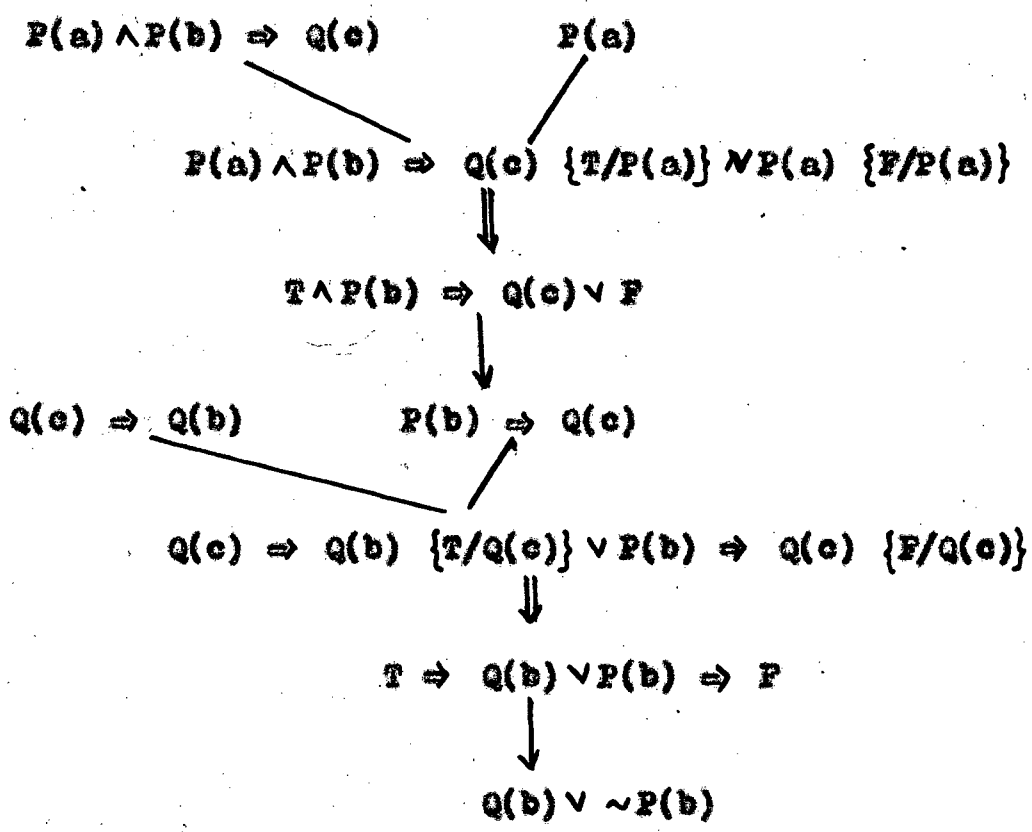


Fig. 13(a)



Let f be the propositional m-abstraction as before, we have the m-abstrated proof:

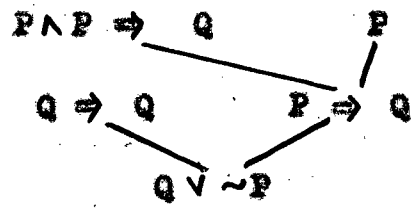


Fig. 13(b)

CONCLUSION

The concept of abstraction, m-abstraction leads to a new complete uniform proof procedure for the first order predicate calculus. These strategies make use of a simplified from a set of non-clauses to guide the search for a proof from the original set of non-clauses. The basic idea is to construct an outline of a proof and fill in the detail later. This is a global strategy. Each step of the search is controlled in a meaningful, non-trival way by the structure of the problem as a whole rather than by local information such as whether two non-clauses can resolve according to certain strategy. Also, near the end of the search the abstracted non-clauses are more restricted than in the middle because an abstraction of the 'goal non-clause' must be devivable from them in a fewer step. Thus the search space tends to get small as the depth of inference increases towards its maximum value, even though the strategy is based on forward reasoning. This contrast the conventional strategies, in which the search space seems to grow exponentially in size with increasing depth. The use of abstraction also allows for the possibility of levels of abstraction, each level keeping less information than the preceeding level. The search at each level can be guided by the search at the next higher level of abstraction. Another

advantage of abstraction is that it automatically selects from the input non-clauses those non-clauses which seem relevant to the given problem. Thus we get the advantage of 'set-of-support' strategies.

The procedure 'ndfind' which is developed does not completely specify the order in which these step will be performed. Secondly 'ndfind' is not complete. First problem can be avoid by applying various resolution strategies such as Locking resolution 8 so that abstracted search can be made smaller. Since 'ndfind' is not complete, we might as well as locking resolution or some similar strategy in abstracted space. Note that what makes 'ndfind' incomplete is when two distinct literals in a non-clause map onto the same literal under an abstraction. Therefore, minimizing the chance that distinct literals will map onto the same literal also maximizes the chance that 'ndfind' will be complete. However it is not a good idea to delete tautologies in the abstracted space or to delete non-clauses in the abstracted space subsumed by other non-clauses in abstracted space.

The following guidlines help in deciding which abstractions are most useful:

(a) The abstracted search space should be small.

(b) The chance that distinct literals of non-clause abstract to the same literal, should be small.

The reason for (a) is to make the abstracted search easier. There are several reasons for (b). If distinct literals usually map to distinct literals, the number of false proofs will be minimized. Also the number of proofs T such that $T \longrightarrow_1 U$ will be minimized which should make 'proof search 1' more efficient. Finally, the deletion of tautologies in the abstracted space is less likely to eliminate useful proofs. However the requirement (a) and (b) are somehow contradictory. If the abstracted space is made small, the chance that distinct literals map onto the same literal is increased. The use of more than one abstraction at the same time can reconcile these requirements to some extent. In order to obtain complete strategies involving more than one abstraction, we need to use m-abstraction and m-clauses. Another way to reconcile requirement (a) and (b) is to use a sequence of abstraction each of which only slightly restrict the search. Suppose we are looking for a proof of NIL from $f_1 \ldots\ldots f_k(S)$, then map these block onto proof of NIL from $f_1 f_2 \ldots\ldots f_{k-1}(S)$ until we obtain a proof of NIL from S. If each f_i only slightly modifies the non-clauses, requirement (b) will be met.

The advantage of m-abstraction mappings and multi non-clauses is that several such mapping can be used together in a natural way in the search proof. However, the search strategies based on m-abstraction turn out to be compatible with other complete resolution strategies such as locking resolution and Pl-deduction. Further work can be done on how to develop procedure 'ndfindm' analogous to 'ndfind' for multi non-clauses and MNC-resolution. This procedure will use m-abstracted proofs as a guide in the search for an MNC-resolution proof. Suppose f is an m-abstraction mapping. We are given MNC-resolution proof U from f(S) and want to find all proofs T from S such that $T \longrightarrow_f U$ with each node N of U, we keep a set mn-clauses (N) of mn-clauses derived from S by MNC-resolution.

One disadvantage of mn-clause is that there are so many of them. The set of ordinary propositional clauses over k distinct predicate symbol is finite, but the set of propositional mn-clauses over k distinct predicate symbol is infinite. This could result in a larger search space for the various abstraction based theorem proving strategies. How how to overcome this problem to some degree, while retaining the advantage of m-abstractions. To achieve this one can use

bounded mn-clauses [38] . They are mn-clauses in which less information about the number of occurrences of literals in a non-clause is kept. One can present abstraction and complete theorem proving strategies based on bounded mn-clauses. The advantage of bounded mn-clauses is that the abstracted search space is often finite and can be searched exhaustively without excessive effort.

Now let us discuss the concepts underlying in Bledsoe's proofs of limit theorems. Bledsoe proved limit theorems without the inclusion of axioms (reference theorems). This is desirable because in automatic theorem provers, the axioms have to be selected by human for each theorem being proved. Of course Bledsoe include the limit heuristic itself which acts like axioms, but it does not hinder the proof of other theorems not requiring it, because it does not release into action unless its needs is detected.

The limit heuristic rule is used to prove many theorems about limits, it is rather an interesting trick. But more interesting and important that the fact that it works some problems is the principle behind it. The principle might be stated:

To establish a conclusion C from several hypothesis,

among which is H, force H tocontribute all it can towards establishing, C and leave a remainder to be established with the help of other hypothesis.

The value of such a 'forcing' technique is two fold. First if one can truely make H contribute all it can toward C, then H is not needed to establish the remainder.

Second, it is implicit in the notion of 'force' that certain facts are used to make an inference in a computational manner. For example the limit heuristic 'uses' many facts about algebra such as the triangle inequality; but these facts are used to compute something, not to make random inferences.

The forcing phenomenon has applications in other areas of theorem proving where two or more hypothesis $H_1, H_2, \ldots H_n$ are needed to establish one conclusion C that cannot be logically divided. In such applications the user must provide a heuristic which will enable the computer to determine how to get a partial result from H_i and leave a remainder C' to be proved by the other hypothesis.

BIBLIOGRAPHY

1. Armer,Paul., Attitudes Towards Intelligent Machine, Computers and Thought; edit by Edward A.Feigenbaum and Jullian Feldman, McGraw Hill,(1963) 292-393.

2. Andrews,P., Refutations by matting, IEEE transaction on Computers C-25 (1976) 801-806.

3. Ballantiyne,A.M. and Bannett,W., Graphing method for topological proofs, University of Texas at Austin, Math Deptt. Memo.ATP (7) (1973).

4. Ballantiyne,A.M. and Bledsoe,W.W., Automatic Proofs of Theorems in Analysis using non-standard Techniques J.ACM 24(3),(1977) 353-374.

5. Bledsoe,W.W., Non-resolution Theorem Proving, Artificial Intelligence 9(1977) 1-35.

6. Bledsoe,W.W., Splitting and Reduction Heuristics in Automatic Theorem proving, Artificial Intelligence 2(1971) 55-77.

7. Bledsoe,W.W., Boyer,R.S. and Henneman,W.H., Computer Proofs of Limit theorems, Artificial Intelligence 3(1972) 27-60.

8. Boyer,R.S., Locking, a restriction of resolution. Thesis University of Texas at Austin Tx (1971).

9. Brown,F.M., Doing Arithmetic without diagram, Artificial Intelligence 8(1977) 175-200.

10. Bundy,A., Doing Arithmetic with Diagrams, Adv.Paper 3rd Int.Joint Conf.Artificial Intelligence (1973) 130-138.

11. Chandrasekaran,B. Artificial Intelligence - The Past decade, Advance in Computers, Vol.13(1975) Ac.Press.

12. Chang,C.L., The Unit Proof and the Input Proof in Theorem proving, J. ACM 17(4), (1970) 698-707.

13. Chang,C.L., Theorem Proving with variable constraints Resolution, Information Sci. 4(1972) 217-231.

14. Chang,C.L. and Slagle, J.R., Using rewriting rules for connection graphs to prove theorems, Artificial Intelligence 12(1979) 159-180.

15. Chang,C.L. and Lee,R.C., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York (1973).

16. Charniak,E., Inference and Knowledge, Computational Semantic, edit by Charniak and Wills.

17. Dixon,J.K., Z. Resolution Theorem Proving with compiled axioms J.ACM 17(4) (1973) 127-147.

18. Dreyfus,H.L., What computers can't do, Hyper Colophen Books (1972) revised edition (1979) p.63.

19. Feigenbaum,E. and Feldman,J.(ed) Computers & Thought McgrawHill (1963) p.2.

20. Fikes, R.E. and Nilsson,J., STRIPS: A new approach to the application of Theorem proving to problem solving, Artificial Intelligence 2(1971) 189-208.

21. Harrison,M.C. and Rubin,N., Another generalization of resolution J. ACM 25(3) (1978) 341-351.

22. Henschen,L. and Wos,L., Unit reputation and Horn set J.ACM 21(4) (1974) 590-605.

23. Kling,R.E., A Paradigm for reasoning by analogy, Artificial Intelligence 2(1971) 147-178.

24. Kowalaski,R., The case for using equality axioms in automatic demonstration, Lecture notes in Math, 125, 163-190.

25. Kowalski,R and Kuehner,D., Linear Resolution with Selection function, Artificial Intelligence 2(1972) 227-260.

26. Kuehner,D., Some special purpose resolution system in Machine Intelligence vol.7, B.Beltzer and D.Michie (ed.) American Elsevier,New York (1972) 117-128.

27. Loveland,D.W., Automatic theorem proving: A logical basis North-Holland, Amsterdam (1977).

28. Morris, J.B., E-Resolution: Extension of resolution to include the equality relation, Proc. First,Intd. Joint-Conf. on A.I.(1969) 287-294.

29. Murray, NeilV., Completely Non clausal theorem proving, Artificial Intelligence 18 (1982) 67-85.

30. Nevins,A.J., A human oriented logic for Automatic Theorem proving, J.ACM 21(4) (1974) 606-621.

31. Nevins,A.J., Plan Geometry Theorem Proving using forward chaining, Artificial Intelligence 6(1975) 1-23.

32. Newell,A., Shaw,J.C. and Simon,H.A., Empirical explorations of the logic theory machine: Feigenbaum and Feldman (ed.) Computers and Thought, 134-152.

33. Nilsson,N.J., Problem solving methods in Artificial Intelligence, McGraw Hill (1971).

34. Nilsson,N.J., Artificial Intelligence (including a review of automatic theorem proving) IFIP, Stockholm, Sweden, (1974).

35. Overback,R.A., A class of Automated theorem proving Algorithm, J.ACM 21(2) (1976) 191-200.

36. Pastre,D., Automatic Theorem Proving in set theory Artificial Intelligence 10(1978) 1-27.

37. Peterson,G.E., Theorem proving with lemmas, J.ACM 23(4) (1976) 573-581.

38. Plaisted,D.A., Theorem proving in Abstraction, Artificial Intelligence 16 (1981) 47-108.

39. Prawitz.D., An improved proof procedure, theoria 26 (1960) 102-139.

40. Prawitz,D., Advances and Problems in mechanical Proof procedure in: Meltzer,B and Michie,D.(ed.), Machine Intelligence 4(American elsevier, New York (1969) 59-71.

41. Robinson,J.A., A machine-oriented logic on the resolution principle, J.ACM 12(1) (1965) 23-41.

42. Robinson,J.A., Automated deduction with hyperresolution International J. Ass. for computer.Math, 1(1965) 227-234.

43. Slagle,J.R., Artificial Intelligence: The heuristic Programming approach, McGraw Hill (1971).

44. Slagle,J.R., Automated theorem proving for theories with simplifiers, commutativity and associativity J.ACM 21(4) (1974) 622-642.

45. Slagle,J.R. and Norton,L., Experiments with an automatic theorem prover having partial order rules, Commun. ACM. 16(1973) 682-688.

46. Turing,A.M., Computing machinery and Intelligence Computer and Thought; edit by Esward A. FE IG-Enbaum and Julian Feldman, McGraw Hill (1963) 11-35.

47. Watanable,S., Paradigmatic symbol - A comparative study of Human and Artificial Intelligence, IEEE trans. on syst. Man and Cyber, Vol.4 Number 1 (1974) 100-103.

48. Watanabe,S., Some thought on Dreyfus, 'Critique' of Artificial Reason, IEEE trans. on Syst. Man & Cyber, Vol.5 Number 1 (1975) 141-145.

49. Weizenbaum,J. Computer power and Human Reason, W.H.Freeman and Co. (McGraw Hill) (1976) 204-205.

50. Wos,L and Robinson,G., Paramodulation and Theorem Proving in first order theories with equality, in machine Intelligence 4, B.Meltzer and D.Michie, ed. Edinburgh, U. Press and American Elsevier, New York(1969) 135-150.

51. Wos,L, Robinson,G and Carsen,D., Efficiency and completeness of the set of support strategy in theorem proving,J. ACM 12(4) (1965) 536-541.