This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
SEMANTIC TREES:

NEW FOUNDATIONS FOR AUTOMATIC

THEOREM-PROVING.

by

Patrick John Hayes

PhD Thesis
University of Edinburgh
CONTENTS

Acknowledgements

0. Introduction
  0.0 Summary ............................................. 1
  0.1 Some notation ....................................... 4

1. What is a theorem-proving system? ....................... 7
  1.0 Introduction ....................................... 7
  1.1 TP systems .......................................... 8
  1.2 Relative efficiency: comparing TP systems .......... 13
  1.3 I/O and control behaviour of a TP system .......... 19
  1.4 Input languages for TP systems ..................... 24
    1.4.1 Assertional vs. Imperative ..................... 26
    1.4.2 Some logics .................................... 33
      1.4.2.1 Propositional calculus ..................... 33
      1.4.2.2 First-order predicate logic ............... 35
      1.4.2.3 Higher-order logic ......................... 40
      1.4.2.4 PC with equality ........................... 43
  1.5 Expressions, proofs and structures .................. 47
    1.5.1 Levels of representation ....................... 48
    1.5.2 Structures .................................... 55
      1.5.2.1 Derivations of structures .................. 56
        1.5.2.1.1 Comparison with other formalisms .... 62
    1.5.3 Localisation and choice ....................... 64
1.5.4 Proofs and relevancy

1.6. The necessity of search

2. What use is a theorem-proving system?

2.0 Introduction: some views about AI

2.1 TP systems as devices for proving theorems

2.2 TP systems as problem-solving subroutines

2.2.1 Problem-solving systems

2.2.2 Representations of knowledge: a rival methodology

2.2.3 The role of TP systems

2.3 Computational vs. assertional knowledge

2.4 Logics as representations of knowledge

2.4.1 Semantics

2.4.2 Examples

2.5 Computation as deduction

3. Search spaces, refinements and lifting

3.0 Introduction

3.1 More on search spaces

3.1.1 To identify or not to identify

3.1.2 On keeping searches simple

3.2 Normal forms, refinements and complexity measures

3.3 Simple and structural redundancy

3.4 Pseudostructures, compulsory operations and admissibility tests

3.5 Propositional properties and the lifting lemma

3.5.1 Propositional relations and substitutions
ACKNOWLEDGEMENTS

I have become indebted to many people - too many to name them all - during the long years it has taken to write this dissertation.

My supervisor, Professor Bernard Meltzer has given constant encouragement, advice and friendship, which I value more highly year by year. I am also indebted to him for providing in the Department of Computational Logic at Edinburgh such a stimulating environment in which to work.

To all my colleagues in the Department, and the rest of the school of AI, I am grateful for many hours of discussion, collaboration and criticism; especially to Bob Kowalski, Gordon Plotkin and Bruce Anderson. The perceptive reader will recognise, in this dissertation, the influence these four, in their various ways, have had upon my thinking. Rod Burstall and Sylvia Weir have encouraged me to continue through my blacker hours.

To Professor John McCarthy I am grateful for many stimulating discussions, much perceptive
advice, and for inviting me to visit his laboratory at Stanford for two summers.

My students at Essex University, Jörg Siekmann and Dave Wilkins, have been a constant source of ideas and enthusiasm during the last year. I am also grateful to my colleagues Richard Bornat and Mike Brady for their patience.

Most of all, I wish to thank my wife, Jackie, for her unflagging support, her patience and for typing the entire manuscript.
0. INTRODUCTION

0.0. Summary

This dissertation is concerned with theorem-proving by computer. It does not contain a great number of new results, in the sense of new computational devices for improving the efficiency of theorem-proving programs. Rather it is intended as an account of a new approach to the fundamentals of the subject. It is a work, in the main, of consolidation and entrenchment rather than of extension.

Accordingly, rather a large fraction of the total is devoted to an examination - a re-examination in fact, since there have been others before me - of the ideas and presuppositions underlying theorem-proving, and an attempt to uncover the underlying reasons why certain ideas - notably that of search - have arisen so consistently in the history of the subject.

The first chapter is entirely devoted to this. In this chapter I occasionally make reference to
illustrative examples from the theorem-proving literature without detailed explanations of the terms used. I spend much space in chapter 1 on what are usually regarded as topics hardly worth discussing. I apologise to those readers who might be offended by such longwindedness, but the idea is to critically examine the foundations rather than presuppose them.

In the second chapter the relationship of theorem-proving to artificial intelligence is critically examined, and certain morals for the future development of the subject are drawn. This examination contains an essay on theories of artificial intelligence.

The third chapter of the dissertation is an extended account of the completeness theory, due in essence to T. Skolem, Dag Prawitz and J. A. Robinson, which underlies modern theorem-proving. In this chapter also the concept of redundancy is examined, and the lifting theory of J. A. Robinson is reconstructed with the axiomatic theory of structures which was introduced
In chapter 1.

In the final chapter it is shown how the concept of semantic tree can be used to provide a uniform account of numerous hitherto separate theorem-proving systems. This unified theory thus enables more realistic comparisons between disparate systems to be carried out, and there is some discussion along these lines.
0.1 Some notation

It is convenient to introduce some technical notation here as it is of general utility throughout the dissertation.

I will use equality (=) informally as an adjunct to ordinary English as well as occasionally in the formal language under discussion.

I will use curly brackets to denote sets, in the usual way: \( A \in \{A,B,C\} \). The empty set is written \( \emptyset \). I will use angle brackets to denote sequences: \( \langle A,A,B,C \rangle \). Sequences are finite structures which can contain duplicates and in which order is important, so that \( \langle A,A \rangle \neq \langle A \rangle \) and \( \langle A,B \rangle \neq \langle B,A \rangle \). I will speak of the nth object in the sequence. The empty sequence is written \( \langle \rangle \). In a convenient abuse of notation, I will use some set-theoretic notation upon sequences. In particular, if \( S \) is a sequence then \( A \in S \) will mean that \( A \) is the nth object in \( S \), for some \( n \), and if \( T \) is a set or a sequence then \( S \cap T = \{ x \in T : x \in S \} \).
Notice that an intersection is always a set. For example, \(<A,B,B,C> \cap <B,B,D> = \{B\}\).

I will write \(S = S_1 \cup S_2\), where \(S_1\) and \(S_2\) are sets, to mean that \(S = S_1 \cup S_2\) and \(S_1 \cap S_2 = \emptyset\).

If \(S\) is a sequence and \(T\) is a set then \(S - T\) is the sequence obtained by removing all elements of \(T\) from \(S\) and collapsing the result. For example \(<A,B,C,C,D> - \{A,C,F,G\} = <B,D>\).

Notice that \((S - T) \cap T = \emptyset\), and that \(S - T\) is a monotonic subsequence of \(S\).

We will sometimes need to speak of equivalence relations on a set. If \(R\) is an equivalence relation on \(S\) then \([x]_R\) denotes the equivalence class of \(x\) under \(R\). The suffix may be omitted when no confusion would result.

A partition \(\prec\) of a set \(S\) is a finite set of sets \(S_1, \ldots, S_n\) such that \(S = S_1 \cup \ldots \cup S_n\). I will write \(\prec(x) = S_1\) when \(x \in S_1\), and sometimes will identify \(\prec\) with the corresponding equivalence relation so that \([x]_\prec = \prec(x) = S_1\).

I will also regard any map \(\prec\) from \(S\) onto a
set T as a partition by identifying each \( t \in T \) with the pull-back \( \{ s \in S : \alpha(s) = t \} \). If \( \alpha \) is a partition from \( S \) onto \( T \) and \( \beta \) is a partition on \( T \) then \( \alpha \beta \) is the partition on \( S \) defined by

\[
\alpha \beta (s) = \beta (\alpha(s)).
\]

\( \alpha \beta \) will be called a coarsening of \( \alpha \): \( \alpha \) is a refinement of \( \alpha \beta \).

If \( S \) is a set and \( f \) some function defined on elements of \( S \) then \( f(S) \) (or \( Sf \)) will denote the set \( \{ f(s) : s \in S \} \) (or \( \{ sf : s \in S \} \)). In general, if we have any transformation upon objects denoted by some notational device (e.g. superscripts, over or under-lining, prefix or postfix additions, etc.) and 'S' denotes a set of the objects then applying the notational device to 'S' will result in a symbol denoting the set of entities denoted by the symbols obtained by applying the notational device to the names of the elements of \( S \).
1. WHAT IS A THEOREM-PROVING SYSTEM?

1.0 Introduction

Almost any account of an on-going research field will be rejected by some people as too narrow, too inclusive or downright wrong. In attempting to delineate the main features of theorem-proving systems, and even more examining the relationship between theorem-proving and artificial intelligence, I am running a risk of being universally condemned. In this first chapter, however, I have tried to be as neutral, on all of the many contentious issues which must be covered, as possible. This first chapter is intended to be purely descriptive of the current state of the art. (The normative opinions all come in the next chapter). Only the reader can judge for himself whether or not I have been successful.
1.1 TP systems

In this first section several basic concepts are introduced which are discussed more fully in later sections. Since it is all introductory, it has a somewhat dilletante air. This impression should be dispelled on reading the later sections.

A theorem-proving (TP) system is a device - usually a computer program - which inputs expressions in some formal language or other, and which outputs, if it outputs anything, proofs of the expressions. Another name for a TP system is proof procedure, emphasising this description.

There are many adjectives which have been applied to TP systems. If the system will always, in principle, output a proof when the expression has one, it is complete. If it will never output a proof when the expression doesn't have one, it is consistent. (Consistency is often taken for granted, but there is considerable discussion of completeness in the literature.) What the system does do when the expression has no proof is not defined: it may sometimes (or always) stop and report failure, or run on for ever, or exhaust some finite resource (time or space, usually), or something else may happen. Other adjectives sometimes applied include efficient, powerful, etc. Unfortunately these
seem to have little operational meaning. Various notions of relative efficiency can be defined, as will be discussed shortly.

If the expressions, whose proofs are to be generated, are the only input to the system then, following Minsky, I will call it uniform. If there are other inputs which are intended to be, in some sense, advice which aids the TP system in its task of finding proofs, then I will call the system heuristic. The extent to which the advice can in fact be so used in a heuristic TP system depends of course on the particular system. This important topic will be discussed later. We note in passing that the description so far would admit, as a heuristic TP system, the simple device which takes as inputs an expression and its proof (considered as advice) and outputs the proof. This TP system will be called the silly system.

Given a heuristic TP system S, we can construct related uniform systems in several ways.

(i) Let A be some piece of advice acceptable to the system. Then the uniform system got by fixing the advice to be A will be called a reduction of S:
(ii) In particular, if \( A \) is empty (this is not always acceptable: some heuristic systems are unable to function without any advice), the reduction will be called the **free reduction** of \( S \):

![Diagram](attachment:image.png)

(iii) Suppose some other device \( D \) is able to **generate** advice acceptable to \( S \), by examining the expression, or/and by communicating with \( S \). Then we can define a uniform system thus:

![Diagram](attachment:image.png)

This will be called an **autonomous extension** to \( S \).

No doubt there are other ways of constructing uniform systems also: and we could clearly subdivide autonomous extensions into several subclasses. But these three ideas are sufficient for the present discussion.

The ideas of completeness and consistency for a heuristic system are not yet well defined. The normal
usage seems to be, more or less, that a heuristic system is complete (consistent) just when every reduction of it is complete (consistent). That is, the advice must not affect completeness or consistency. This is the usage we will adopt here. Note that according to this definition the silly system is neither complete nor consistent.

We have not defined the nature of the advice accepted by a heuristic proof procedure. It varies considerably from system to system, in fact. I am aware of the following examples:

1. Set of support
2. Function nesting bounds
3. Clause length bounds
4. Selection function
5. \( A \)-ordering
6. Locking orderings
7. Models
8. Renamings
9. Search heuristic functions

And no doubt there are others. Certain of these are conventionally used largely in the contexts of reductions (e.g. 7) or autonomous extensions (e.g. 4, 8, 9). The only case known to me in which a system is provided with any guides as to the best way of using the advice it makes available, is the set of support (Wos & Robinson 1965), and this is rather vague and
not supported by any particular theory. It is significant that this is also the only case of advice (as far as I am aware) which has been used in a nontrivial way by another program, to which the TP system is interfaced (Kling 1971: his program adjusted the set of support in the light of "analogies" between the expression and one previously proved by the system).

All of the above is heavily dependent upon just what is meant by "expression" and "proof", and this in turn partially depends upon the formal language which the TP system is supposed to handle. In particular, the definitions of completeness and consistency depend upon the idea of an expression having a proof, and this is supposed to be defined by the language. We deliberately leave it open at this stage whether this is defined syntactically or semantically: the only point is that it is independent of the proof procedure. Thus a given language may have many proof procedures. Various languages and classes of language are discussed below. Our usage of 'proof' is such, however, that the relationship between expression and proof is not defined wholly by the choice of language, but also by the choice of representation for the TP system. This will also be taken up later.
1.2 Relative efficiency: Comparing TP systems.

There are a great number of TP systems in the literature. Not surprisingly, therefore, there is considerable interest in comparing one with another. The literature is filled with many explicit and implicit claims for the merits of various systems, making a comparative theory even more desirable. Now there are many ways in which we can compare and classify proof procedures.

One obvious feature is the formal language from which the input expressions are taken. TP systems for different languages are comparable only relative to some comparison of the languages. For example, there have been attempts to relate TP systems for first-order and for higher-order logic by axiomatising the 'same' assertion in the two languages, of course in different ways, and comparing the resulting behaviours of the two TP systems. Again, there have been many similar comparisons between systems for first-order logic and ones for first-order with equality. Here the comparison between the two languages is very obvious.

Even if we assume that the language is fixed, however, it is still not clear exactly what differences in behaviour might lead us to prefer one TP system to another, in general. It clearly depends upon what use
we wish to make of the system. Possible uses for the TP systems are the subject of a long discussion later.

One case which does seem fairly clear is that in which we have two complete, consistent, uniform TP systems for the same language. Then we can say that one is more efficient than the other if, for any input, it terminates with a proof before the other does. This simple criterion has been widely used in the literature to compare TP systems, in the form of experimental results. Half a dozen or so example expressions are presented, on which system A is found to perform better than system B: by induction one is to assume that the same will occur for any expression, i.e. that A is more efficient than B.

Objections to the validity of this simple criterion are often based on the fact that extraneous accidental circumstances, e.g. more efficient computer languages, the use of American rather than British hardware, etc., may be causing the superiority of one system. Thus, measures other than simple running time, of the computational effort expended by a TP system before termination have been suggested. For certain of these measures, it is sometimes possible (under several rather restrictive assumptions about the nature of the TP systems) to actually prove that one system is more efficient than another. The theory which enables such proofs to be obtained is almost entirely due to R. Kowalski (1970, Hayes & Kowalski 1971), and it is a very considerable
step forward. However, the assumptions necessary to make it work are quite strong, so it applies, as yet, to only a rather narrow class of uniform TP systems. Moreover, the measures of computational effort considered are rather highly idealised. (In terms of section 1.5 below, they are defined only at the structural level.)

It sometimes occurs, even in this straightforward case, that neither of two procedures is more efficient than the other, but that one is superior only on a certain subclass of expressions. In this case, if the class in question can be neatly defined and is of interest, it may be worth defining a notion of relative increased efficiency. One example of this is the well known case of Horn clauses: this subclass of predicate calculus expressions is one on which several uniform TP systems perform well, which are quite poor on other expressions. Another example is provided by TP systems which are designed to handle decidable subcases of predicate calculus.

All of this however only applies to uniform TP systems. To compare the efficiency of two heuristic TP systems is quite a different matter. Clearly, simply comparing performance - however measured - on the same input expressions, is not adequate. For, what of the advice given to the two systems? It is hardly fair to count one the better system because it
performs better, if it is getting better advice than the other. (The silly system would always come out on top, for example, given such a basis for comparison.) The concept of "fairness" involved here will be elaborated later. (Before going further, I invite the reader to consider whether or not he seems to find it intuitively appealing. The answer will probably have some bearing upon his attitude to the uses of TP systems, discussed later.)

But on the other hand we cannot insist, usually, that two systems get the same advice; since, as was remarked earlier, different heuristic systems take as input quite different kinds of advice, the relative usefulness of which are not usually easily measurable. The following seem to be the only ways in which the 'efficiencies' of two heuristic TP systems might be compared.

(i) If in fact we do have some way of comparing the advice acceptable to system $A$ with the advice acceptable to system $B$, then we could define $A$ to be more efficient than $B$ if, for any expression and any advice, $A$ produces a proof quicker (or with less effort) than $B$ does with the same expression and comparable advice.

(ii) Suppose that every reduction of $A$ is more efficient than any reduction of $B$: then $A$ is
absolutely more efficient than B.

(iii) Suppose that for every expression, and every piece of advice acceptable to B, there exists a piece of advice acceptable to A, such that A performs better, with that advice, than B does with its advice on the same expression. Then we will say that A was conditionally more efficient than B.

Of these, (i) rarely happens, in practice. There are interesting cases of (ii) (e.g., SL-resolution vs 'minimal' resolution, with appropriate caveats about search strategy), and also of (iii). Notice that the silly system is conditionally more efficient than any system, strictly speaking. If this is considered unacceptable, then it (and cases like it) can be ruled out by the requirement that A and B both be consistent and/or complete in order for a respectable comparison to be made between them.

This suggests a different point of view towards comparing heuristic TP systems, however. Perhaps one should regard it as an advantage of a TP system, that it is able to accept, and make good use of, powerful advice. For example, if the TP system is to be interfaced with another system (human or machine) which is capable, upon occasion, of making such advice available, why not have the system make use of it? To take an extreme example: if one proposed to use a TP
system to generate proofs of expressions for which one already, in fact, had proofs to hand: then one could hardly do better than use the silly system. What is silly about this example, according to this view, is not the TP system, but the idea that one would ever consider using it. The definition of conditional efficiency reflects this attitude, to some extent, in giving credit to system B's ability to accept good advice.

These issues will receive a fuller ventilation later, when uses of TP systems are discussed. For the time being we will merely sum up by emphasising that comparisons between heuristic TP systems are more problematic than between uniform ones, and seem to depend upon the use one envisions making of the advice input.
I/O and control behaviour of a TP system

As defined, a TP system is a device which has the overall characteristics, in programming terms, of a subroutine. That is, it accepts inputs: runs to termination (or fails to terminate): gives an output: stops. Its behaviour has no other describable aspects. It does not, for example, cooperate with some other process, like a coroutine: it does not pass or receive control in any other way than as defined by the initial call and the final termination. It does not generate partial results on the way, or give any other outputs than the proof upon termination. If interrupted (by some executive routine, for example), it cannot then accept further inputs: all that such an executive could do with it would be to let it carry on.

Almost all systems described in the literature, which would be generally regarded as theorem-proving devices, share these characteristics. (Several systems output "system" information, e.g. tracings of their progress, or statistics on their behaviour: I am disregarding such outputs for the purposes of this discussion.) The exceptions I know of are the following.

(1) Several heuristic systems are capable of accepting advice on a more or less continuous basis, and making use of altered advice in new
ways as time progresses. For example: SL-resolution's selection function may be redefined at any time; several systems can accept dynamically varying search heuristics, etc. It is often proposed that autonomous extensions of such systems be constructed to take advantage of this ability: the autonomous extension is then a simple subroutine, of course.

(ii) The interactive theorem-proving device called SAM (Bennet, Easton, Guard 1967) has quite a different character. Rather than generating a proof and terminating, SAM was designed to generate consequences of the input expression(s) as a continuous process. A human was able to watch this process via fairly sophisticated printing and display software and a CRT. He could interrupt SAM whenever he felt that something interesting was happening, give it fairly elaborate advice so as to adjust its behaviour, and then restart it.

In this case the system is better defined as a process than as a subroutine. SAM is clearly not a TP system as defined here. For example, the above notions of efficiency simply do not apply to SAM. I think that the exclusion of such interactive systems would be considered acceptable in the normal usage of "TP system".
In passing we note that the 'interactive' TP system described by Luckham & Allen (1970) is quite different from SAM, and in fact falls quite neatly under our description: it is a heuristic TP system which is used in a rather more highly interactive way than usual. All this amounts to, is that these authors have implicitly rejected some of the above definitions of efficiency.

The restricted nature of the I/O behaviour of a TP system means that its role, in any larger system of which it is a component, is similarly restricted. For many purposes this may be irrelevant: for others it may be a severe restriction. We will discuss this at more length later. For now, we merely note the fact that TP systems have the character of a subroutine, in most cases.

Lest this be thought a nonsequitur, we list some other things it could be:

- a process (cf. SAM);
- a coroutine, sharing control with another device on a reciprocal basis;
- an executive, acting as the coordinator of a number of other devices which it treats as subroutines;

(This is to be sharply distinguished from the idea of the TP system being written in a language by using an executive routine to control subroutines. In this latter
case the TP system would consist of an executive +
subroutines, not act as an executive. The distinction
is like that between the source language of a compiler
and the language the compiler is implemented in. Many
TP systems are in fact implemented in this sort of way:
it is a natural way of writing a program with fairly
complicated parts whose overall organisation is simple
and repetitive, like most TP systems.)

- a transducer, requesting input and delivering
processed output as a continuous activity.

In all of these cases, a plausible case can be made
for wanting a TP system to have that sort of organi-
ation in certain contexts.

For example:

- it might be desired to have several TP systems
(perhaps using different methods) cooperate
on proving a difficult theorem. Coroutining
would be a natural way to organise such cooperation.

- A TP system being used as the main deductive and
plan-making organ of an integrated robot would
best act as the coordinating executive for all
the lower-level activities, since it alone
has direct access to the most abstract world-
model.
A TP system used in an on-line question-answering application, e.g. a management information system or a computer-aided instruction project, would best be regarded as a transducer operating with assertions and requests. (In such an environment, also, the interrupt behaviour of the TP system becomes important, since it must maintain communication with the outer world in real time.)
1.4. Input languages for TP systems

In this section are briefly discussed several topics concerned with the expression input language. Whether or not the possession by this language of a given property is thought to be a recommendation of it, depends of course upon the use for which the TP system is intended. For example, the expressive power of the language - its ability to phrase useful or appropriate descriptions of some domain - will be thought important if it is desired to use the TP system as some kind of problem solver for that domain. On the other hand, for certain uses of a TP system (e.g. computing minimal forms of Boolean polynomials), expressive power may be almost totally irrelevant. Expressive power will in fact be important to the later discussion, and so we discuss it here. In particular, we classify languages by their ontological commitment (in Quine's phrase). This is probably the most important aspect of the expressive power of a language, since it can express only those concepts which concern entities that it can describe.

We do not discuss the other input to a heuristic TP system, the advice. In almost all cases known to me, the range of possible advice inputs is comparatively unstructured. In some cases it is some small finite
set (e.g. set of support: if there are $n$ clauses then there are less than $2^{n-1}$ possible sets of support); in others it is a large infinite set (e.g. search heuristics can often be any computable function from, say, resolution derivations to positive integers). But it never comprises a language: there are never such notions as "subadvice", or grammars of advice. Questions such as the expressive or descriptive power of the advice, or what it means (i.e. a semantic theory), are never discussed. The only exceptions are systems which would not conventionally be regarded as TP systems, viz. interpreters for certain programming languages. Again, whether one regards it as a criticism of heuristic TP systems, that their advice input is so much less structured (& hence less expressive) than their expression input, depends upon the intended use of such systems.
1.4.1 Assertional vs. Imperative

In all systems in the literature which are called TP systems, the input language is a formal logical calculus. In most cases it is some formulation of the first-order predicate calculus, perhaps with equality. But at any rate, an assertional language: one which is used to make statements.

Such languages are often contrasted with imperative languages, or procedural languages, which are used to give commands or specify procedures to be carried out. Assertional languages describe, while imperative languages control, their respective domains of interpretation. This distinction is almost universally recognised as one which is quite clear-cut and very important. Theorists both of artificial intelligence and of programming language design (computational linguistics, in John Laski's terminology) have emphasised such a distinction.

Now according to all the definitions so far, an interpreter for a programming language, such as LISP, could be regarded as a TP system. We have only to define what we mean by a "proof" of a - say - a LISP S-expression. We could, for example, define the proof of an S-expression to be its LISP value:
then the LISP interpreter is a TP system, which is complete, consistent and uniform, by definition. But such a usage would be widely regarded as abnormal and confusing. It is felt that there is a profound difference - qualitative rather than quantitative - between theorem-provers which search for a proof, and interpreters which perform computations. And this distinction is often based upon the observation that the input languages are respectively assertional and procedural (Hewitt 1969, etc.).

I feel this view is entirely mistaken. I wish to emphasise the similarities between theorem-provers and interpreters: in fact, they are formally indistinguishable. The real difference between them is, I suggest, not intrinsic to the devices themselves, but rather is a difference of methodology on the part of the people who design and use them. What makes a TP system into a "theorem-prover" on the one hand, or an "interpreter" on the other, is the intended use of the system. In the first case, it is supposed (usually), to operate more or less autonomously and display intelligence, or at least skill: in the second case, it is supposed (usually) to operate in cooperation with other devices so that the whole system of which it is a part displays intelligence or skill. This will be discussed more fully later; for now, let us return to the nature of the input language.
If, as I maintain, interpreters and theorem-provers are not so different, what of the difference between assertional and imperative languages? Now, although this distinction seems intuitively clear and is widely celebrated, closer investigation reveals certain problems.

For one thing, different workers seem to mean rather different things. To take the most extreme example: LISP is regarded as the sine qua non of procedural languages by the Minsky-Hewitt-Winograd school of AI theory: but it is regarded as the most extreme case of assertional programming language (as contrasted with, say, FORTRAN) by computational linguists such as Landin and Strachey. Thus LISP is often regarded as an applied lambda calculus; and it was considered a milestone in the development of computational theory when Dana Scott provided a semantic domain for the lambda calculus. In lambda expressions denote objects (functions) in the domain in exactly the same way that first-order expressions denote objects in first-order interpretations. There is a truth-recursion, defining the denotata of complex expressions in terms of their simpler subexpressions, in precisely the Frege-Tarski tradition. While it is true that lambda-calculus expressions do not denote truth-values, in general, this is a minor difference.
Applied lambda calculi - such as LISP - do have expressions which make assertions, in any case. Moreover, from the earliest times, LISP has been defined axiomatically.

(There is a real difference, one must admit, between the 'mathematical' semantics of Scott, and certain other semantical theories which have been suggested for programming languages, notably the 'abstract-interpreter' semantics of Landin's SECD machine and the Vienna school of theoreticians, and Bahrendrecht's semantics for the lambda calculus, in which expressions denote those transformations between expressions which they produce when lambda-conversion rules are applied. It may be that a watertight distinction between 'assertional' and 'imperative' could be based on a difference between classes of semantic theories. However, it is unlikely in the extreme that this distinction would coincide with the Minsky-Hewitt distinction, since these deviant semantical theories depend crucially upon special properties of the languages in question, notably the Church-Rosser property. Without this property, there would be no preferred notion of "abstract interpreter", since different deductive strategies would yield different denotata. Thus, they are most unlikely to be sufficiently general to encompass those languages - such as PLANNER and CONNIVER - considered 'procedural'.
by the Minsky-Hewitt school.)

For another thing, why does the assertional/imperative contrast seem so intuitive? The usual answer is, because in natural language the two classes of expression are clearly demarcated and have different conversational uses. But is the distinction really so clear? A statement can serve as a command: "I want you to stop making that noise!" , and an imperative can make a statement: "Look in the oven if you want your dinner."

There are, too, very close links between commands and predictions: "Go out!" and "You will (now) go out." seem almost entirely equivalent. This latter is especially interesting as it seems to correspond to the difference between specifying an action to be carried out - a procedural expression - and describing the process of its being carried out - an assertion.

The effects of these upon an agent are equivalent; provided that he is disposed to obey the first, or believe the second. We could subsume commands under predictions if we reduced obedience to belief by assuming that a command contained an implicit claim to obedience. Thus the command "You do X!" (cf. Rescher 1966) would be analysed, according to this view, something along the lines "You will do X; and you believe what I tell you because I am in authority over you". As evidence for this view, note that in
situations where the second conjunct is beyond question, it is often omitted and the commands conveyed simply by the prediction: "Platoon will come to order!" or, a mother to a child: "You will eat your dinner all up today!". The reason why English (for example) has such a sharp syntactic distinction between assertive and imperative expressions is probably because the question, of who is in authority over whom, is one which is of basic importance to the social fabric. Language having been evolved in a social context, specially compact devices have been developed for conveying this important information rapidly and effectively. Analogous special devices for handling tense information have also evolved, for example; this being another aspect of the world (not of language!) which is of importance in communication. But these historical relics present in English need not concern us when discussing languages for communication with computers. There is no need to claim obedience from a computer: it doesn't understand the concept, in any case!

There is not space here to develop this idea any further: but I hope to have cast some doubt upon the reader's confidence in the assertional/imperative contrast.
Having said all this, I will now temporarily revert to my attempt to be descriptive, and accept the common usage. For the time being, then, let us accept that the input language to a TP system is one which would conventionally be regarded as a logical calculus containing assertions.

In any such logical language, the symbols used to write expressions are divided into two disjoint classes, the logical and the non-logical symbols. The former are those which have a special meaning assigned them by the language, or which are otherwise involved in the grammatical definition of expressions in the language (e.g. brackets, commas etc.). The latter are those which denote the entities which the language has in its ontology. This distinction is vital to TP systems, since they must be able to recognise logical symbols and treat them in special ways. An exactly similar distinction holds for compilers and interpreters for "imperative" languages: these must be able to recognise symbols or subexpressions which have special meanings defined by the language in question. For example, reserved words are treated quite differently from identifiers by an ALGOL compiler.
1.4.2 Some logics and their ontologies

1.4.2.1 Propositional calculus

This is a language with virtually no ontology. It commits us only to the idea of assertions and truthvalues. Its expressive power is correspondingly limited. One can say virtually nothing about most domains using only propositional logic, since it provides no way of naming anything other than the facts; and hence, in particular, no way of saying what the facts might be about.

The one basic supposition of propositional logic is:

1) whether a given proposition is or is not the case, is always a meaningful question with a single determinate answer.

In spite of this paucity of expressiveness, propositional calculus provides a basis for much of the theory of TP systems. Its importance stems from two properties. Firstly, it is decidable and hence its metatheory can make use of inductive arguments. Secondly, provability in first-order predicate calculus is closely related to provability in propositional calculus via the Herbrand/Skolem and
Prawitz/Robinson theories. There are uniform methods for constructing proof procedures for predicate logic out of one for propositional logic, as will be described later.

A significant property of propositional calculus is the existence in it of normal forms. There are several highly restricted syntactically defined classes of expressions which are universal in the sense that for any expression, one can effectively find one in the class which is equivalent to it (has the same truthvalue in all interpretations: hence expresses the same assertion). This means that a TP system which handles only this restricted class can be regarded as accepting assertions from the whole of propositional calculus, by providing it with a front end which transforms expressions to normal form:

```
expressions -> normal expressions -> proofs
```

The design of such preprocessors is so straightforward that this construction is often taken for granted. However, it is not clear that one gets a particularly efficient or useful TP system this way. Indeed, several criticisms have been levelled at this way of proceeding, of which probably the most cogent is
the observation that in a heuristic TP system, advice appropriate to the original expression may be quite inappropriate to the normal expression (Norton 1966). Again, Minsky has noted that an expression with a particularly simple form, and hence a simple proof, may transform into quite a complex normal form which has only a rather long proof. This latter point is answered, in the usual TP context, by the fact that the normal form usually chosen (conjunctive) is such that for many common expressions one wants to input (those consisting of a conjunction of implications between literals), this just doesn't happen: these expressions are already close to this normal form. (Note however that this answer assumes a particular use for the TP system.) Conjunctive normal form assumes a greater importance when we consider first-order predicate calculus.

### 1.4.2.2 First-order predicate logic (PC)

This has been the most commonly used input language for TP systems. The reason for this choice is probably its combination of expressive power and simplicity. Indeed, first-order logic in one form or another, has been the basic language of formal logic since Frege.
The ontological presuppositions of PC are, briefly:

(2) there are objects which can be distinguished one from the other and have each a separate and identifiable existence;

(3) between these objects hold relationships, which can be identified.

Moreover, that a named relationship holds between a collection of named objects constitutes a proposition in the sense of propositional calculus.

This ontology is extremely basic: indeed it seems to embody the very idea of ontology, without imposing any restrictions upon the entities which we might want to consider. This flexibility is what gives PC its central role in formal axiomatics.

Some philosophers go so far as to regard formalisation within PC as being a rigorous test of a theory's ontological commitment, indeed. (W.V. Quine is probably the most outstanding example: he makes the test explicit in his famous maxim "To be, is to be the value of a bound variable." . A similar attitude is found in the writings of, for example, A. Tarski and D. Davidson. Notice, however, that natural language contains some assertions which have ontological presuppositions incompatible with (2) above. For
example, assertions involving mass terms: "Snow is white". "Snow" here clearly denotes an entity, but not an object. Rather it denotes a substance which occupies space. Substances can perhaps be got into PC ontology by talking of pieces of substance as objects: this heap of snow, that heap of snow. But these are rather odd objects, at best: for one thing, they can combine and divide at ease, which is rather against the spirit of individuation. Predicating relationships of these sort of objects as though there were no problems involved leads one quite rapidly into some ancient difficulties such as the paradox of the heap. (If you add one stone to a small heap, it's still a small heap. A single stone constitutes a small heap. Hence, by induction, all heaps are small.) Thus, although PC's ontological commitment is very flexible, it isn't universal. Quine is rather cavalier on this issue: he simply dismisses all non-PC ontologies as being unscientific !)

PC makes no assumptions about the nature, properties or interrelations between the objects it talks of. If any such are required, they must be described by PC assertions explicitly. However, it is not always possible to specify the properties one wants in this way. Even without wishing to enrich the ontology of PC, therefore, one may wish to move to a more expressive
language which enables more things to be said about the objects. In particular, whenever one wishes to make an assertion which holds when one of a wide class of expressions is substituted for a given subexpression of the assertion, (as in higher-order logic, or in logics of causality (Hayes 1970), or in calculi of change such as STRIPS (Fikes & Nilsson 1971), (Hayes 1971)), one has to move outside of PC: but in none of these cases is there any essential change to PC's ontology. They are all cases where restrictions need to be placed upon objects which cannot, for one reason or another, be expressed in PC.

The PC language itself is a rather simple extension of propositional calculus, in which atomic propositions acquire some structure: in particular, they contain variables (denoting objects) for which can be substituted names of objects. The rules of inference are basically those of propositional calculus, together with the entailment \( \forall x \varphi(x) \vdash \varphi(a) \), where \( a \) is any name and \( \varphi \) any expression. Since the variable \( x \) occurs only inside propositions, however, the effect of this rule is localised to changes within atomic propositions. This means that such instantiations commute with propositional rearrangements within the scope of the quantifier. Thus if \( \psi \) is some expression which is a purely propositional consequence of \( \varphi \), then
the two sequences

\[ \forall x \phi(x) \vdash \phi(a) \vdash \psi(a) \]

and

\[ \forall x \phi(x) \vdash \forall x \psi(x) \vdash \psi(a) \]

yield identical conclusions. This fact underlies both Herbrand's theorem and the Prawitz/Robinson lifting lemma.

The normal-form results for propositional calculus extend easily to similar results in PC. The normal-form usually used is conjunctive. The end result of the familiar construction is a conjunction of universally quantified disjunctions of literals. The existential quantifiers are eliminated in favour of function symbols. (This neat trick automatically takes care of the inference rule of existential instantiation.) The transformation to conjunctive form is no longer an equivalence; but preserves unsatisfiability. This is handy, since we are often interested in expressions of the general form

\[ (\&_1 A_i) \supset B, \text{ where the } A_i \text{ are disjunctions.} \]

If such an expression is negated, it is close to conjunctive form: and we can then test it for unsatisfiability rather than validity. This is the usual (but not universal) approach to TP systems for PC. In keeping with this, our usage of "proof" will henceforth be extended to include demonstrations of unsatisfiability as well as of validity. This has no important effect
upon anything, so will be tacitly ignored most of the time.

The same criticisms levelled against the use of normal forms in TP systems for propositional logic apply here, probably with rather more force since the problem is so much more difficult. The reply about the appropriateness of conjunctive form is less convincing, since the transformation is now rather more dramatic. However, the use of conjunctive normal form is now motivated more strongly. Notice that the scope of the quantifiers has been restricted to single conjuncts of the whole expression. This means the non-propositional relationships which have to be investigated by the TP system have been greatly localised and simplified. This is an important property of conjunctive normal form.

1.4.2.3 Higher-order logic

As usually phrased, higher-order logic has a greater ontological commitment than PC. It presupposes in addition that:

(4) the relationships are themselves objects which can take part in relationships.

Syntactically: relation symbols can be quantified variables.
Now, one can straightforwardly map this syntax into $\text{PC}$ by the use of named application relations. To look at it semantically: so what if some of the objects are interpreted as relationships? That's just a restriction upon the PC ontology. So why is this any more than some theory formulated in PC? The answer, of course, is that one cannot state in PC the required restriction.

The reason why not is instructive. It follows from (4) that any relationship is an object, hence that any relationship expressed by a PC expression is an object. Thus we need, in order to begin to capture the force of (4), to state, for every relationship-defining expression of PC, that an object exists which is that relationship. Thus there has to be some expression which names every relationship-defining expression of PC: and no such PC expression exists. The required assertion is the axiom of comprehension. One can regard it as an axiom scheme added to PC.

A rather different approach to higher-order logic is to abandon PC notation altogether and use a syntax which enables names of relationships to be constructed at will by rules of inference. This is the typed lambda-calculus approach. Not surprisingly, the only
significant progress in constructing uniform TP systems for higher-order logic has used this syntax (Pietrykowski 1971, Huet 1973). Moreover, LISP (which can be regarded as a heuristic TP system for a higher-order logic), and many related programming languages, use a similar notation.

Higher-order logic shares with PC essentially the same relationship to the propositional calculus: the difference lies in the richness of the rule of instantiation. The key 'commuting' property still holds, with a slight modification. Normal forms with their localisation of quantification still exist. As one would therefore expect, there are important similarities between the theories of uniform TP systems for higher-order logic and for PC.

The motivation for using higher-order rather than first-order logic is sometimes said to be that many theorems have shorter proofs in higher-order logic. That is, one can so axiomatise many assertions in higher-order logic so that their proofs are shorter than would be the proof for any formulation of the same problem in first-order logic. As Kowalski (1970) has pointed out, this is not conclusive, since the difficulty of finding a proof depends upon many other factors than its length. It is also claimed that
higher-order logic is a more natural language to express many problems in. This is only an important factor for certain uses of TP systems, of course.

We note in passing that unless a language has some device which does the job of the comprehension schema, it isn't a formulation of higher-order logic, no matter how much its syntax may suggest otherwise. This point has been sometimes misunderstood in the literature: for example, Hewitt claims, quite erroneously, that PLANNER is basically a higher-order logic (Hewitt 1969).

Higher-order logic will not be discussed further in this dissertation. This is not because I do not think it important, but it is too early in the theory to tell whether it can be brought within the framework described here, or whether the present approach will have to be abandoned.

1.4.2.4. PC with equality

This seems at first sight like a very minor extension to PC: the ontology is the same, and the expressive power seems little greater since it is well known that one can easily define equality within a PC theory. However, appearances are misleading.
For a start, the expressive power is increased in significant and, to me at any rate, surprising ways. And the idea that equality is definable in PC is really false: it takes an axiom schema to define it (Leibniz' Law: \( (x=y \ & \ \phi(x) ) \supset \phi(y) \) for any expression \( \phi \)). It is just that (unlike comprehension) this schema can be reduced to a finite number of instances for a given vocabulary.

More importantly for the design of a TP system, the equality inferences sanctioned by the schema do not commute with instantiation. By instantiation a complex name can be introduced: a part of this can be altered by an equality inference. No equality substitution followed by instantiation could have produced the same effect, since the complex name simply wasn't there before instantiation. Thus the lifting lemma fails in PC with equality.

Moreover, there is no normal form for equality, in the sense in which there is for quantification. The scope of equality inferences cannot be localised in any way.

Unfortunately, most of the results on uniform TP systems for PC with equality have this negative character. This is especially regrettable since
such a useful expressive device in many applications. Some have drawn the conclusion that equality should simply be abandoned and other devices used instead. (for example, a reducibility relation which lacks symmetry & hence has a preferred direction of substitution; or syntactic identity rather than semantic equality). While some of these ideas have their merits, I would prefer not to abandon equality, in view of its deep roots in linguistics (cf. Quine's "Word and Object", for example), but rather move to heuristic TP systems equipped with advice to control equality inferences.

In many cases it is, anyway, demonstrable that the intended relation is equality even if not all of the properties of the equality relation are being used in a particular application. Take for example arithmetic expressions: suppose we wish (following Kowalski, personal communication) to have a reducibility relation in a PC axiomatisation corresponding to the computational reduction of an arithmetic expression to a numeral, say VAL. A true assertion of the theory might be VAL(PLUS(3,TIMES(4,8)),35). Now, clearly VAL(ϕ, ̂n) entails ϕ = ̂n (where ̂n is the numeral of n : e.g. ̂35 = "35") : this is just a statement of correctness of the computational scheme. Moreover, for any integer n, VAL( ̂n, ̂n) is true.
Hence VAL denotes a relation, in any model on the integers, which both includes and is included in the identity relation: i.e. VAL is equality. The only escape from this argument is to declare that in the intended interpretation, the individuals are not the integers but rather the expressions themselves. But now VAL (φ, ˇn) does not entail φ = ˇn. There is no way of stating the correctness or otherwise of such computations without using equality.
1.5 Expressions, proofs and structures

So far, TP systems have been treated as black boxes. In this section I develop some terminology for describing the internal workings of TP systems, especially those whose expression input language is PC. In particular, I will distinguish various levels of representation within TP systems.

The job of a TP system is to construct a proof. It is reasonable to suppose, therefore, that its activities will consist largely of the performance of constructive operations upon partial proofs. Exactly what this looks like depends of course upon what constitutes a proof and a part of a proof: and this depends critically upon the choice of expression input language.

We note in passing that in a heuristic TP system one would naturally expect that the form of the advice input would depend in turn upon the form of partial proofs, since (one might expect) the advice would be, in part, about partial proofs. However, in none of the cases listed earlier is this in fact so.
1.5.1 Levels of representation

In logic it is conventional to define a proof to be a more or less structured collection of expressions. Sometimes it is merely a sequence (in Hilbert-type axiomatisations); sometimes a tree labelled with expressions (in Natural-Deduction formalisations), sometimes a tree labelled with pairs of sequences of expressions (Gentzen systems), etc. But in every case the basic building blocks are expressions, and proofs are constructed out of expressions and other proofs by rules of inference. There is some notion of order in the structure, and expressions must be valid consequences of certain other expressions earlier in the ordering, in order for the proof to be wellformed: which expressions exactly, is defined in terms of the structural relationships, if it matters. (For example, in Hilbert-type systems it doesn't matter, and the ordering is all the structure there is.)

Any TP system has to be based upon some notion of what shape proofs and partial proofs have. And it has to have some way of representing expressions, proofs and their parts inside itself. I wish now to distinguish three levels of representation. First, at the highest level (this choice of "high-low" terminology is meant to be analogous with the same
usage in discussing programming languages) there are the expressions themselves, considered as linguistic objects. For example, we might be interested in disjunctions of proposition letters.

The actual operations of a TP system will not, however, be described at this linguistic level. Rather we will say that there are structures, some of which may represent expressions. For example, these structures might be sets of proposition letters (representing disjunctions), or trees labelled with atoms, or jagged arrays of literals, etc. A given structure may have several corresponding expressions. This will be called the structural level. I will not attempt a precise general definition of "structure". This seems to be a difficult and rather unrewarding exercise in general algebra. We could for example define a structure on a set $S$ to be: either a subset $S_0$ together with a set of relations on $S_0$ - in fact, an $\mathcal{L}$-algebra on $S_0$; or, recursively, an $\mathcal{L}$-algebra on some $\text{structures}$ on $S$. This would also make precise the later ideas of localisation. The rest of the theory could be defined algebraically on this as a basis. The lifting lemma, for example, would be phrased as the existence of a certain kind of homomorphism between structures. (One which was closely related to the pull-back of the instantiation
mapping defined by substitution application.) I have not taken this course: rather I will proceed semi-axiomatically, by making progressively more restrictive assumptions about the properties structures are allowed to have. All the important general results - in particular, the Ehrenfeucht-Rabin theorem and the Prawitz-Robinson lifting lemma - can be obtained within this framework.

Also important at this level are operations upon structures. These correspond to inference rules at the linguistic level, and are used by the TP system to construct structures which are partial proofs. Notice that proofs, partial proofs, etc., are structures. (Not: "are represented as structures", notice.)

Finally at the lowest, or data, level are the actual realisations of the structures in terms of data structures and programs in the physical computer underlying the TP system. A tree might be realised by a list structure involving upward pointers, for example: an array might be realised in contiguous storage or in list structure.

These distinctions are often implicit in the literature on theorem-proving, but they are rarely
described explicitly. For example, a paper describing a new theorem-proving procedure will often be phrased entirely at the structural level: it will, perhaps, talk of sets of literals, or trees with atoms at the nodes, etc. The proof procedures described in it will be defined in terms of these structured objects. Nevertheless, we are not surprised to read that the language being considered is, say, first-order logic: the structures represent first-order statements (in a way which is familiar, perhaps, from other papers): that goes without saying. Moreover we are not surprised if the paper gives no details of exactly how these structures - which may be quite complex - are represented in a computer program. That question is considered to be more or less independent of the definition of proof procedure. To put it another way: we could completely change the data representation of the structured objects and still regard ourselves as having a program which performed the same proof procedure.

The situation is exactly analogous to that obtaining in descriptions of high-level programming languages.

TP systems may be - and usually are - defined entirely at the structural level. However, any
discussion of 

**efficiency** should make clear whether
or not the structures involved can be efficiently
implemented at the data level. In most cases, to
be fair, this is fairly obvious: but when the
structures involved become more complicated, it can
become less so. It is dangerous, of course, to
assume in any particular case that no efficient
representation is possible, and use this as a criticism
of a TP system; since some more ingenious programmer
may come up with a better data representation than one
can think of oneself. (I mention this delicate point
because I myself have been thus refuted by Bob Boyer
& J. Moore.). Nevertheless it behoves one who claims
efficiency, to come up with an efficient data
representation.

Why is the structural level needed? Why not
define the operations of a TP system directly at the
linguistic level, in terms of operations upon
expressions? Now it is indeed possible to proceed
in this way. One simply has to regard (some)
structures as expressions in an (often rather unusual)
syntactic reformulation of the expression input
language. Rules which define the construction of
new structures from old are then called rules of
inference, as in familiar logical systems. This
approach works well when the structures involved are fairly simple, in particular when they are clauses. It is not difficult to regard a clause as an assertion. But the only kind of more complex object which can naturally be recognised in this scheme is the derivation, which is usually a tree labelled with expressions; and even in the context of clausal inference rules, this is really rather a restrictive vocabulary for describing TP systems.

It is important to notice that the nonlogical vocabulary of the language will be explicitly present in the structures. For example, truth-tables for propositional calculus contain proposition letters explicitly. This is because the relationships which are represented by the structure are defined only on the logical vocabulary, since only the symbols in this vocabulary are given a fixed meaning by the language. Thus it will often be that the structures contain no logical symbols; all the logical relationships which hold in proofs and partial proofs will, in such cases, be absorbed into the structural relationships within and between structures and their parts. However, nonlogical symbols cannot be so absorbed, and must therefore appear in the structures explicitly. If $S$ is a structure, $|S|$ will be defined to be the set of nonlogical symbols occurring in $S$. If $\mathcal{S}$ is a set of structures, then $|\mathcal{S}| = \bigcup_{S \in \mathcal{S}} |S|$. 
To sum up: we will regard the activity of a TP system as consisting of operations upon structures, some of which are susceptible of interpretation as expressions of the formal language under consideration. There are initial structures which correspond to the input expression, and terminal structures which are regarded as proofs, and possibly other structures as well. In part, the activities of the TP system will consist in the performance of constructive operations upon structures, which result in the creation of new structures. (For example, a TP system for propositional calculus might build up truth tables for its input expression in the form of trees. Then these labelled trees would be a complete truth-table, every line of which validated the expression.)

We assume that all other activities of a TP system are effective terminating computations. Thus the only way a TP system can fail to terminate is by generating infinitely many structures.
1.5.2 Structures

As remarked above, I will not attempt to give a formal definition of "structure". Intuitively, a structure is an object which has parts organised in some way. These parts may themselves be structures, perhaps of a different kind; or they may be fragments of expressions, or integers etc., in fact any entities which are to be regarded, for the purposes of defining the TP system, as atomic. The recursiveness in this description indicates a hierarchy in structural relationships which it is natural to represent by a tree; and indeed trees are widely used as structures in TP systems. We will assume that the idea of the set of all possible structures is meaningful, for a given TP system, and call this set $\mathcal{S}$. We will, for the time being, use $S_1, S_2, \text{etc.}$, to refer to structures.

We will define a constructive operation upon structures to be a relation on $\mathcal{S}$. The intuitive meaning is that if $R(S_1, \ldots, S_n)$ holds then $S_n$ can be constructed from $S_1, \ldots, S_{n-1}$ by the TP procedure. We will assume in fact that for every $n$-adic $R$ there is an easily computable $(n-1)$-adic function $\widehat{f}_R$ such that $\widehat{f}_R(S_1, \ldots, S_{n-1})=\{S : R(S_1, \ldots, S_{n-1}, S)\}$. ('Easily' here entails 'effective' but is intended to have a stronger
meaning, as discussed below). We will say that \( S_n \) results from the application of \( R \) to \( S_1, \ldots, S_{n-1} \) when \( R(S_1, \ldots, S_n) \), following this functional idea.

### 1.5.2.1 Derivations of structures

Starting from a given set \( \mathcal{S} \) of structures one can create new structures by repeatedly applying constructive operations to \( \mathcal{S} \) and to new structures so created. We need the concept of the derivation of a structure from a set in this way. There are at least three notational schemes in the literature for handling this idea: Sandewall's lattice-theoretic approach (Sandewall 1971), Kowalski's theorem-proving graphs (Kowalski 1970, 1972), and Landin's polygraphs (Landin 1970). This last, although originally proposed for a different purpose, is historically prior, gives a particularly neat description and will be adopted here.

All of these schemes are graph-theoretic: derivations are defined as certain kinds of labelled graph. This seems to be inevitable. The point is that a derivation must make apparent the detailed history of how the final structure was created, and this involves a record both of the structures in its history and of the particular pattern of applications of constructive operations. This
The pattern is naturally expressed as a graph of some kind.

Following Landin (with some minor modifications) we define a polyedge to be a finite directed graph consisting of a vector of \( n > 0 \) input nodes, a single body node, and a set of \( m > 0 \) output nodes, all distinct, arranged:

![Diagram of a polyedge](image)

There may be more than one arc between an input node and the body node: otherwise no duplicate arcs are allowed. If \( E \) is a polyedge then \( i(E) \) is the vector of input nodes, with duplications if indicated, \( o(E) \) the set of the output nodes and \( b(E) \) the body node, and we will write \( E = \langle i(E), b(E), o(E) \rangle \).

When the vector order is unimportant, we will regard \( i(E) \) as a set. An edge is a polyedge \( E \) in which \( o(E) \) is a singleton.

If \( E_1 \) and \( E_2 \) are polyedges in which \( o(E_1) \cap i(E_2) \neq \emptyset \), we will say that \( E_1 \) and \( E_2 \) are connected. Clearly, a set of polyedges in which every element is connected to some other, is a connected directed graph, which Landin calls a polygraph.

If, for every \( E \) in the graph, all sequences \( E = E_1, E_2, \ldots, E_1, \ldots \), in which \( E_{i+1} \) is connected
to $E_1$, are finite; then the graph will be called a search space graph.

Suppose $N$ is a node in a search space graph, $N \in i(E)$ for some $E$ but $N \notin o(E)$ for any $E$, then $N$ is a premis node of the graph. Similarly if $N \in o(E)$ but $\notin$ any $i(E)$, then $N$ is a conclusion node of the graph.

Let $k$ be a function from sets of nodes to their subsets. We will call $k$ a refinement mapping. If $k(o(E)) \neq \emptyset$ then we will say that $k$ admits $E$.

Let $G$ be a search space graph and $k$ a refinement mapping, then the refined search space graph $k(G)$ is defined inductively as follows:

$k(G)$ contains

(a) all premis nodes of $G$

(b) for every polyedge $E$ in $G$, if all input nodes of $E$ are in $k(G)$, and $k$ admits $E$, the polyedge $\langle i(E), b(E), k(o(E)) \rangle$.

Let $N$ be an output node in a search space graph $G$. Then a derivation graph of $N$ in $G$ is a smallest refinement of $G$ which contains $N$ and every premis of which is a premis of $G$. There may be several derivation graphs of $N$ in $G$. 
Clearly, every derivation graph is finite, and contains just one conclusion node. Indeed, every output node in a derivation is the output node of just one polyedge.

All of this is mere graph theory. To make these graphs carry meaning, we introduce labels.

In general, a labelled graph is a graph together with a function \( f \) from nodes to some set of labels, called a labelling function. We will write \( N_f \) for the label at the node \( N \) in the graph \( <G,f> \), with the obvious extension to sets or vectors of nodes.

Let \( G \) be a search space graph, and \( s \) a labelling function such that
(a) for every body node \( b(E) \), \( b(E)s \) is a constructive operation;
(b) for every input or output node \( N \), \( Ns \) is a structure;
(c) If \( N_1 \) and \( N_2 \) are distinct input or output nodes, then \( N_1 s \neq N_2 s \);
(d) for every polyedge \( E \), \( \sigma(E) \subseteq b(E)_s (i(E)_s) \)
(e) for every pair of polyedges \( E_1, E_2 \),
\[ i(E_1)s = i(E_2)s \supseteq b(E_1)s \neq b(E_2)s \]
then \( <G,s> \) will be called a search space.
If \( G \) is a derivation graph, the search space is a derivation of the label of its conclusion node from the labels of its premises nodes.

It is sometimes convenient to slightly modify these definitions so that \( i(E) \), for a polyedge \( E \), is a set rather than a sequence. All the results on search spaces are true for this modified version.

Let \( \langle G, s \rangle \) be a search space, and \( \mathcal{R} \) a finite set of constructive operations such that \( b(E)s \in \mathcal{R} \) for every \( E \) in \( G \). We will say that \( G \) is a complete relative to \( \mathcal{R} \) if, whenever \( G \) contains nodes \( N_1, \ldots, N_{n-1} \) such that there is an \( R \in \mathcal{R} \) with \( R(N_1s, \ldots, N_{n-1}s, S) \) for some \( S \), then there is a polyedge \( \langle \langle N_1, \ldots, N_{n-1} \rangle, M, K \rangle \) in \( G \) with \( MS= R \) and \( S \in Ks \). That is, all structures which can be derived from the premises of \( G \) by operations in \( \mathcal{R} \), are so derived in \( G \). It is easy to verify that all search spaces with the same premises which are complete relative to \( \mathcal{R} \) are isomorphic, so we will simply speak of the search space defined by \( \mathcal{R} \) from a given set of premises. A structure is derivable from a set of structures if it labels some node in the search space from that set as premises; all relative to some \( \mathcal{R} \), of course.
(i) Polygraphs

(ii) Theorem-proving graphs I

(iii) Theorem-proving graphs II

(iv) Sandewall lattice
1.5.2.1.1 Comparison with other formalisms

The example opposite shows how the same search space would be depicted in this system, in Kowalski's two versions of 'theorem-proving graphs', and in Sandewall's lattice-based presentation. In each case, one derivation of $G$ is indicated in heavy lines, and some arrows are omitted for clarity. Notice that 'and-bundles' in theorem-proving graphs correspond to arcs from the body of a polyedge to an output node; and also correspond to $\cap$-nodes in Sandewall's lattice. The polygraph representation shares with the second theorem-proving graph and the Sandewall lattice, the property that nodes with the same label are identified. (This is indeed the only sensible arrangement, as one can easily make distinctions by adding structure to the labels: while in the other scheme, there is no way of making identifications. This will be discussed again later.)

The polygraph representation has several advantages over the others, most notably that it makes explicit which operations are involved in the derivations. For example, the following distinct search-spaces are rendered indistinguishable in the other presentations:
Note that a derivation is to be carefully distinguished from a proof. A derivation is the history of how a structure was built; a proof is one kind of structure. It is usual in the literature on logic and theorem-proving to identify structures with statements, and their derivations with proofs of the statements. But this is too restrictive for the present discussion. While it is possible to identify proofs with derivations, it is also sometimes essential not to do so. This point will be taken up later.
1.5.3 Localisation and choice

- We now have a sketch of the operation of a TP system: building new structures from old ones by constructive operations. But care is needed. Consider the case of propositional calculus. This is decidable. Hence a constructive operation could apparently be: determine whether or not the input expression is valid; if so, construct "1", otherwise, construct "0". This would not be considered an interesting description of a TP procedure. We would want to know how the test of validity was performed.

This description is too global: we want it to be reduced to a sequence of operations which are local in the sense that each one can be performed and understood without great computational effort. If a certain operation, for example, requires one item to be selected from a large collection, then we must know that this choice does not involve a complex and lengthy computation. The difficulty of this computation must not be comparable with the difficulty of the whole theorem-proving problem; for if it is, the question of effectively describing the TP system is being begged. Again, if any operation requires an alteration to a large structure, we must be sure that it can be reduced to a sequence
of operations which are locally defined on relatively restricted parts of the structure.

This requirement of localisation seems to be part of the very idea of proof, in fact. What makes proofs useful is precisely that they enable us to be convinced of the truth of an assertion even though we cannot intuit its truth directly: for by checking locally that the proof is properly constructed, we are guaranteed that the expression, whose proof it is, is valid. If proofs were not built by the application of operations whose application could be checked locally; if each step required as much computation or insight as the whole proof; then the concept would lose its utility for people as well as for machines. A related theme runs through the whole of computer science: the ideas of computation and program are similarly based upon the reduction of complex wholes to assemblages of simpler parts, whose interrelationships are defined by local operations. (So, for that matter, is a car; or indeed any other complex artificial system). Indeed, another way of phrasing this localisation requirement would be to insist that all operations be programmable.

We need some way of describing how the
operations used by a TP system might be localised. Intuitively, structures will have parts (which might themselves be structures, of perhaps a different kind), and operations will typically be defined upon only certain of these parts. This has two aspects: determining whether and how the operation is applicable; and defining the effect of the operation. It is not necessary that the result of the operation should be a small part of a structure: but the differences between the result and the input should be defined locally.

For example, resolution is defined locally on literals, which are parts of clauses in this sense. For although the result of resolution is a clause - a complete structure - it is defined in terms of local operations on the parent clauses (deletion of literals and unioning). There is no need to consider the clauses as wholes in defining the processes which constitute the resolution rule. (In fact, of course, there is a need to search through every part of the clauses, because of merging. But this search is itself a sequence of local operations: and in any case this is an issue only at the data level, in the usual presentation of resolution in which clauses are sets. It is a
question of the implementation of an operation which is a primitive at the structural level. As was remarked earlier, such questions can have important consequences for efficiency; and this is a case in point. It is often claimed in the literature that having 'expressions' which are sets eliminates the need for the rule of inference $P \lor P \vdash P$. This is nonsense: all that has happened is that the operation in question has been moved from the structural to the implementation level. This can be important conceptually, but the machine still has to do the operation somewhere; so as far as efficiency is concerned, little has changed.

I will not attempt to give a formal and precise definition of "local" in this dissertation. To do so would seem to require the development of an algebraic theory of structure which makes precise the intuitive idea of "part", as discussed earlier.

One consequence of the idea which does seem intuitively clear, and which will be important for what follows, is the fact that locally defined operations will usually be applicable to a given structure or structures in several distinct ways.
For, if one can determine whether or not an operation applies to (a) given structure(s) by examining only a part of the structure(s), then it seems likely that by examining other parts one might find other ways in which the operation is applicable. On the face of things, then, a TP system using locally defined operations is faced with the task of first choosing which parts of the structure(s) it is to operate upon, before deciding to operate. All systems in the literature, in fact, are continually faced with such choices, and there is considerable discussion concerned with how best to make them, if indeed it is possible to make them at all. One might expect that its strategy for dealing with these choices would be an important feature of a TP system, having important consequences for its efficiency and usefulness.

Let us define a choice range to be a set of parts of structures from among which the system is obliged to make choices. For example, any structure which consists of a set of parts related in some way defines that set to be a choice range if there is a structural operation which is locally defined on those parts. (In the algebraic approach we would define choice ranges to be the carriers of the
component $\Omega$-algebras of structures.) A function which picks out an element from any one of a given class of choice ranges will be called a (simple) choice function for that class:

$$\forall R \in \mathcal{R}, f(R) \in R$$

where $\mathcal{R}$ is some set of choice ranges.

For example, in SL-resolution, the set of literals in the leftmost cell of a chain constitutes a choice range, and the selection function is a simple choice function for the set of such ranges. Again, a set of clauses constitutes a choice range in a resolution program, but one for which there is no recursive simple choice function which is adequate for completeness.

Typically a choice range is defined by the set of immediate parts of a structure. If such a range is not the 'topmost' range of a structure, it will be presented to the TP system only after a prior choice, at a higher level, has been made. (For example, the task of choosing a literal from a clause only arises when the clause has been chosen from a set of clauses.) We will say that the former range is dependent upon the latter. We will say that a TP system uses a choice function for a given set of ranges if, whenever it is faced with the task of
making a choice from a range in the set, it applies the function and uses its output as its choice from the range; and if the structure chosen is in the derivation of a proof whenever all structures containing ranges upon which this range is dependent are. For example, the selection function in \( \text{SL-resolution} \) is used in this sense. Whenever the search strategy decides upon a chain, the selection function makes a choice from the range of literals in the leftmost cell: and if the search strategy was correct in its choice, the selection function will be.

The extent to which easily computable choice functions can be used will greatly affect the behaviour of a TP system. Suppose for example, that the system is able to make use of such a function for every class of choice ranges which arises in its workings. Then its task is straightforward: it is essentially faced with no choice at any stage, and simply has to perform a deterministic computation. If however it has no simple choice function for some class of ranges, then it has to consider all possibilities whenever it is faced with a choice range from the class. It is thus obliged to search the space
defined by this class of ranges.

The formalism developed in section 1.5.2 above makes this precise. Suppose a TP system has available a set $R$ of constructive operations and begins with a set $S$ of initial structures, then the complete search space with premises $S$ relative to $R$ contains all derivations which could possibly be generated by the TP system. If we suppose that no choice functions are available to the system, then this complete search space represents the actual space which the system must search. At any given moment it will have generated structures labelling some refined search space of the complete space, and will at that moment be selecting some output of some polyedge, all of whose input nodes have been generated, as a candidate for generation next. Following Kowalski (1970), we will call the process that decides which output of which polyedge to generate next at each moment, the search strategy of the TP system.

Formally, we could perhaps define a search strategy on a search space to be a function from refined search spaces of the space to polyedges in the space. However, search strategies will not be discussed in any detail in this dissertation.
A search strategy can be complete or incomplete in its traversal of the search space. Kowalski discusses this matter very fully. However, there are few results concerning incomplete search strategies, and the concept seems to be of little utility. (This is no doubt due in part to the difficulty of finding a precise formulation of what is meant by a search strategy. The above suggestion is not entirely satisfactory, for example). In this dissertation I will consider only complete search strategies. TP systems which might be regarded as employing an incomplete strategy will here be regarded as employing a complete strategy on a refined subspace containing only those derivations which would have been generated by the incomplete strategy. It is easy to see that this is always possible: for no derivation can be generated by a search strategy unless its subderivations have been generated previously by the search strategy.

Notice that a search strategy is not to be confused with a choice function: it is the lack of a choice function which makes a search strategy necessary.
When some choice functions for certain classes of choice range are available to the TP system, the effect of their use is that various candidate output nodes in the complete search space are not generated. The resulting search space is some refined search space of the complete search space. Exactly how the choice functions determine a refined search space is not defined, since the choice ranges open to the system are not all displayed explicitly in the structure of the search space graph, but are only apparent when the structures labelling nodes are examined in detail.

A search strategy then, is a method of completely surveying a search space which is defined by some initial structures, a set of constructive operations, and a (or several) refinement mapping(s) which remove certain derivations from the consideration of the search strategy.

Consider a TP system in operation. At a given instant there are a number, say \( k \), of output nodes of polyedges which are candidates for generation. This is the branching rate of the search space at that point. More precisely, it is the branching rate of the search space at the refined search space which has been generated by the TP system at that instant.
For example, the branching rate in the search space pictured below, at the search space represented by the heavy lines, in 5.

Note that branching rate is dependent upon the search space and has nothing to do with search strategy. One might intuitively expect that branching rates are an important property of search spaces. Search spaces with high branching rates are expensive to search, for example.
1.5.4 Proofs and relevancy

Since we can speak of parts of a structure, we could regard the whole collection of structures, which the system has constructed at a given moment, as being a single entity with parts which are the original structures. To some extent this is a matter of taste. But it is important to distinguish those structures which are regarded as being proofs or partial proofs. The importance stems from the requirement that a 'reasonable' notion of proof should not allow proofs to contain substructures which are totally irrelevant to the proof.

This intuitive idea of "relevance" needs some delicacy to define properly, as discussed below. But the need for it is clear: for otherwise we could for example, regard the whole structure built by the
TP system as a single partial proof. It would be rather difficult then to discuss the relative efficiencies of TP systems, since the variety of intuitive partial proofs would not be apparent. It is of course possible to hold the view that this is the correct way to describe a TP system, but it doesn't seem to be a very fruitful approach.

The problem of defining 'relevancy' is to avoid making the notion so strong that it is computationally as difficult to recognise relevancy, as the TP problem is in the first place. As above, we will overcome this problem by localisation.

Let $R$ be a constructive operation. We will say that $R$ is nonsensible if, for some $S_1, \ldots, S_{n+1}$, $R(S_1, \ldots, S_{n+1})$ holds and either there is a partition $\alpha$ of $\{1, \ldots, n\}$ such that

$$\bigcap_1 \left( \bigcup_{j=1}^{\alpha(j)} | S_j | \right) = \emptyset$$

or

$$| S_{n+1} | \not\subset \bigcup_{i=1, n} | S_i |$$

And we will say that $R$ is sensible if it is not nonsensible. Briefly, $R$ is sensible if it never applies to structures which can be divided into blocks sharing no nonlogical symbols, and it never
introduces new nonlogical symbols not present in its arguments.

We will say that a TP system is sensible if every operation in it, whose output is a proof or partial proof, is sensible.

This definition clearly avoids the computational pitfalls since it is defined relative to local constructive operations. Moreover it is a very weak definition. The meanings of two structures which share no nonlogical symbols can have no relationships defined by the language. There can be no motivation for putting two such together in constructing a proof, therefore, unless such motivation depends upon extra-linguistic matters.

In fact, every TP system which I know of is sensible. Any system based on matching or unification is of course sensible. In particular, language interpreters such as LISP are sensible, under the natural interpretation of 'proof' as 'computation path'.

Sensibleness has some interesting consequences. First notice that if \( |S| = \emptyset \) then \( S \) cannot be
an input to a sensible operation. This is in keeping with the usual idea of inference rule. (e.g. the null clause cannot resolve with any clause.) Again, let \( S \) be derived from \( \mathcal{S} \); then by a trivial induction, \( |S| \leq |\mathcal{S}| \). Hence if \( |T \cap |\mathcal{S}| = \emptyset \) then \( |T \cap |S| = \emptyset \) and so no sensible operation will accept \( T \) and \( S \) as inputs. It follows that if \( S \) is derived from \( \mathcal{S} \) by sensible operations, and if \( \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \) where \( |\mathcal{S}_1 \cap \mathcal{S}_2| = \emptyset \), then \( S \) is derived wholly from just one of \( \mathcal{S}_1 \) or \( \mathcal{S}_2 \). This fact will be important in the next section. We will call it the Sensibleness Lemma.
1.6 The necessity of search

In this section I prove a version of the celebrated Ehrenfeucht-Rabin theorem. This proof is closely based on one told to me by D. Luckham for the resolution case. The chief interest of the present proof is the generality and weakness of the hypotheses. In fact, most of the terminology developed in section 1.5 has been motivated by an attempt to uncover the true extent of the theorem's application.

We will, temporarily, say that a language is expressive enough if R.E. sets are representable in it. That is, if for any R.E. set $S$ of integers we can effectively determine a statement $S_n$ in the language, for each integer $n$, such that $S_n$ has a proof iff $n \in S$. (First-order logic is expressive enough: propositional calculus isn't expressive enough. It seems intuitively clear that any descriptive language which has an expressive power sufficient to be of interest in artificial intelligence, would be expressive enough.) Let us also say that a language is additive if it has a conjunction operator $\Theta$ in it: that is, if whenever two expressions $E_1$ and $E_2$ are well formed then so is $E = E_1 \Theta E_2$, such that a
proof of $E_1$ or of $E_2$ is a proof of $E$. (For example, first-order logic is additive: if 'proof' means 'demonstration of unsatisfiability', then $\Theta$ is $\land$; otherwise $\Theta$ is $\lor$. Notice that LISP, for example, is not additive: in order for a programming language to be additive it would have to admit a "nondeterministic evaluation" operator. This is the technicality that enables LISP, etc., to escape the Ehrenfeucht-Rabin theorem: see later.).

Finally suppose an expression $E = E_1 \Theta E_2$ is input to a TP system for an additive language; then the set of initial structures will contain, we will assume, structures which correspond to $E_1$ and other structures which correspond to $E_2$. These parts will comprise a choice range for the TP system's constructive operations, if it sensible, since in general not all these parts will share nonlogical symbols with one another. We will call this choice range the input range of the TP system. It is not dependent upon any other range.

**Theorem 1** (Ehrenfeucht-Rabin). Let $L$ be an additive language which is expressive enough. Let $T$ be a complete consistent sensible TP system whose expression input language is $L$. Then the system can use no choice function for its input range.
Proof. Suppose it can. We will show that any two disjoint RE sets \( A \) and \( B \) are recursively separable, i.e., there exists a recursive set \( C \) such that \( A \subseteq C \) and \( B \subseteq \overline{C} \). But this is known to be false, hence the theorem follows by reductio ad absurdum.

For each integer \( n \), we will give a recursive procedure for determining whether \( n \in C \).

1. Form the two expressions \( A_n \) and \( B_n \), ensuring by appropriate choice of vocabulary that \( |A_n| \cap |B_n| = \emptyset \).

2. Apply \( T \) to \( A_n \oplus B_n \). (If \( T \) is not uniform, use some arbitrary advice input.)

3. If \( T \) terminates without having applied a constructive operation, then \( n \in C \).

4. If \( T \) first chooses an initial structure derived from \( A_n \), then \( n \in C \).

5. Otherwise \( n \notin C \).

Clearly \( C \) is recursive and well-defined.
Let \( n \in A \); we will show that \( n \in C \). \( A_n \) has a proof which is also a proof of \( A_n \oplus B_n \), by additivity; and \( B_n \) has no proof. By completeness, \( T \) will construct a proof of \( A_n \oplus B_n \), for any advice input. By sensibleness, any proof - in particular, this proof - must be derived entirely from \( A_n \) or entirely from \( B_n \). But \( B_n \) has no proof, hence it must be derived from \( A_n \). Therefore \( T \) must have chosen from \( A_n \) initially, and so \( n \in C \).

A symmetrical argument shows that \( B \subseteq \bar{C} \). Hence \( C \) separates \( A \) and \( B \).

QED

This theorem is often quoted as showing that a 'perfect' proof procedure is impossible. In our terms, it shows that for at least one choice range, no choice function exists. Thus the TP system cannot operate like a simple computer and generate the proof directly from the input expression by local operations applied in some fixed deterministic manner. Rather it is obliged to consider various possibilities at several points and conduct a search through the space of these possibilities. Of course such a search itself can be reduced to a sequence of local operations, otherwise it would not be
programmable. But it is a much more complex and time-consuming sort of computation. It requires the system to maintain histories of how structures were developed (to avoid loops), to keep explicit lists of current possibilities, etc.. All of this involves heavy computational expense. Worse, search spaces have a way of growing exponentially as computation proceeds.

Let us consider briefly just how wide-ranging is the application of the theorem. We have not assumed that the input language is 'assertional', in particular that it is predicate calculus. We have not used any semantic notions (such as unsatisfiability). We have assumed of the TP system only that it is complete and sensible. (The point of the sensibleness restriction is that it gives some real force to the meaning of "proof", which in turn defines the notions of completeness and additivity.) Consider LISP and a LISP interpreter. If we interpret 'proof' to mean 'computation path', then LISP interpreters are complete, uniform and sensible. Moreover, LISP is expressive enough. And yet the theorem is clearly false for LISP interpreters, which do no searching.
The crucial missing factor is additivity: LISP is not additive. Thus suppose we tried to interpret $\Theta$ as LISP "OR", which would seem to be the most natural interpretation. But even if $B$ has a proof (e.g. if $B$ is a terminating LISP program) it does not follow that $(\text{OR} \ A \ B)$ does, since $A$ may not terminate.

Clearly the only way we could make LISP additive would be to introduce a "nondeterministic" evaluation primitive: let us call it ORR. $(\text{ORR} \ A \ B)$ is now to be evaluated by evaluating $A$ and $B$ in parallel and terminating when either one does. This new language - let us call it LISPR - is additive and obeys all the other conditions. Hence Theorem 1 applies to it. And this is just what we would expect, for an interpreter faced with parallel evaluations is obliged to schedule its effort between them just as a resolution program, say, has to maintain several paths in the search space.

Let us now however return to LISP. We could regard conventional LISP as LISPR with a restriction; that $(\text{ORR} \ A \ B)$ is to be evaluated in a certain special order. That is, we can regard a LISP interpreter as a LISPR interpreter of a rather special kind. (In conventional terms, one which
has a depth-first search strategy. But then this TP system escapes Theorem 1 because it is incomplete. For it sometimes fails to find a proof for (ORR A B) even though there is one.

Notice how the requirements of additivity of the language, and completeness and sensibleness of the TP system, interrelate. Intuitively one can see that the Theorem is true because additivity enables one to present the TP system with a choice which, by completeness and sensibleness, it cannot "refuse" to make. (If we abandon sensibleness then the TP system can so "refuse", in effect, by blurring the distinctions between one proof and the other. Of course, this would be no victory in real terms: one can avoid the letter, but not the spirit, of the Theorem by relaxing the sensibleness requirement, but one would not thereby obtain better TP systems.)

Notice that the Theorem applies equally to uniform and heuristic TP systems, provided that 'completeness' for heuristic systems is interpreted, as here, as being independent of the advice. One would expect one role of the advice in a heuristic TP system would be to control and limit the searches involved.
One reaction to this fact, with which I am in some sympathy, is to suggest that it is evidence for considering heuristic systems whose completeness depends upon their advice. We might call such a system gullible: it can be completely misled by bad advice. It seems possible indeed that any non-gullible system will be insufficiently responsive to its advice to be a useful part of a larger system. However, in this dissertation we will continue to discuss only complete TP search spaces.
2. WHAT USE IS A THEOREM-PROVING SYSTEM?

2.0 Introduction: some views about AI

We now have a fairly detailed picture of what a TP system is, and why the concept of search is so central. In section 3 below we will investigate the first-order predicate calculus case in full detail. Before embarking on this task however, this seems a good moment to enquire into the motivation for studying and constructing TP systems.

I will now abandon all pretense at being descriptive. I believe that the role of theorem-proving technique has been widely misunderstood, both by its practitioners and by its detractors. In this section I will attempt to justify this rather sweeping statement and put forward what I believe to be a more realistic account of the role of theorem-proving in computer science and especially artificial intelligence (AI).

I will proceed by first outlining, and then criticising, several views as to the role of
theorem-proving which I have heard.

In some cases these seem to be implicit in the literature, in some cases they are explicit, and in some cases they have been communicated to me verbally. To give detailed attributions of these views to various people would be invidious in view of the use I intend to make of them, and I will not do so.

They can be summarised as follows.

1. Theorem-proving is a subject which is worth study for its own sake. It has no necessary connection with AI. It is closely related to proof theory and (perhaps slightly less so) to the philosophy of mathematics.

2. Theorem proving is part of AI because it is (defined to be) what certain people do who are in AI groups and who publish their papers in the AI Journal, etc. Moreover, Computing Reviews classifies theorem-proving papers under 'Artificial Intelligence Applications', etc. Hence as a sociological fact, theorem proving is part of AI.
3. Theorem proving is concerned with how to make computers prove theorems (usually, mathematical theorems). It is de facto a part of AI, since proving theorems is a clever thing to do: that is, having the ability to prove a theorem shows that one has a certain amount of wit, be one man or machine.

4. TP systems are a kind of problem-solver, theorem-proving is hence de facto a part of AI since problem-solving is a clever thing to do, etc.. Moreover, as problem-solvers go, TP systems are superior to most in view of the extreme generality of their input language.

5. Theorem-proving is totally irrelevant to AI and is an almost complete waste of time and should be discouraged, because it is concerned with manipulation of assertional languages, and imperative (computational, procedural: these are synonyms for the present purpose) languages are essential for good modelling of knowledge in AI programs.

6. Theorem proving is part of AI because it
is concerned with methods for manipulating assertional languages, and these languages are a useful method of modelling knowledge in an AI program.

These are not all incompatible, of course. Some people seem to hold a view which is a combination of 2., 3. and 4., for example.

I will not discuss the first view in any detail. For the present purpose, it is an admission of defeat. I feel also that, considered as a branch of modern formal logic, theorem-proving is at best somewhat undistinguished. There is simply very little logical motivation for studying computationally feasible proof procedures. The results which are interesting to logic mostly depend only upon the existence of proof procedures (e.g. completeness). And although one might hope that more subtle proof- and model-theoretic results could be extracted from more delicate notions of provability, this will only happen when the ideas of the difficulty of proving a theorem is made precise. As yet there is virtually no sign of this occurring: we don't even have firm results for the propositional case.
The paradigmatic kind of new result in theorem-proving - the existence of yet another complete TP system - only confuses the issue even more.

The second point of view is rather more difficult to deal with. To some extent it is inarguable: to resist it is liable to result in a semantic quarrel about the meaning of 'AI'. I myself, while freely admitting that theorem-proving as presently understood can be regarded as AI in this sociological sense would wish also to distinguish an area within this sociologically defined complex. It is this inner area which I would wish to call 'AI': but I am quite willing, temporarily, to use a different term if that is thought inappropriate. It could be called perhaps, cognitive engineering. I would define this area (under whatever name) to be the organised attempt to understand and construct mechanisms which exhibit general intelligence. Now of course one can argue about what this means, but I will stop the discussion at this point. Notice the word 'general'. This entails a certain view as to what intelligence is; but it is a view which is widely held in the AI field and in psychology, and seems to be in accord with common usage.
From now on throughout this dissertation, I will take the above as my definition of the words 'Artificial Intelligence'. Let me emphasise that I am defining the words in a way different from the sociological definition. I also claim that this definition coincides with one held by the majority of the workers in AI (considered sociologically). The reader is free to disagree with the claim - it is after all a contingent fact - but he must allow me the definition if we are to be able to communicate at all. This disposes of the second view, then.

(I have taken all this trouble over what may seem a trifling matter of terminology because I know from bitter experience how easy it is for communication to fail unless one is very careful at this point. There seems to be a widespread feeling that it is slightly wicked to attempt to define 'AI' other than sociologically. To do so is to make it possible to reject certain work as not being concerned with AI. If we adopt the sociological view, this becomes impossible, by definition, since all one has to do is get his paper published in an AI Journal or contribute to an AI conference and his work automatically becomes AI. This is probably why any attempt
to so define the subject meets with hostility. But I believe it to be important to do so: for otherwise our subject will become a refuge for all kinds of oddball ideas, a ragbag of unconnected miscellanea at the edges of other disciplines. There are already some signs of this. I think we should let general spanticians, cyberneticians, operations researchers, etc., find their own academic homes.)
2.1 TP systems as devices for proving theorems

The third view is the traditional view of the role of theorem-proving, and provided, it seems, the initial motivation for studying TP systems. It also seems popular today. However it rests upon some dubious assumptions.

The first assumption is that the problem of finding a (mathematical) proof of a mathematical theorem is suitably formulated as the problem of finding a (logical) proof of a logical expression. This seems in practice to be a reasonable assumption for some simple mathematical theorems, such as elementary group theory problems, certain other algebraic manipulation theorems, simple number-theoretic problems, etc. In all the examples known to me, the methods which one would informally use to prove the theorems are easily transcribed into (say) PC. That is, each inference step in the informal proof is a first-order valid inference or perhaps a short sequence of such inferences (e.g. the observation that \((a.(b.c)).d = (a.b).(c.d)\), where \(\cdot\) is group product, requires two applications of the associativity axiom). The proofs found by the TP system are thus easy to recognise as
mathematical proofs; and, more importantly, the processes of searching for the logical proof are easy to interpret as processes of searching for mathematical proofs. However, as soon as one looks at nontrivial mathematical theorems this ceases to be true in any obvious sense. Group theory after Lagrange's theorem, for example, requires a lot more than the first-order group axioms: one needs to have groups, subgroups and group elements in one's ontology. This requires at least set theory or higher-order logic. Now the use of set theory enables one to reduce mathematical theorems to PC theorems, but the first-order proofs often no longer reflect in any natural way the mathematical proofs: for the latter do not usually make explicit mention of sets, for example.

The second assumption is more serious. It is that the tasks of proving mathematical theorems are best tackled in isolation. This is a necessary corollary of the use of TP systems. The only vehicle for transferring mathematical expertise from one such task to another is by the advice input to a heuristic TP system. But the kinds of
advice which are available, and which seem to be crucially important to human mathematicians, are phrased in mathematical rather than logical terms. They often depend for example upon concepts of similarity between mathematical proofs and difficulty of a proof; the extent to which certain kinds of proof method (topological vs. algebraic, for example) are useful in various areas, etc.

It is not easy to see how this kind of mathematical knowledge can be phrased in terms acceptable as advice to a TP system, even in principle: for these concepts are not obviously relevant to first-order axiomatisations. In practice, no TP system known to me is able to make even slight use of the kind of mathematical knowledge which would seem to underly reasonable proof-constructing ability.

A direct AI approach to mathematical theorem-proving would concentrate upon this issue of representing useful knowledge about some mathematical field, and how such knowledge might be deployed in the task of proving a theorem. The ways in which such knowledge can interact with theorem-proving processes are richer than can be captured
by a TP subroutine's interactions with another program.

It is notable that the one attempt I know of to marry a conventional TP system to another program which has access to some mathematical knowledge (Kling 197/) suffered from all of these problems. The knowledge in question was that two theorems were 'analogous': but this could only be expressed as a rather simple mapping between nonlogical symbols in the first-order axiomatisations. This knowledge was to be used by the TP system to guide its search for the proof of a theorem, given the proof of an analogous theorem. But such advice could not be expressed to the TP system (nor indeed to any other conventional system I know of). Kling had to be content with merely adjusting the set of support: all structural information implicit in the analogy was lost. He complains rather bitterly about this himself.

For all these reasons I do not feel that the third view is particularly cogent. In any case, even if in practice TP systems were good at proving mathematical theorems, I would not regard that as a
valid claim to relevance to AI. It may be that there are unintelligent ways of getting theorems proven (this is certainly true for easy theorems, for example, as Siklossy's (1971) work shows clearly): that is, ways which do not require the deployment of appropriate knowledge. I do not regard performance on a particular task or class of tasks as sufficient evidence alone for intelligence.

I will return to this subject after discussing problem-solving systems.
2.2 TP systems as problem solving subroutines

This view seems to me to be partly correct. A uniform TP system certainly is best regarded as a problem-solving system. And most conventional heuristic TP systems can also be appropriately described in this way in view of the kinds of advice they are designed to accept. However, I do not regard the activity of constructing problem-solving systems as obviously part of AI. Although historically important, this activity is no longer clearly useful, and is certainly not directly relevant to AI.

In order to justify these statements I must describe just what is meant by a problem-solving system. (I am using the term in the sense in which GPS, Sandewall's PPS, Quinlan & Hunt's FDS, Michie's Graph Traverser, etc., are representative problem-solving systems. This seems to be the generally accepted usage.)
2.2.1 Problem-solving systems

A problem-solving (PS) system accepts as inputs problems, stated in some language (the problem language) associated with the PS system, and generates solutions to the problems. It may also accept other inputs which are supposed to help the system in its task of generating solutions: we will call this advice. The similarity between this and a TP system is obvious: the expression language is the problem language, expressions are problems, proofs are solutions.

There are several extra methodological assumptions involved in PS system construction, however. The assumptions include:

(a) The problems should pose a real intellectual challenge to the PS system. If advice is input, for example, it should not embody the solution in any obvious sense: that would be cheating.

(b) The PS system should be as general as possible. That is, a system which can accept problems from a wide range of problem 'areas' (considered intuitively) is superior to one which is closely tied to a specific area.
(c) The PS system should be as powerful as possible. That is, it should be efficient at finding solutions to its problems.

It is widely recognised that (c) is incompatible with (b), and it is often, apparently, thought that (b) is prior: that generality is more important than power, by and large. Of course, some power is essential. Nobody is interested in PS systems which solve no problems. But it is remarkable how close to this ludicrous extreme some systems come, notably GPS. One has to simplify problems to an extreme extent to get GPS to solve them, even with considerable advice. A simple exhaustive search, efficiently programmed, usually performs better.

As evidence for (a), note that Newell & Shaw declare, in an important paper on GPS methodology, that a vital aspect of problem-solving is that the person who states the problem in the problem language, and who gives the advice, should have no knowledge of the internal workings of the PS system. This is their way of ensuring that the advice and the problem statement do not cheat. Again, Michie in
his review of GPS (Michie 19) chastises it on the grounds that its problem language is so complex that it is often easier to solve simple puzzles oneself than to work out how to say them to GPS (and more complex puzzles are beyond its power). But why is this a criticism? Because, presumably, one feels that the work involved in phrasing the problem in the problem language is partly contributing to solving the problem: to have one's intuitive problem translated into so highly formalised a form is to have already 'nearly' solved it, and all the PS system has to do is finish off one's task. But if the whole task can be accomplished more easily, that makes the PS system redundant. This line of reasoning depends upon the idea that a problem is to be regarded as a challenge.

Let us ask: why are these assumptions made? They all follow fairly directly from the basic idea that:

(d) The PS system should be seen to possess at least a modicum of general intelligence.

This idea arose, I believe, in the early days of AI (about eighteen years ago), as a response to an
uninformed but popular objection to the idea of an intelligent program, which could be called the 'taperecorder' objection. According to this view, a program which seems to exhibit intelligent behaviour merely embodies part of the intelligence of its designer, as a taperecorder might play back a recording of a human voice. Just as we do not regard the taperecorder as having the power of speech, so we should not regard the program as intelligent. (This is often supposed to be an inevitable consequence of the fact that computers obey programs written in their instruction codes slavishly, and therefore cannot exhibit initiative or plasticity.) If one points to a program which indubitably behaves in a rather intelligent way, (say a chess playing program), in an attempt to refute this view, one will meet the response that the writer of the program has merely encoded into some simple mechanism all his own insights into chess. No credit should accrue to the program, only to the programmer.

The popularity of this view among laymen is so widespread and deeply entrenched that it must be
given consideration. Nobody who knows very much about actual modern AI programs would find it very plausible, but in the early days it probably seemed a powerful argument. (I have myself heard it forcibly asserted by apparently sane men of considerable academic accomplishments, even as recently as 1972. True, they clearly knew virtually nothing about actual AI programs.). To refute it, one would have had to exhibit a program which clearly behaved intelligently and which equally clearly did not merely 'play back' the intelligence of its designer. (One way, perhaps, would be to exhibit a program which behaved intelligently and which had no designer: a "self-organising" program. Hence, I suggest, much of the early interest in such devices.) PS systems are a clear answer to this challenge. They embody no special knowledge about a particular domain, and no such knowledge can be inserted into them in some cunning way - the 'no-cheating' rules are devised just for this purpose -, they exhibit general intelligence, problem-solving ability, etc.. The paradigm task they are supposed to be able to tackle is precisely one in which no domain-dependent knowledge can be used: meaningless symbolic manipulations, for
example. (GPS evolved from the Logic Theorist. Both LT and FDS were explicitly compared with human performance on various tasks, and in both cases great care was taken to ensure that the human subjects were unable to bring to bear any special knowledge.) They are thus clearly not taperecorders, since there is no 'recording'. They do, in fact, clearly refute the taperecorder view, and have been important as paradigms in psychological theory for precisely this reason.

However, let us return to AI. It is surely no longer necessary to refute the 'taperecorder' view: our subject has outgrown such naïve objections. Are there any further uses then for the PS methodology?

One valuable effect of PS systems is that several apparently generally useful mechanisms have been discovered. One thinks immediately of heuristic search, for example. GPS has given us means-end analysis and some ideas upon how to use measures of syntactic differences between expressions, and notions of simplicity of problems. All of this is very laudable and no doubt will be of lasting
value: and insofar as they are regarded as test-beds for trying out such ideas, PS systems have no doubt a permanent place in AI research. For example, Sandewall's PPS system had such a role. But no other purpose is served by PS systems: and attempts to incorporate actual systems into less trivial AI programs have, predictably, failed.

The mechanisms embedded in PS systems, perhaps cleaned up a little, have their uses (e.g. means-end analysis in STRIPS). But the methodological packaging is actively detrimental to such incorporation: it makes communication between the mechanism and other parts of the program virtually impossible, for example. The methodological assumptions force a PS system to have the control structure of a subroutine. It would be anathema for a PS system to actively communicate with some other process, especially if the pattern of communication between them were in any sense collaborative with regard to getting the problem solved. Many workers have indeed referred, with evident approval, to the use of "problem-solving subroutines" in AI programs. (And in fact they have often meant a TP system.)
2.2.2 Representations of knowledge: a rival methodology

In contrast to 'problem-solving', there has been an attitude in AI from the very beginning (it is spelled out clearly in (McCarthy 1959), for example) that the use of appropriately represented knowledge is a key to intelligence. The representation issue has always been central to the subject.

(We note here in passing that there are two distinct meanings given in the literature to "the representation issue". Newell (1966) and Amarel (1968, 1971 and etc.) are discussing representations of problems, in the sense of problem-solving. Amarel, for example, is much concerned with ways in which an appropriate representation of a problem can ease the burden of the problem-solver. McCarthy however is talking about representations of knowledge to be used by a program in, among other things, solving problems. And this latter is the more recent concern of the "computation-modelling" school of MIT workers. The two meanings of "representation" are perhaps related but must be carefully distinguished. I am using the words in McCarthy's sense.)
In the long run, this view recognises programs will be self-sufficient, will be equipped with ways of acquiring knowledge for themselves and of modifying and improving it in the light of experience. But this question of learning is subsevient to the prior question of representation. Before we can consider adjusting or acquiring knowledge we must have at least a few clear ideas as to what knowledge looks like. This is spelt out clearly in McCarthy's 1959 paper and had its first positive demonstration in Winston's thesis (1970). I mention it here again in order to emphasise that when we write a program embodying large areas of our own knowledge, this is done as a step in a larger and longer-term enterprise: which is part of this methodology's answer to the "tape-recorder" objection.

This attitude does however, place some strong requirements on the ways in which knowledge is represented and used. For example, it entails in the medium term, that representational schemes should be sufficiently transparent that it is at least plausible that a learning program could generate such knowledge.
Knowledge is represented, often, as expressions in a language or languages. But the mere storing of expressions does not give a program useful knowledge. It must be able to use the expressions. Exactly how such use is described depends upon the cultural background against which the language is defined: thus if the language is considered a logic, "use" will include the derivation of consequences; if a programming language, the running of programs; if a language of structures, structural operations; etc. But in all of these cases there will be some program which takes the expressions and performs appropriate operations or transformations: a deduction program; an interpreter; a data-structure package; etc. Let us use the neutral term "processor" to refer to this device. We now have a third example of a language/device pair, representations and processors, to compare with PS systems. But there are important differences in methodological overtones.

For a start, there is absolutely no need to assume that the representation should be stated without knowledge of the internal operations of the processor. In no sense is it desirable that the representations should constitute a challenge to the computational power of the processor. On the
contrary, the simpler and more efficient the processor the better, and the richer the interactions between various kinds of knowledge and their processors, the better. (Several of the larger AI programs depend crucially upon such possibilities: Winograd's program and the SRI robot system both use several distinct interpreters with complex interactions between them.) This follows from the lack of any need to claim that the processor is intelligent. On the contrary, it is the whole system, including the representations as well as the processor(s), and probably other parts as well, which is supposed to exhibit intelligent behaviour. To attempt to ascribe this intelligence to any particular piece of the whole system is inappropriate.

If one attempts to use a PS system as a processor, identifying knowledge with problem statement, the "no-cheating" rules completely frustrate one's desire to achieve the kind of rich interaction between knowledge and process which seems to be required in difficult AI programming. The computational issues, for instance, meaning-directed search, of heterarchical control flow, of data-base maintenance; none of them can even arise in the PS approach, since there is no way of getting knowledge in the processor except via the advice input, and this is purposely
designed so that detailed control over the processing is made impossible. To use a metaphor from scene analysis (Winston 1973), the "no-cheating" rules deliberately set up hourglass bottlenecks in the system, and particularly vicious ones at that.

Secondly, the generality of the language is not so important. On the contrary, there is a clear need at the present time in AI programming to find representational languages and corresponding processors suitable for quite specific areas of knowledge. Examples include vision (knowledge of shape and shadows) and understanding of children's stories and narrative text more generally (knowledge of elementary physical causality, of human motivation, of beliefs and the ways in which they can be mistaken, etc.). This is rather a delicate point, for one wants representations which are of "general" applicability within a given subject-matter; and very general representational languages can often express particular kinds of knowledge. But excessive generality (as in the predicate calculus) can be an embarrassment: the frame problem and the qualification problem (McCarthy and Hayes 1969, Hayes 1971) attest to this. In any case, there is little
motivation for finding representations which can simultaneously handle such widely different kinds of knowledge as those covered by the problems GPS was able to tackle (Ernst & Newell 1969).

In practice one finds that processors and PS systems are presented to the public in rather different ways by their authors. For example, processors often have manuals explaining in detail how they work and how their behaviour can be modified by appropriate representations. This would be anathema for a PS system, of course. This contrast in attitudes can perhaps best be illustrated by imaginary conversations between a hypothetical user of two such systems and their authors, whom we will call respectively PROC and PROB:

(1) USER: I don't know how your thing works, so I can't see the best way of expressing my knowledge/problem.

PROC: Have you read the manual?

PROB: Mind your own business! Just write it down, and the system will solve it.

(2) USER: Your thing doesn't act appropriately with my input.
PROC: Perhaps you're not very good at writing down knowledge representations. Why not re-read the manual and try again, perhaps using a different technique?

PROB: I'll have to make the system more powerful.

(3) USER: No matter how hard I try, I just can't express myself adequately in your thing's language.

PROC: Evidently the knowledge you're trying to express isn't in the range of applicability of the representation.

PROB: I'll have to make the system more general.

(4) USER: It's harder to express myself in your language than it is to do the task, which I want the program to perform, myself in the first place.

PROC: So what? Did you think it was going to be easy to make an intelligent program?

PROB: (rather embarrassed). Oh, er, yes. Well, we're working on a more human-oriented input language, etc..
This last perhaps illustrates most clearly the difference. It also emphasises that even if the PS approach were successful, it would do little more than scratch the surface of AI tasks. For, most areas in which intelligence is deployed simply do not admit of description as "problems" requiring "solution". Scene analysis, understanding mathematics, understanding natural language: none of these are "problems" in the PS sense. Of course it is easier to see things than it is to write a seeing program, or even to express the sort of knowledge needed in a seeing program.

To sum up, the PS methodology, although historically important, is not especially successful in practice and is antithetical to the use of knowledge in complex AI programs.
2.2.3 The role of TP systems

Formally, TP systems, PS systems and processors are indistinguishable. But there are dramatic methodological contrasts between the latter two. Is a TP system to be regarded as a problem-solver or as a processor?

The bulk of the work in the development of conventional theorem-proving has either explicitly or implicitly accepted that TP systems are a species of PS system. As evidence, notice the following:

(a) The kinds of advice which have been considered as inputs to TP systems (see 1.1 above) all obey the PS "no-cheating" conditions.

(b) Conventional TP systems all have the character of subroutines. Every use of a TP system in a larger system has accommodated to this. As already noted, authors have sometimes complained of the resulting inflexibility.

(c) Often, authors refer to conventional systems explicitly as "problem-solving subroutines", or draw parallels or contrasts with other problem-solving systems such as GPS.
(d) There is great interest in relative evaluations of TP systems with regard to efficiency (cf. 1.2 above) and generality. These are vitally important issues in the PS methodology, but far less so otherwise. On the other hand, evaluations of conventional TP systems which would be relevant to their role as processors, is almost completely absent from the literature. (For example, one would hope that they might be able to accept highly structured advice concerning proof generation, since such advice is often available. But I know of no discussion of this issue in the conventional TP literature.)

However, the main reason why I am convinced that this attitude, or something closely similar to it, is common in theorem-proving work, is that I myself held the "problem-solving" view for several years while working in the theorem-proving field. And almost everyone else in the field seemed to implicitly agree with it. We were certainly very conscious of the need to avoid "cheating", and scornful of the occasional piece of work which seemed to succumb to temptation. I remember very few occasions when
methodological disagreements made communication difficult. On the contrary, the theorem-proving field was notable in AI generally for its homogeneity and sense of purpose. Since that time I have argued against the PS methodology in theorem-proving and have met in response explicit defenses for it. I have heard it declared for example, that the subroutine character of conventional TP systems is desirable; that advances in the power of TP systems will continue, making them more and more useful; and, most importantly, that their inability to accept structured advice is not a disadvantage because such knowledge is inappropriate to the task involved. It would be hard to find a clearer statement of the PS idea.

Thus, I claim, conventional TP methodology is wholly consistent with the view that TP systems are problem-solvers, as the fourth view mentioned at the beginning of the chapter suggests. And, I further claim, this is inconsistent with, or at best irrelevant to, the use of TP systems in AI.

Before leaving this topic, recall the third view,
that TP systems are devices for getting theorems proven. My criticism of this can be succinctly stated in the terms of this section by saying that a theorem-proving problem-solver is inherently unable to make use of the kind of knowledge one would expect a "mathematical expert" to have: so it is unlikely to be a useful part of an expert mathematical AI program.
2.3 Computational vs. assertional knowledge

The fifth view - that theorem-proving should be suppressed because it is unconcerned with computational representations of knowledge - is currently very popular and influential in AI. This popularity springs in part, no doubt, from the very active and professionally managed public relations exercise which the MIT school have mounted over the past few years. But it is true that many successful AI programs have been written within its aegis, notably Winograd's. Evidently it cannot be dismissed as rapidly as some defenders of theorem-proving would like. (Darlington 1973, for example).

The view is, I believe, an exasperating blend of much deep insight and some misunderstanding. In order to clear up the confusion I must ask the reader to bear with me through some delicate distinctions.

The argument used by the proponents of this view runs something as follows, using the present terminology. (1) A conventional TP system takes as its expression input purely declarative statements,
typically in predicate calculus. (2) Its advice
input, if any, is very poorly structured and of little
use therefore, in helping the system cope with
large combinatorial searches. (3) Typically when
one uses such a declarative language to model
knowledge in an AI program, the search space of
possible inferences which results is very large.
(4) Conventional TP systems are of very little use,
therefore. (5) If one uses a suitably designed
programming language, with its interpreter, as
one's representation of knowledge and associated
processor, these large searches can be avoided or
at least brought under control by using the control
aspects of the language. Responsibility for the
behaviour of the processor will then devolve more
upon the knowledge in the representation than upon
the mechanism of the processor. And this is in
accord with the more sophisticated view of the
role of knowledge-representations in AI programming.
(6) Therefore, we should be concerned with devising
suitable such programming languages and investigating
their use. (7) These languages, being procedural,
are qualitatively different from declarative
languages. They have in particular a quite different
semantics: the Tarskian model theory of assertional
languages is inappropriate for AI. (8) Work on
techniques for manipulating assertional languages - and hence, in fact, all the work in the conventional theorem-proving literature - is therefore quite irrelevant to this goal.

Now, all of this up to and including (6) is entirely accurate; but (7) and (8) are a basic error.

The proponents of this view correctly react against the prevailing methodology in the TP literature, but they are mistaken in their diagnosis of the fault. It is not that declarative expressions are by their nature wrong or inappropriate: only that alone they are insufficient. A uniform, or nearly uniform, TP system whose expression input is predicate calculus is too weak to be useful. But a non-uniform such system, which was able to accept complex descriptive advice, could thereby overcome the weakness. The knowledge-representation which such a processor accepts ("processor" because it couldn't be a PS system) has two parts: the assertional (logical) expressions, and descriptions of what the processor is to do with these expressions. This is the missing input which TP systems have long lacked.
The need for these finely drawn distinctions comes from the fact that this pair, logic + metaductive advice, could well be said to constitute a programming language. Indeed I will argue in some detail below that this is a fruitful way of describing some existing programming languages as well as approaching the design of new ones, more suitable for AI work. I am thus not arguing against the MIT emphasis upon procedural representations, provided the word "procedural" is understood in a sufficiently broad sense. My disagreement is only with the idea that procedural is inconsistent with assertional, especially with regard to semantics. I have discussed earlier the close linguistic relationships between imperatives and declaratives (section 1.4.1). A close reading of the MIT polemical literature suggests that their usage of "procedural" is consistent with mine, in fact.

The real significance of the MIT attack on classical theorem-proving methodology, with which I am wholly in sympathy, is the idea that representations should take responsibility for controlling the behaviour of the processor. It is this that gives force to the word "procedural".
It might be objected that one gains nothing by regarding programming languages in the way I suggest; perhaps even that it is counterproductive (Hewitt, personal communication). In the next sections I will attempt to argue in its favour.

Before leaving this topic it should be declared that although my disillusion with conventional TP methodology predated any close acquaintance with the MIT philosophy (Hayes 1970, 1971; when this latter was written I had read only Hewitt's first brief non-polemical account of PLANNER (Hewitt 1969)), my present views owe a very great deal to conversations with Bruce Anderson, Carl Hewitt, Johns Rulifson, Gerald Sussman and especially Seymour Papert. The critique of "problem-solving" is my own.
2.4. Logics as representations of knowledge

The sixth view - that assertional languages are a valuable medium for representing knowledge in AI programs, and theorem-proving technique has some utility by being concerned with manipulations of such languages - has quite a long history in AI. It underlies Cordell Green's work (Green 1969), for example. It is closely related to the work of McCarthy (1959) and more lately McCarthy and myself (McCarthy & Hayes 1969, Hayes 1971) on special-purpose logical languages for AI representations. And it is the view in support of which I now wish to argue, after some preliminary remarks.

(a) It is totally inconsistent with the view of TP systems as problem-solvers. And as I have argued above, much work in theorem-proving is at least in part based upon that view. So a good deal of theorem-proving work will be irrelevant. Most of the dozens of minor completeness results for the various ad-hoc refinements of resolution are irrelevant, for example. But on the other hand many results have been obtained which uncover fundamental computational properties of predicate calculus and related languages, and which will continue to be of basic importance.
to the future of AI programming.

(b) Secondly, I wish to emphasise again that this view is not put forward in opposition to main points of the MIT computation-modelling philosophy, appropriately understood, even though it differs on some issues.

(c) Finally, note that I am not arguing that all the knowledge stored in an AI program should be represented in PC. Such reductionism is consistent only with the problem-solving methodology. On the contrary, realistic AI programming involves many different representational issues, including probably the use of data structures to "directly" model aspects of the world (Sloman 1971), as well as considerable brute programming skill (Minsky & Papert 1971). But I do advocate that PC-like languages, supplemented by suitable control advice, are a useful way of representing many kinds of knowledge, and have a permanent place in the AI repertoire. In particular, they are not made obsolete by the new programming languages PLANNER, CONNIVER, POPLER and QA4.

Support for the use of logical representations
of knowledge must ultimately come from successful programs which make such use. Green's work, for example, although now out of date and in retrospect clearly hampered by the prevailing PS methodology, was at the time such a successful demonstration. At present however such a convincing demonstration must await the construction of a TP system which can accept structured advice, and the development of techniques for programming such a device. The rest of this dissertation will be largely taken up with preliminary studies towards this end.

The first argument is that it seems obvious from introspection and from our linguistic habits that a great deal of human knowledge is suitably modelled by the generation of deductive consequences from such beliefs. This view has been disputed, but on ground which I find quite unconvincing (Davies & Isard 1973), and it has successfully motivated a great deal of work in philosophical logic and the philosophical analysis of language.

Secondly, the use of logical representations enables one to take practical advantage of this work in proof theory (Prawitz 1971). Philosophical
logicians and analytical philosophers more generally are often concerned with finding formal, or at least precise, models of intuitive reasonings and mental processes. They have been working for a considerable time, and have amassed several important results and methodological guidelines. I do not claim that they have solved any problems facing AI, but it seems foolish not to try to take advantage of what they have done. I have mentioned some relevant work in several papers cited earlier.
2.4.1 Semantics

One important insight obtainable from philosophical logic is the role of semantics. A well-defined semantic theory is a basic prerequisite for a representational language. For it is only such a semantics which defines the exact relationship between the representational expressions and the things they represent. While it is often possible in practice to get a long way using a representational scheme in an ad-hoc way without having any very precise semantics, in the long run a semantic theory will clearly be essential. Even in the short term it is often very helpful to have a precise description of the ontology of a representational scheme (see 2.4.2 below, for instance.). I would also argue that on general grounds of scholarliness it behoves AI to investigate precisely a topic which is so fundamental to it as that of representation.

Now, the Taskian model theory of PC and its relatives is the only precise semantic theory which I know of for a language which even approximates to the sort of richness one requires for representations of knowledge. This theory provides a clear account of
the ontological assumptions underlying PC. It allows the use of powerful model-theoretic arguments in the metatheory of the language, greatly simplifying the development of formal systems. (The formal analysis of intuitive reasonings involving the modalities such as tense, for instance, has only been made possible by the development of a clear semantic theory for the otherwise almost empty formalisms.) It makes it possible to give precise meaning to intuitive ideas of completeness. But perhaps most important of all, it shows how PC can represent partial knowledge about a complex domain. This ability to partially describe a subject, and the related fact that one can add further statements later, building up a fuller description in a piecemeal fashion, is of crucial importance in AI, as McCarthy has emphasised (McCarthy & Hayes 1969).

It might be objected that most complicated languages - in particular, programming languages - admit large expressions to be built up in a piecemeal fashion by the combination of subexpressions. This is true; but without a semantic theory there is no corresponding way of understanding the meaning of the whole in terms of the meanings of the parts.
Since the early years of computer science there has been a concern to elucidate a precise and realistic semantics of programming languages. This has often gone hand in hand with reforming zeal, so that many more recent programming languages are designed to make apparent certain semantically motivated distinctions. Two examples: Strachey's $\xi$-value/$\tau$-value semantics for explaining assignment and data structures, which is one of the foundations of CPL and BCPL; and the idea of functional programming which underlies LISP and has provided the backbone of the theory of computation for some ten years (McCarthy 1963, Landin 1964, Scott 1970, for example). This latter is often accompanied by a desire to eliminate assignment and goto from programming languages.

One can clearly see two distinct approaches to defining such semantics. In the first, the meanings of expressions are defined in terms of an idealised 'abstract' computer. This approach depends upon the notions of state in the abstract machine and state-transformation, and is valuable for elucidating programming languages which essentially depend upon the corresponding notions in real-life computers,
such as ALGOL. In the second, there is no such reference to any machine-like features; rather, the expressions of the language are understood to refer to functions, (understood mathematically) upon domains of data items, such as integers. (Notice that the Landin approach of mapping programming languages into the lambda-calculus is neutral in this respect until one decides how to regard the lambda calculus. Thus Landin's own account (1964) via the SECD machine, is of the first variety, while Scott's more recent work (Scott 1970) is of the second.).

Now this second approach depends crucially upon the Church-Rosser theorem. This states that, under very general conditions, the particular sequence of application of computational steps does not affect the final outcome. Without this result, it would be quite implausible to construe, for example, LISP computations as being concerned with mathematically defined functions, since the individual computational steps would not be universally valid in the intended domain, and the meanings of LISP expressions would depend
crucially upon the particular sequencing of deductive steps used by a LISP interpreter. While the functional approach has proven valuable in the understanding of LISP-like languages, it cannot for this reason be extended in any direct way to a semantics for nondeterministic languages, or, more generally, languages whose meaning is sensitive to the flow of control through parts of a program. Indeed, the very notion of flow of control depends upon an interpreting device which embodies "control". There is every evidence that languages of this latter kind are needed in AI programming. Certainly all the new languages mentioned earlier have this character.

It seems, therefore, that the most fruitful approach to a semantics for the control aspects for an AI programming language - in the present terms, the structured advice input to a heuristic TP system - would be via an abstract interpreter. That is, an "idealised" TP system.

Now, the ontology of such a language will contain entities like partial proofs, relationships between these, properties of them, such as their
complexity, states of knowledge which the system might have at various times, events such as the production of a new partial proof, and so on. The language will have primitives for describing or instructing the flow of control through various parts of the TP system, and probably also, between the TP system itself and other parts of the larger system. (For example, the TP system might act as a co-routine relative to other parts.) The point I wish to emphasise is that this ontology and these primitives are quite different from the ontology of the representational language. Although the semantical theories of logic and of conventional programming languages have many points of contact, (for instance, the question of referential transparency arises in both cases with closely similar meanings ) the kinds of entity they refer to are usually very different. Logical ontologies can encompass physical objects, and other exotica. Computational ontologies, on the other hand, contain only entities within the computer. AI programming needs both, but it should not confuse them.
2.4.2 Examples

There are several examples of the need for an explicit ontology in current AI programming languages. PLANNER contains at least two infelicities due, I believe, to an insufficient care for these matters.

The first concerns situation variables. (McCarthy 1960) The idea of having situations, possible worlds or the like in one's ontology has proven basic to the representation of common sense reasonings both in philosophical logic and AI. There is not space here to summarise the extensive literature on this topic, so I will assume the reader is fairly familiar with the ideas involved. The PLANNER authors, observing the difficulties associated with the frame problem, sought to suppress mention of situations in the language. (Their "THERaSE" mechanism for handling the frame problem is not, in fact, inconsistent with the use of explicit situation variables, as described below.) However, in common with everyone else, they were obliged to use situations in their ontology. An unfortunate but inevitable consequence
is that they are unable to distinguish between an expression \( \phi(a) \), where \( a \) denotes a situation, and the related but different expression \( \exists s \phi(s) \). These are both written simply as \( \phi \), with all reference to the situation suppressed. The system thus has no way of knowing, when asked to prove \( \phi \), whether this refers to the current situation, some other named situation, or to any situation. In practice this means that the user of the system has to try to keep track of these distinctions by the very awkward device of duplicating all his theorems and having essentially two separate theories, a 'static' one, in which no state-transformations are contemplated, and a 'dynamic' one; and recommending, when he asks the system to prove \( \phi \), which theory it is to use. This distinguishes "true-now" from "true-sometime", but is still unable to handle the case where we want to refer to some other definite situation. (e.g.: where were you last Saturday?)

The recent but closely related system POPLER attempts to overcome the difficulty by having two different ways of asking the system to prove an
expression (two different THGOAL primitives). But this is clearly only a way of aiding the user to manage the above mentioned ad-hoc partial solution. The real difficulty is that there are entities which are central to the ontology of the system, which have a role in defining the meaning of expressions of the language, but which cannot be referred to. It is exactly analogous to the difficulties which beset quantified modal logic (although far easier to cure).

The second concerns PLANNER's lack of negation. There is no negation operator because there is supposed not be any notion of truth involved. But users of the system typically do have a fairly conventional notion of truth in mind, and often wish to distinguish falsehoods from assertions which merely cannot be proved. That is, if we take \( \vdash \phi \) to mean '\( \phi \) is provable', to distinguish \( \vdash \neg \phi \) from \( \nvDash \phi \). Now PLANNER does have something like the latter: (THNOT \( \phi \)) succeeds - is considered provable by the system - just when \( \phi \) doesn't, i.e. when \( \vdash \neg \phi \) is false. If one wishes to express that \( \phi \) is false, one can arrange for a failure to be generated whenever an
attempt is made to assert or prove it. But this does not distinguish the two cases mentioned. Again, an awkwardness in the system arises directly from the lack of clear semantics. For the semantic distinction between the two negations; that is, the difference in meaning, is quite clear. (In Hewitt's defence it should be said that he seems to want to relate PLANNER to intuitionistic logic, where this distinction is less clear. But the semantics of intuitionism is itself somewhat murky.)

PLANNER's THNOT primitive provides a third example of the importance of semantic clarity. In attempting to find a semantics for THNOT, I have been led directly to an important expressive device. The PLANNER authors missed this because, I suggest, they were content to ignore the semantic consequences of their formalisms.

What does THNOT mean? The expression (THNOT φ) succeeds when φ fails, i.e. is unprovable. But this means unprovable relative to the current state of the system. For example, we may evaluate (THNOT φ) and have it succeed, then THASSERT some new facts,
altering the state of the system, and now an evaluation of \((\text{THNOT } \phi)\) may fail since \(\phi\) has become provable from the \text{THASSERTions}. This sort of behaviour is extremely useful in practice, as John McCarthy has pointed out to me: it suggests an approach to the qualification problem (McCarthy & Hayes 1969, and see below). But if we regard the first \((\text{THNOT } \phi)\) and the second as being the same expression, we are in logical trouble, since in any logic with a Tarskian semantics, we cannot make a theorem false by adding new assertions to a theory. If \(\cap \vdash \text{Th}\), then \(\cap \cup \Delta \vdash \text{Th}\) for any \(\Delta\).

A possible response is to claim that THNOT has no Tarskian semantics, of course: one then has to explain what sort of semantics it does have. PLANNER avoids the implicit inconsistency by an "ostrich" approach. It never remembers whether a call of \((\text{THNOT } \phi)\) succeeded or failed (THNOT is only syntactically legal in a THGOAL: it can't be THASSERTed.), so is not put out by its change of heart when the database changes. This works in an operational sense but rather militates against clear understanding, I feel.
Let us however regard a call of \((\text{THNOT } \phi)\) to be asking the question "is \(\phi\) unprovable from the current state of belief?": this is clearly what is in fact meant by such a call. Let us transcribe it as \(\neg S_1 \vDash \phi\); where \(S_1\) denotes the current state of belief, and will be called a state expression. State expressions are terms which denote states of belief, following the methodological requirement that entities which control meaning must be able to be referred to. Now, the second call of \((\text{THNOT } \phi)\) will be transcribed \(\neg S_2 \vDash \phi\), where \(S_2\) denotes the changed state of belief after the assertions. The first might be true, the second false: they are, after all, different expressions.

Not only does this approach make the semantic basis and the ontologic suppositions explicit, it also, I claim, more closely corresponds to introspective reasonings. Consider McCarthy's example. I decide to drive to the airport: I can think of no objections ((THNOT(KAPUT CAR))). I then find, however that the tyre is flat ((THASSERT (FLAT-TYRE CAR)), and assuming we have (FLAT-TYRE X) ⊨ (KAPUT X) ) and decide therefore
that after all, I can't. (THNOT (KAPUT CAR) now fails). But I don't now think that formerly, when I intended to drive, I was crazy or stupid: only that I was misinformed. If you asked me, say, whether it was reasonable to have then decided to drive, I would still say it was. I would say, "I didn't know about the flat tyre then." (The PLANNER approach is to retort, when challenged about such a change of heart, "I don't remember saying that!"

These state expressions seem to be very useful. A wide range of intuitively clear natural-language assertions and reasonings seem to involve such states of belief quite explicitly, for example: "I used to think he was going to become an actor; but that was before I had met his mother". And they provide a good starting point for tackling the qualification problem which is a fundamental problem in robot reasoning. Again, if we identify the times of belief states with those of situations we are able to write expressions which relate internal states to external observables. So far as I am aware, this is the first time any formal calculus has been
proposed which enables this to be done, and it seems to be quite important. A vision program, for example, might well need to relate past states of hypothesis-formation to the phenomenal inputs which give rise to them: "when I thought this line was the edge of the chair, I was examining the top left-hand corner of the picture".

To make the ideas quite precise one has to specify more exactly what is meant by a 'state of belief', and provide some syntactic machinery for naming them. Care is of course needed here to avoid making the system so strong that it becomes inconsistent, along the lines of Gödel's second theorem. Basically one must not include too rich a descriptive apparatus for describing possible states of belief. The present point is made merely by the observation that some such notion makes sense of THNOT, however.

A quite useful facility (richer in some ways than PLANNER's) is provided by the simple device of allowing ostensive definitions of states, i.e.
simply writing expressions like

$$S : \{\phi_1, \ldots, \phi_n\}$$

to mean that \(S\) is the state of belief in which just \(\phi_1, \ldots, \phi_n\), and all their deductive consequences, are true. (This is clearly quite safe since we could regard these state expressions as proposition letters and transcribe

$$S : \{\phi_1, \ldots, \phi_n\} \equiv (\phi_1 \& \ldots \& \phi_n)$$

and \(S \vdash \phi\) as \(S \supset \phi\)

Provided the underlying logic obeys the (strong) deduction theorem, this transcription preserves meanings.) To obtain the THNOT mechanism one has to introduce the notion of the present state, called 'NOW,' say. This is defined (ostensibly) to be at any given moment the state of belief of the system at that moment; it is thus a sort of metasyntactic variable. Every time a change is made to the state of belief, the meaning of 'NOW' changes, and all assertions in which 'NOW' occurs are altered so that 'NOW' is replaced by some newly generated constant term, say \(T\). Moreover, we add to the database the expression \(T : \{\phi_1, \ldots, \phi_m\}\), where \(\{\phi_1, \ldots, \phi_m\}\) constitute the set of beliefs which were current before the alteration. Apart
from this one extra-logical mechanism, the rules of inference we need are the following, where \( \Gamma \) denotes a set of assertions \( \phi \), and \( \vdash \) means derivability:

1. If \( \Gamma \vdash \phi \) then \( S : \Gamma \vdash \phi \)
2. If \( \Gamma \nvdash \phi \) then \( S : \Gamma \vdash \neg \phi \)

These clearly correspond to the THNOT rule of swapping success and failure, provided we imagine that THNOT always refers to NOW: but it enables considerably more things to be done, since "THNOT" assertions can be stored and take part in further inferences.
2.5 Computation as deduction.

In this section I briefly indicate, by an example, how one might approach conventional computation from the deductive point of view. This example is taken from (Hayes 1973) which discusses these issues more fully.

Consider the theory of the integers defined by the constant 0 and the successor function s. We can regard the predecessor function p as a Skolem function generated by the closure axiom

\[ \forall n. \, n=0 \lor \exists m. \, n=sm \]

i.e.

\[ n=0 \lor n=spn \]

Now, we can define various algorithms which compute the addition and subtraction functions on this basis:

- \( \text{addl}(n,m) = \text{if } n=0 \text{ then } m \text{ else } \text{addl}(pn,sm) \)
- \( \text{add2}(n,m) = \text{if } m=0 \text{ then } n \text{ else } \text{add2}(sn,pm) \)
- \( \text{add3}(n,m) = \text{if } n=0 \text{ then } m \text{ else } s(\text{add2}(pn,m)) \)
- \( \text{subl}(n,m) = \text{if } m=n \text{ then } 0 \text{ else } s(\text{subl}(n,sm)) \)
- \( \text{sub2}(n,m) = \text{if } m=0 \text{ then } n \text{ else } p(\text{sub2}(n,pm)) \)

All of these algorithms are different. If we evaluate addl, say, in the usual way, we get the following sequence, ignoring inferences of the form \( psn \vdash n \):
addl(ss0,ss0)
addl(s0,sss0)
addl(0,ssss0)

Now, consider the axioms

A1 \quad y + 0 = y
A2 \quad 0 + y = y
A3 \quad x + sy = sx + y
A4 \quad s(x + y) = sx + y

and the (paramodulation: Wos & Robinson, 1969) rule of inference:

\textit{from } t_1 = t_2 \textit{ and } A(t_3), \textit{ where } t_1 \textit{ and } t_3 \textit{ are unifiable terms with most general unifier } \mathcal{G}, \textit{ infer } A(t_2)\mathcal{G}, \textit{ and where } t_2 \textit{ and } t_3 \textit{ are unifiable with mgu } \mathcal{G} \textit{ infer } A(t_1)\mathcal{G}.

(If } A(t_3) \textit{ is } t_3 = t_4, \textit{ then this is basically the transitive law of equality, notice. } A \textit{ is called the assertion \textit{paramodulated into}.})

The evaluation sequence of addl can be mirrored exactly by a sequence of inferences from A2 and A3 using the above rule:
Notice that the variable \( y \) becomes fully instantiated to the result of the algorithm only in the last line of the proof.

These are precisely the inferences made by the algorithm add1: the algorithm is a recipe for generating proofs of the form above.

There are other proofs of the assertion, for example:

\[
\begin{align*}
ss0+ss0 &= y \\
\phi \\
0+ssss0 &= y \text{ by } A3 \\
\phi \\
y &\leftarrow ssss0 \\
\text{true} &\quad \text{by } A2
\end{align*}
\]

corresponding to add2. The search space of derivations defined by: (a) the axioms \( \{A2, A3\} \) (b) the initial assertion \( S^N0 + s^M0 = y \), (c) the above rule of inference, and (d) the set-of-support
restriction (only descendants of the initial assertion can be paramodulated into), contains evaluation sequences of both algorithms. We could say that $A_2 \& A_3$ is a 'nondeterministic algorithm' for computing the addition function, relative to an interpreter which is a TP system embodying the rule of inference and the set-of-support restriction. Notice that to supply such an interpreter with the axioms $\{A_2, A_3\}$ alone would not result in any inferences taking place (because of the restriction). To achieve activity one would have to insert an assertion $s^n_0 + s^m_0 = y$, which is therefore proper to regard as a procedural call.

If we take the same nondeterministic algorithm and supply it with a different call, a different proof results:

- $y + s^m_0 = s^n_0$
- $u + s^{m+1}_0 = s^n_0$ by $A_3$
- $v + s^{m+2}_0 = s^n_0$ by $A_3$
- $w + x^n_0 = s^n_0$ by $A_3$
- $w \leftarrow 0$
- $\text{true}$ by $A_2$
where by back-substitution we find \( y \leftarrow s^{n-m} \).

This is the algorithm subl. If we call by

\[ s^m + y = s^n \]

we obtain sub2, as the reader may easily check.

The same pair of axioms and theorem-proving evaluator will perform all four of these algorithms.

Now, the search space in each case contains more derivations than those corresponding to the conventional algorithm. By using A3 and A2 the 'wrong' way round, the TP system will investigate blind alleys of various kinds, many of which are infinitely long. A conventional evaluator of the algorithms does not conduct such a wasteful search but operates deterministically at each stage. The conventional algorithms embody tight control over the deductive process.

We can easily specify this control, however, by stating restrictions, which the interpreter must obey, upon the form of proofs, thus defining a refinement of the search space. Set-of-support is a refinement, and we will now incorporate it into the control assertions rather than have it fixed in the theorem-prover.
To specify this control I will define a very simple ad-hoc 'refinement language' (which is not to be taken too seriously). If A is an axiom, containing the variable u, let A:u mean that A must occur in proofs only as the topmost line, that all paramodulations must be into A or its descendants in the proof, and that the variable u must not be fully instantiated except in the last line of the proof. Let t₁=t₂:→ mean that in the application of the rule only the first case (t₁ replaces t₂) can be used on this axiom in a proof; and let t₁=t₂:← similarly mean only the second case. Let A: simply mean that the axiom A can be used anywhere in a proof. Unless something is said about an axiom it can't be used anywhere in a proof. These are all true/false assertions, expressing properties of proofs. In general we allow the name of an axiom to stand in its place in control assertions. This is enough to specify the deterministic algorithm above.

add₁ is (A₂:→) & (A₃:←), called by sⁿ₀+sᵐ₀=y:y
add₂ is (A₂:→) & (A₃:→), called similarly
add₃ is (A₂:→) & (A₄:←), called similarly.
sub₁ is add₁, called by sᵐ₀+y=sⁿ₀:y
sub₂ is add₁, called by y+sᵐ₀=sⁿ₀:y
(In evaluating sub₂, one finds assertions of the form sy=sⁿ₀ rather than y=psⁿ₀. This does
not bother the inference system, which is based upon unification rather than recursion. In general, p is unnecessary when the 'output variable' is explicitly involved in the evaluation process.) In each case these simple restrictions completely narrow down the search space so that only a single path is legal and hence only deterministic computations are possible. Note, however, that assertions like A2: \implies are not imperatives. They describe proofs. Their pragmatic role for the theorem-prover is to delimit its freedom, but they do not instruct it directly. In general, it is free to schedule its activity within the constraints as it chooses. The imperative: "do this" has been replaced by the 'permission': "do anything legal you like; but all the others are illegal".

Other algorithms are implicit here also: for example, calling add1 by: y+y=s^n: y gives a (deterministic) algorithm for dividing by two, which is non-terminating when n is odd. We can ask for it to choose, nondeterministically, two numbers greater than two which add to eight:

\[ s^3y + s^3x = s^8 \quad y : x \]
If we add axioms defining multiplication, and slightly enrich the control language, the possibilities multiply also:

\[ A_5 \quad 0 \times x = 0 \]
\[ A_6 \quad x \cdot y + y = s(x) \cdot y \]

If \( A_1 \) and \( A_2 \) are axioms, let \( A_1 \leftarrow A_2 \) mean that the theorem-prover, when faced with a choice between using \( A_1 \) and \( A_2 \) to extend the proof, is to examine the consequences using \( A_1 \) (under its control restraints, of course) to exhaustion before using \( A_2 \). (This is a restriction upon the search strategy of the theorem-prover rather then upon its refinement. It is directly analogous to the use of recommendations in PLANNER. It cannot be checked by examining derivations in isolation.) The usual multiplication algorithm of repeated addition is

\[ \text{addl} \& (A_5: \to) \& (A_6: \leftarrow) \& (A_6 \leftarrow A_3), \text{ called by} \]
\[ s^0 \cdot s^m_0 = y : y. \]

If we call this with \( s^m_0 \cdot y = s^p_0 : y \), we obtain a rather unusual and inefficient integer division algorithm. (Both of these algorithms are slightly non-deterministic, although correct and single-valued. They can be made deterministic.
and hence more efficient by more subtle control assertions which refer to the internal structure of $A_6$, but I will not go into details for lack of space. The more conventional division algorithm of repeated subtraction is $(A_1: \rightarrow) & (A_2: \rightarrow) & (A_3: \rightarrow) & (A_5: \rightarrow) & (A_6: \leftarrow) & (A_1 \leftarrow A_3) & (A_3 \leftarrow A_5) & (A_5 \leftarrow A_6)$, called by $y.s^m_0 = s^n_0:y$.

For example, below is the search space obtained from the last algorithm by dividing 2 into 4.

All the branchings are OR-branchings indicating choices open to the interpreter. Underlining denotes a dead end.

$$\begin{align*}
\text{test for } y \leftarrow 0 & \quad y.s_0 = sss_0: y \\
& \quad A_6 \quad \{ y \leftarrow s_u \}
\end{align*}$$

$$\begin{align*}
& \quad A_6 \quad \{ y \leftarrow s_u \}
& \quad u.s_0 + s_0 = sss_0
& \quad \phi A_3
& \quad s(u.s_0 + s_0) = sss_0
& \quad \phi A_3
& \quad ss(u.s_0 + s_0) = sss_0
& \quad \phi A_1
\end{align*}$$

$$\begin{align*}
& \quad u \leftarrow 0 \quad A_5
& \quad ss_0 = sss_0
\end{align*}$$

$$\begin{align*}
\text{test for } z \leftarrow 0 & \quad A_5
\end{align*}$$

Result is $y \leftarrow ss_0$

$$\begin{align*}
\text{test for } z \leftarrow 0 & \quad A_5
& \quad A_6
& \quad \text{true}
& \quad \text{etc.}
\end{align*}$$
I wish to suggest that a heuristic TP system which could accept control advice in a properly designed language would be most powerful in AI work. In order to design such a language, however, one would need to have a clear idea of the form of proofs, and the current TP literature contains a bewildering variety of ideas on this subject, with no clear winner. In the rest of this dissertation I will attempt to bring some coherence to this topic.
3. SEARCH SPACES, REFINEMENTS AND LIFTING

3.0 Introduction

In this chapter we return to the fold of conventional theorem-proving and cover the preliminaries necessary before discussing particular TP systems for first-order-logic. The emphasis will be upon comparing the kinds of search space with which TP systems are faced, rather than upon comparing individual systems.

Much of the material in this chapter can be found in essentials in the literature, but the approach here is more general than the others I have seen. The distinction between derivations and structures, the idea of structural redundancy and the discussion of locality in refinements, are new. Sections 3.1 - 3.4 are concerned with search spaces in general: the remaining section with search spaces arising in TP systems for PC, and is largely devoted to proving Robinson's (1966) lifting lemma in a general setting: this is also new.
3.1 More on search spaces

3.1.1 To identify or not to identify

As noted in section 1.5.2.1 above, the present account of search spaces follows the second version of theorem-proving graphs in identifying nodes with the same label. A node in a search space thus corresponds to a structure rather than, as in the first version of theorem-proving graph, to a derivation of a structure. Kowalski's motivation for this convention is quite different from mine, and so in this section I will defend it from my point of view.

The general methodology adopted here is that all operations, restrictions etc., which are used by the TP system are defined upon structures rather than their derivations, and their interrelationships in the search space are purely theoretical entities. The TP system needs to store only the structures themselves in order to proceed with its proof-generating activity.

It is possible to take this position because we are free to make structures as rich as we please. In the limit, indeed, the structures can be made isomorphic to their own derivations. (cf. derivation...
spaces, section 3.3 below). In this case, of course, we are in a similar position to the first theorem-proving graphs in that nodes of the search space would correspond inter alia to derivations. On the other hand, by simplifying structures we can blur distinctions and cause the search space to collapse into a smaller space. In general, given a collection of constructive operations and refinements, (cf. 3.1.2 below) we can ask the question: how rich must we make the structures in order that the operations and refinements can be defined? Thus a new dimension - that of essential complexity of structure - along which to classify TP systems is made available.

If we insist, as is conventional, that structures are statements and derivations are proofs, this is not possible.

All of this will be discussed in more detail later in this chapter. For now, consider two examples.

(1) The set-of support refinement of resolution can be defined on a space whose structures are pairs consisting of a clause and a bit:

\(<C, b>\), where \(b = 1\) if \(C\) is in the set of support and \(0\) otherwise. The constructive operation is \(R(<C_1b_1>, <C_2b_2>, <C,1>)\)

where \(C\) is a resolvent of \(C_1\) and \(C_2\) and \(b_1 \lor b_2 = 1\).
(2) Consider the computation-as-deduction example of section 2.5 above. Notice that in the search space for evaluating addl(x,y), all constructive operations and refinements are defined only on the last node of the derivation. Thus, structures need only be single assertions of the form \( s^m_0 + s^n_0 = y \). Taken together with the fact that derivations are linear, this means that the evaluator needs storage only sufficient to store a single assertion, and this storage can be used over again at each step of the proof generation process. The net effect of such re-use is to convert the recursive addl into the iterative algorithm

\[
\text{add3}(x,y) = L: \text{if } x=0 \text{ then } y \text{ else } (x:=px; y:=sy; \text{goto } L)
\]

in which the assignments indicate that the storage is to be re-used, and the jump (as opposed to a recursive call) indicates that only the most recent node of the derivation is needed. (In computing terminology: don't bother with the stack.) This example suggests how considerations of localisation and essential complexity might enable a theorem-proving interpreter to optimise its programs.
3.1.2 On keeping searches simple

This methodology extends also to search processes. Here, search spaces are purely disjunctive. The system is at any moment faced with a number of alternative ways of proceeding. There is no 'and/or' structure, no bi-directional search, etc. Kowalski's (1972) motivation for the second form of theorem-proving graph is his desire to consider bi-directional search strategies in the resulting space. Here, we would regard this as a conventional disjunctive search in a space whose structures are pairs \( \langle s_1, s_2 \rangle \), where \( s_1 \) is a structure from the 'forward' search space and \( s_2 \) a structure from the 'backward' search space. This paired search space inherits constructive operations from the two component search spaces in an obvious way, and also has constructive operations which replace pairs \( \langle s_1, s_2 \rangle \) in which \( s_1 \) and \( s_2 \) "meet", by some terminal structure or perhaps (depending on the details of the space required) by a pair \( \langle s_1', \text{empty} \rangle \), for example, where \( s_1' \) is the result of absorbing \( s_2 \) into \( s_1 \).

Similarly, a 'GPS-type' search would here be regarded as a conventional disjunctive search in
the space whose structures were goal types in the original space. I will not elaborate these examples in detail for lack of space, but hopefully the basic idea is clear.

There are several advantages to this way of approaching search. The first is simplicity and clarity. The difficulties, severe already, of comparing one search process with another are greatly compounded by a proliferation of different search paradigms. Secondly, this approach makes clear just what operations and structures are involved in the search space. More elaborate ideas sometimes lead to confusion on these points. 'Bi-directional' search for example, involves checking at each stage whether the two directions of search have met. The difficulty of performing this check increases with increased depth of search, at a rate which is quadratic compared with the branching rate of the component spaces. It is easy to forget this uncomfortable fact, and dismiss the problem with vague talk of hash-coding methods. Describing the space disjunctively forces one to face the issue since these "meeting" checks have to be put into explicit structural operations. There are similar pitfalls in describing GPS-type search spaces. Thirdly, this way of describing search gives a clear appreciation of the true branching rate of the spaces involved.
The chief advantage, however, is that this approach seems to get directly to the heart of the idea of search. Search arises because there are various possibilities and no way of knowing which will succeed. So the system has to try them all in some systematic way until one wins. The search strategy controls the order in which the system tackles the possibilities. Some possibilities give rise to others, that's all.

The idea of "and/or search spaces" has arisen because structures in search spaces often have a natural 'and' character; that is, they consist of substructures which can be considered more or less in isolation; and the structure as a whole leads to a solution just in case all its substructures do. Some constructive operations apply to substructures in isolation, the rest of the structure being carried along more or less unchanged during the application of the operation. It is perhaps natural to describe the resulting space in an and/or fashion; but it is not necessary to do so. Any and/or space can be regarded as a 'disjunctive' search space containing 'conjunctive' structures.
The semantic trees described in Chapter 4 are conjunctive in this sense, and several advantages accrue from regarding them as structures in a conventional search space.
3.2 Normal forms, refinements, and complexity measures

TP systems are searching for proofs. In a logical language there will usually be several different proofs of a given statement. Two different proofs may not, however, be essentially different. They may differ only in some trivial way, for example, in the order in which certain independent subgoals are tackled. Or, one may contain certain irrelevant steps which are eliminated in the other. Almost any reasonably complicated proof can be made arbitrarily more complicated by inserting minor irrelevancies, for example by decomposing an expression into its parts and promptly rebuilding it in a sequent calculus (Prawitz 1971), or by re­deriving unit clauses over and again unnecessarily in non-minimal resolution (Hayes & Kowalski 1971, page 44: the theorem is Kowalski's).

In a search context it is clearly desirable to avoid generating structures corresponding to several proofs which are essentially the same. What is wanted is to select, from each equivalence class of essentially-the-same proofs, a single representative which contains no irrelevancies. Such a representative will be called a normal proof. To make this concept
precise, we would, following Prawitz (1971), define a set of transformations on proofs which yield a normal proof when applied to exhaustion. However, for search purposes it is sufficient that we be able to effectively recognise non-normal partial proofs early in their construction and have the TP system avoid their further generation. (Ideally, it would only generate normal proofs in the first place; but this may not always be possible.) The effect of the restriction to normal proofs is that the system will use refinement mapping to restrict the search space. For, as soon as a partial proof is generated, by the application of a constructive operation, which is detected to be not normal, it is removed from further consideration. Notice that this assumes that every partial proof of a normal proof is normal: we now make this assumption explicit.

In order for it to be able to detect normal forms, a TP system must use structures which are sufficiently complex to enable the relevant distinctions between one partial proof and another to be made explicit. On the other hand, a structure's possessing complexity which is not made use of by the TP system in some way, (either in the application
of structural rules or in the application of refinement mappings) is wasteful and irrelevant (cf. section 3.3 below). Thus there is a need to establish a balance between normal form results and the richness of the structures used in a TP system based upon those results.

We note in passing that normal form results can be used in two rather different ways. The first, discussed above, is to eliminate redundant non-normal proofs in favour of their normal forms. For any reasonable notion of normal form, this is almost certainly a good idea. The second is to so control the TP system's activities that it tries to find only proofs of a particular sort of normal form. This often cannot be managed by tests of non-normality, but typically requires redesigning the TP system from the ground up. For example, in TP systems for the PC with equality, the difference between 'equality-elimination' systems like paramodulation, and 'equality-introduction' systems like Sibert's (1969), cannot obviously be phrased in terms of different refinements of the same underlying space. Rather it seems that the two approaches are fundamentally different. However, if one can phrase such a use in terms of restrictions upon the form of legal proofs,
then, again, the effect upon the search space is the application of a refinement mapping.

Let us define a refinement test to mean a predicate $K$ on structures. We will say that $K$ admits a structure $S$ if $K(S)$, otherwise that $K$ rejects $S$. Clearly, any refinement test defines a refinement mapping $k: k(S) = \{ S \in \mathcal{S} : K(S) \}$. We will use the word 'refinement' ambiguously to refer to either. Notice, however that refinement tests are defined on structures.

Applying a refinement to a search space cuts down the branching rate and simplifies the search problem. However, it may not be an unmixed blessing, as Kowalski (1971) points out. For if the refinement eliminates certain proofs which are simple to generate and only leaves proofs which are complex to generate, the TP system may be worse off with the refinement than without it. We therefore introduce the notion of a complexity measure: a function $m$ from derivations to some ordered comparison space, which will usually be the integers. We assume that it is monotonic: if $D_1$ is a subderivation of $D_2$ then $m(D_1) \leq m(D_2)$. 
Notice that complexity measures are defined on derivations rather than structures. The idea is that a complexity measure should reflect the effort which the TP system has to put into constructing that particular derivation. This is rather difficult to quantify precisely, but we will as a first approximation often use the size of the derivation, defined to be the number of output nodes of polyedges occurring in the derivation. For example, the size of the following derivation is six.

Now, if a refinement admits proofs with derivations which are not too complex, and yet cuts down on the branching rate, it seems clear that it is advantageous. Unfortunately to make this intuition into a rigorous proof of increasing efficiency seems to require
rather strong and unrealistic assumptions about the search strategies used (viz. that they generate all less complex before any more complex derivations). As a heuristic principle, however, we will accept reduced branching rate without consequent increase in derivation complexity as a desirable property of a refinement. As remarked above, I am interested more in comparing search spaces than in complete uniform TP systems. The improvement in a search space represented by a refinement of this sort would be relevant to a flexible heuristic TP system, for example, a programming language interpreter, as well as to a classical TP system; even though, as argued in section 1.2, notions of relative efficiency do not apply.

As a general rule, refinements associated with the restriction to normal forms will tend to have this desirable property, or at least to come close to it. For, a reasonable notion of the normal form of a proof would entail that the derivation of the normal form should not be more complex than the derivation of the original proof. The removal of irrelevancies, for example, would not complicate
the derivation and should simplify it. However, refinements associated with the selection of a particular normal form may well involve the elimination of simple derivations, and will be treated therefore with more caution.
3.3 Simple and structural redundancy

We will say that a search space is (simply) redundant (at a node N) if there is more than one derivation of N in the space. (The word 'redundant' is taken from (Kowalski 1971). The concept was first discussed, in the context of resolution, by Wos and Robinson.)

Clearly, redundancy is an unfortunate property for a search space to have. It can only be wasteful to re-derive the same structure over again. Interest attaches, therefore, to procedures for removing redundancy, by restructuring the search space or by imposing suitable refinements to eliminate some alternative derivations early on.

Unfortunately, simple redundancy is too weak an idea to capture the intuitive idea of wasteful duplication underlying redundancy. For it is easy to totally eliminate simple redundancy by enriching the structures labelling nodes of the search space.

Let $\mathcal{G}$ be a search space, obtained from the complete space from $\mathcal{S}$ relative to $\mathcal{R}$, by applying the refinement mapping $k$. Let $A$ be the set of
derivations in $\mathcal{G}$. For each $D \in A$ let $c(D)$ be the label of the conclusion of $D$. Now for each $n$-ary $R \in \mathcal{R}$, define an $n$-ary constructive operation $R'$ on $A$ by:

$$R'(D_1, \ldots, D_n) \equiv R(c(D_1), \ldots, c(D_n)).$$

Let $\mathcal{R}'$ be the set $\{R': R \in \mathcal{R}\}$. Define a refinement mapping $k'$ on subsets of $A$ by:

$$\forall B \subseteq A, \ k'(B) = \{D: c(D) \in k(\{c(D'): D' \in B\})\}.$$ 

Let $\mathcal{G}'$ be the refined space under the map $k'$ of the complete search space relative to $\mathcal{R}'$ from the set $\mathcal{B}'$ of initial structures which are the premis nodes of $\mathcal{G}$. $\mathcal{G}'$ will be called the derivation space of $\mathcal{G}$.

Clearly, $\mathcal{G}$ and $\mathcal{G}'$ are closely similar. There is a 1:1 correspondence between derivations in $\mathcal{G}$ and in $\mathcal{G}'$ which preserves subderivations. Any search strategy for $\mathcal{G}'$ induces a corresponding one for $\mathcal{G}$. $\mathcal{G}$ and $\mathcal{G}'$ have the same branching rate. If we identify nodes in $\mathcal{G}'$ which have labels with the same conclusions (in $\mathcal{G}$), we obtain a space isomorphic to $\mathcal{G}$: $\mathcal{G}$ is a homomorph of $\mathcal{G}'$, as explained below.

The conclusion of every derivation in $\mathcal{G}'$ is labelled by the corresponding derivation in $\mathcal{G}$. There is therefore a 1:1 correspondence between the derivations in $\mathcal{G}'$ and the labels of their conclusions:
That is, $G'$ has no simple redundancy. If $G$ has no simple redundancy, then $G'$ is isomorphic to $G$.

This elementary trick eliminates simple redundancy, but clearly does not give us a better search space. Most of the structure in the labels of nodes in $G'$ is irrelevant to the operations defining $G'$. We could throw it away without thereby eliminating any derivations. A search space whose labels contain irrelevant information like this, which is ignored by all the operations defining the search space, will be called structurally redundant. We can make this concept precise by considering homomorphisms of search spaces.

Let $<G, s>$ be a search space and $\sim$ an equivalence relation on nodes of $G$. Let $E_1$ and $E_2$ be polyedges in $G$ such that $b(E_1) = b(E_2)$, such that there is a 1:1 correspondence $\alpha$ between $i(E_1)$ and $i(E_2)$ with $N_1 \alpha N_2 \Rightarrow N_1 \Leftrightarrow N_2$. If, for every such pair of polyedges, there is a 1:1 correspondence $\beta$ between $o(E_1)$ and $o(E_2)$ with $N_3 \beta N_4 \Rightarrow N_3 \Leftrightarrow N_4$, then $\Leftrightarrow$ will be called a congruence on $G$. A congruence on $\sim$ thus induces an equivalence relation upon polyedges in $G$ which preserves inputs and outputs and respects
body labelings. Thus we can define a graph $G'$ whose input and output nodes are equivalence classes of nodes of $G$. The polyedges of $G'$ are of the form 

$$<[i(E)], b(E), [o(E)]>$$

for some polyedge $E$ of $G$ (where $[s]$ denotes the set or vector of equivalence classes of members of $s$, in the obvious way). The polyedges $<X, Y, [o(E)]>$ and $<[i(E)], 2, U>$ in $G'$ are connected in $G'$ just in case there are $M \in o(E)$ and $N \in i(E)$ with $M \Leftrightarrow N$. $G'$ will be called a quotient graph of $G$. For example:

**Congruence:**
- $C \Leftrightarrow C'$
- $D \Leftrightarrow D'$
- $F \Leftrightarrow F' \Leftrightarrow F''$
- $G \Leftrightarrow G'$
- $H \Leftrightarrow H'$

**Quotient graph**
The point of these definitions is that if some congruence identifies two nodes then these nodes must behave indistinguishably in the search space. Thus whenever nodes have labels which differ only in some inessential structure which is ignored by all the operations defining the search space, then identifying those nodes in an equivalence relation will yield a congruence.

Let \( \langle Q, S \rangle \) be the search space obtained by applying the refinement test \( K \) to the complete space from \( \mathcal{R}_0 \) relative to \( \mathcal{R} \), and let \( \mathcal{R} = \{ N_s : N \in \mathcal{G}_s \} \). Let \( \leftrightarrow \) be an equivalence relation on \( \mathcal{R} \) such that

1. For every \( R \in \mathcal{R} \), and every \( S_1, \ldots, S_n' \in \mathcal{R} \),
   \[
   S_1 \leftrightarrow S_1' \land \ldots \land S_n \leftrightarrow S_n' \Rightarrow R(S_1, \ldots, S_n) = R(S_1', \ldots, S_n').
   \]
2. For every \( S, S' \in \mathcal{R} \),
   \[
   S \leftrightarrow S' \Rightarrow K(S) \equiv K(S').
   \]

Then the induced equivalence relation \( \leftrightarrow \) on \( \mathcal{G} \) defined by

\[
N_1 \leftrightarrow N_2 \equiv N_1 s \leftrightarrow N_2 s
\]

is a congruence. This follows immediately from conditions (d) and (e) in the definition of search space.
We can define structural operations $R'$ for each $R \in \mathcal{R}$, and a refinement test $K'$ upon equivalence classes of structures under $\equiv$ in the obvious way:

$$R'(\llbracket s_1 \rrbracket, \ldots, \llbracket s_n \rrbracket) \equiv R(s_1, \ldots, s_n)$$

$$K'(\llbracket s \rrbracket) \equiv K(s).$$

These are clearly well-defined, by (i) and (ii) above.

Now let $\tilde{G}'$ be the quotient graph of $\sim$ relative to $\equiv$, and define a labelling function $t$ on $\tilde{G}'$ by

$$\llbracket N \rrbracket t = \llbracket N s \rrbracket$$

for input & output nodes,

$$N t = (N s)'$$

for body nodes.

Clearly this is well-defined.

Lemma $<\tilde{G}', t>$ is the search space obtained by applying the refinement mapping $K'$ to the complete space from $\{\llbracket S \rrbracket : S \in \mathcal{S}_o\}$ relative to

$$\{R' : R \in \mathcal{R}\} = \mathcal{R}'.$$

Proof Conditions (a)-(c) in the definition of search space are trivial. Condition (e) follows immediately from the definition of quotient graph, since $\mathcal{R}$ and $\mathcal{R}'$ are in 1:1 correspondence. To see that (d) holds, let $<\llbracket i(E) \rrbracket, b(E), \llbracket o(E) \rrbracket>$ be a polyedge in $\tilde{G}'$. Then we have
\[ [i(E)] t = [i(E)s] \text{ and } [o(E)] t = [o(E)s] \]

But

\[ o(E)s \subseteq b(E)s \text{ (i(E)s)} \] since \( \langle \mathcal{G}, s \rangle \) is a search space.

and hence

\[ [o(E)s] \subseteq b(E)s, \quad \left[ {i(E)s} \right] \text{ by definition of } R'. \]

\[ = b(E)t \left[ {i(E)t} \right] \text{ by definition of } t \]

which establishes (d).

An exactly similar argument shows that every polyedge in \( G' \) contains output nodes labelled with every generable structure admitted by \( K' \).

QED.

We will call the space \( \langle G', t \rangle \) the collapse of \( \langle \mathcal{G}, s \rangle \) generated by \( \leftrightarrow \). In fact we will loosely refer to any space isomorphic to a collapse, as a collapse. In particular we will allow a collapse to be labelled by more compact representations of equivalence classes. In applications, collapses arise through structures being unnecessarily complex: and it will usually be easy to find a less complex structure to use as a label for a node in the collapse.
space. For example, any search space is the collapse of its derivation space under the equivalence which relates all derivations with the same conclusion.

This important fact dramatically illustrates the relationship between structural and simple redundancy. In general, any structurally redundant space has a collapse in which structural redundancy has been replaced by simple redundancy.

Structural redundancy is more pernicious than simple redundancy, for obvious reasons. A programme of improving search spaces by eliminating simple redundancy must therefore avoid introducing structural redundancy. Simple redundancy can be eliminated either by the use of refinements which cut out some alternative derivations, or by enriching the structures in the space. To avoid structural redundancy, these two methods must go hand in hand. There are limits on how far useful refinements can be defined without enriching the structures (remember, refinement tests are defined on structures); contrariwise, any enrichment which is not made use of in some refinement test or structural operation only leads to structural redundancy.
Clearly, the ideal situation would be a monogenic search space, isomorphic to its own derivation space yet with no structural redundancy. Suppose we tried to set this up. Since this space is to have no simple redundancy, we may temporarily identify structures with their derivations. We require a collection then, of structural operations and refinement test defined on derivations, so that there is no structural redundancy. That is, no part of the derivations must be irrelevant to the operations and tests. If our operations and test ignore some aspect of the form of the derivations: if they can be regarded as in any sense locally defined on parts of the derivations, then we could collapse the space by replacing derivations by simpler structures just complex enough to support the operations and test. It does not necessarily follow that the space is structurally redundant, for the irrelevant structure may occur only once in the space. But it is extremely likely, a priori, and borne out in practice.

In most inference systems, inference rules (which are structural operations when, as here, structures are identified with derivations) are
defined very locally indeed; typically on the conclusion of the derivation alone. Resolution is such an inference rule, for example, as are all the rules in a conventional Gentzen sequent calculus, except the \( \land \)-introduction and \( \lor \)-introduction rules; and these depend only upon very weak properties of the derivations. In both cases there is enormous structural redundancy when we identify structures with derivations, and enormous simple redundancy, therefore, when we move to a collapse. In the resolution case, many attempts have been made to eliminate redundancy by introducing new refinement tests. But many refinements in the literature are again very local. Set-of-support for example, can be collapsed to structures consisting of the conclusion of the derivation paired with a single bit, as already noted. One of the most globally defined resolution refinements known to me is the minimality refinement: a minimal derivation must not have literals with the same atom resolved upon/in any branch. And even this is branch-local: a derivation can be replaced by the set of its branches. This loss of structure is reflected in the very considerable structural redundancy present in the minimal-derivation search. (For example, from certain, admittedly
artificial, sets of ground clauses containing $n$ distinct atoms, there are $\frac{n-1}{i=0} (n-i)^2$ distinct minimal derivations of the null clause. This number increases explosively as a function of $n$. See section 4.4.3 below.

Non-locally defined refinements and structural operations do not of course guarantee decreased redundancy. But they are necessary prerequisites. We can thus realistically propose a programme for improving search spaces by finding structural operations and/or refinement tests which are more global in their definition.

But this programme is faced at the outset with a severe difficulty. If operations are to be defined upon derivations, we must have some way of referring to derivations and describing their properties in some effective way. There are infinitely many derivations from most nontrivial input sets. We seem therefore, to need ways of describing infinitely many possible derivations in a finite effective vocabulary. Moreover, the primitives of our derivation-describing language had better be fairly
locally testable, for otherwise the structural operations and refinement predicates will be hard to compute at the implementation level (cf. the discussion in section 1.5.3). Thus it seems inevitable that our descriptions of derivations will necessarily ignore much of the derivation's structure in order to be programmable.

This, although vaguely outlined here, is I believe a fundamental dilemma of theorem-proving. In order to be programmable, a TP system's search-space-defining operations must admit infinite structural redundancy. (I believe that this intuition can be hardened into a formal result on the limitations of TP systems: but this result will not be found in this dissertation, unfortunately. Like the Ehrenfeucht-Rabin theorem, it applies equally to heuristic as to uniform TP systems.)

Our programme for removing structural redundancy can be given life in two ways. Firstly, if we can find strong normal form theorems for proofs (cf. section 3.2), then perhaps some ad hoc derivation descriptions based on the normal forms can be used to introduce globality without sacrificing programmability. A beautiful example of this is
provided by the lambda-calculus and the Church-Rosser theorem. This is the ultimate normal form theorem: given any statement which has a proof, there is a single normal form for the proof: and that normal form can be uniquely characterised by local tests on the derivation. Thus all redundancy can be completely eliminated. It is this fact which makes efficient deterministic lambda-calculus interpreters such as LISP, possible. But the lambda-calculus is a very much simpler language than PC (for example, it is not additive), and there seems little hope of anything like the same sort of result in PC. The second way of introducing globality is to deliberately enrich derivations by adding artificial extra structure which can be used by the operations and tests to distinguish one derivation from another, but which we can arrange to not increase the branching rate. This is often informally described as using a plan. Various ad-hoc ideas for organising plans of this sort can be found in the AI literature. For example, much of the control information in a PLANNER program can be interpreted as such extra information added to derivations and used by the TP system to restrict the search space (However, almost all of PLANNER's
control is defined locally on small parts of derivations. There would seem to be great promise in finding means of conveying more global plans to a TP system.). Another example from conventional theorem-proving is provided by the various ordering refinements which use artificial orderings to cut down redundancy (cf. chapter 4 below).
3.4 Pseudostructures, compulsory operations and admissibility tests.

In this section I discuss a slight technical complication in the description of search spaces which, although trivial, causes immense difficulties if ignored.

As the reader may have noticed, the concept of refinement test is not formally necessary. Any refinement test \( K \) can be incorporated into the constructive operations defining the search space. For every \( R \), we define a new \( R' \) by

\[
R'(S_1, \ldots, S_n) \equiv R(S_1, \ldots, S_n) \& K(S_n)
\]

However, refinement tests are an intuitively satisfactory way of defining many search spaces, having close links with implementation methods and enabling comparisons between search spaces to be made more easily. Moreover, it is often simply easier and more direct to describe a given search space using a separate refinement mapping.

Now it sometimes happens that the constructive operations defining a search space can themselves be naturally divided into two parts, just as \( R' \) could
be divided into R and K. And often in such cases it is similarly easier and more direct to describe the search space in terms of the two separate suboperations. That is, instead of writing
\[ R(s_1, \ldots, s_n) \]
we would have
\[ R_1(s_1, \ldots, s^O_n) \land R_2(s^O_n, s_n) \]
where R is decomposed into R_1 and R_2. S^O_n acts as an intermediary between these two suboperations.

Now, S^O_n may not be a legal structure in the search space. For example, the structural operations and refinement test may not be defined on S^O_n. Moreover, in describing the search space, the derivation of S_n from S_1, \ldots, S_{n-1} should be a single polyedge: we do not have

\[ S_1 \circ \ldots \circ S_{n-1} \]

in the search space. An entity such as S^O_n will be called a pseudostructure.

This is all fairly straightforward: a
pseudostructure is a convenient technical device for defining structural operations, no more.

In the literature (notably in (Loveland 1969), followed by (Kowalski & Kuehner 1971)), however, this course has not been taken. Rather, pseudostructures have been considered to be structures in the search space in their own right. In order to prevent the search space from branching artificially at nodes labelled with pseudostructures, we must somehow ensure that only the 'second halves' of operations are applied to pseudostructures. We must therefore have some way of distinguishing pseudostructures from proper structures. The standard terminology for this, introduced by Loveland, is to refer to 'proper' structures as admissible and pseudostructures as preadmissible. This assumes that there is only one class of pseudostructures; we will make this assumption from now on for simplicity, although both methods below could clearly, with some complication, be extended to the general case of several operations being broken up to yield several classes of pseudostructure.

Armed with this distinction we can proceed in one of two ways. Firstly, we can boldly declare
that the only operation which can be applied to a preadmissible structure is \( R_2 \), and that \( R_2 \) must be applied to a preadmissible structure 'immediately'. This seems intuitively clear and has the merit of simplicity. But in order to make it precise we have to completely change the definition of search space. Moreover, such compulsory operations are very awkward to handle theoretically: for example, the lifted operation (cf. section 3.5.3 below) of a compulsory operation need not be compulsory but also cannot be regarded as 'free'.

The second method uses a refinement test to control the branching rate. This is more in line with the general theory but perhaps somewhat less intuitive. Suppose we enrich the structures, if necessary, to include a note of whether any structure immediately earlier in the derivation (i.e. in the same polyedge as the conclusion) is permissible. Then we can define a refinement test \( K(S) \) to be:

- \( S \) is not preadmissible with a preadmissible immediate ancestor. Applying this refinement test cuts out derivations containing a preadmissible structure which is not 'immediately' made admissible by applying \( R_2 \). We could call such a structure
inadmissible, following Loveland.

It is sometimes convenient to decompose $R$ into an initial part $R_1$ followed by some sequence of applications of an $R_2$, the idea being that one goes on applying $R_2$ as long as one can. When the pseudostructure (preadmissible structure) resists the application of $R_2$, it is a structure (an admissible structure).

We thus have

$$R(S_1, \ldots, S_n) \equiv R_1(S_1, \ldots, S_{n-1}, S_n) \& R_2(S_{n-1} S_n) \& \ldots \& R_2(S_{n-1} S_n)$$

for some $k$, where the $S_i$ are pseudostructures.

If $k=0$, then $S_n = S_n$. The sequence $<S_1, \ldots, S_n>$ of pseudostructures will be called a reduction sequence for $R_2$, and we will write $R_2^*(S_n, S_n)$.

We will write $R = R_1 \otimes R_2$ to indicate this mode of composition. This case can also be handled by a refinement which is a slight generalisation of $K$; this time the refinement insists that no preadmissible structure occurs in a polyedge whose body node is labelled with any operation other than $N_2$.

Given a search space relative to $R_1$ and $R_2$
and this refinement, we can regard it as being built from larger steps relative to R. But it can also be rebuilt differently. Suppose we define a structural operation $R'$ on preadmissible structures as follows:

$$R'(S_1, \ldots, S_n) \equiv R_1(S_1', \ldots, S_n') \land \forall i, 1 \leq i \leq n, R_2(S_i, S_i').$$

so that, in $R'$, the applications of $R_2$ precede those of $R_1$. Clearly, any search space relative to $R_1$ and $R_2$ can be regarded as built from $R'$, possibly with an extra $R_2^*$ step at the end of finite derivations:
In the R' derivation the roles of preadmissible and admissible structures are reversed: admissible structures serve only as intermediaries in the construction of preadmissible structures.

Which of these methods of describing the search space is adopted, is largely a matter of taste. In some circumstances the second mode of composition is preferable, however. We will use the notation $R_1 \otimes R_2$ to refer ambiguously to either R or R'.

This language of admissibility thus enables the original search space to be regarded as a refinement of a more complete space of a conventional sort, in which $R_1$ and $R_2$ are separate operations.
3.5 Propositional properties and the lifting lemma

In this section for the first time we consider search spaces for a particular input language: first-order predicate logic (PC). From now on throughout this section, the input language of the TP system under consideration will be assumed to be PC.

As remarked in section 1.5.1, structures must contain all the nonlogical symbols involved in proofs. In PC, nonlogical symbols all occur in atoms of the form \( P(t_1, \ldots, t_n) \), where \( t_i \) is a term which is a variable or a constant, or has the form \( f(t_1, \ldots, t_m) \) for some \( m \)-ary function symbol \( f \) and terms \( t_i \). We will now make the important assumption that structures contain atoms, and that all nonlogical symbols occur within atoms. We assume that the set of all atoms considered by a TP system is meaningful, and denote it by \( \mathcal{A} \). The set of atoms occurring in \( S \) will be denoted by \( \| S \| \).

Every TP system for PC in the literature obeys this natural rule. It is important because
without it there would be no way of relating structures for PC to structures for the propositional calculus. Getting this relationship clear is vital to the design of reasonable search spaces for PC.

Let us first consider the choice ranges with which the TP system is faced. PC formulas are built from atoms by applying propositional connectives such as negation, conjunction, disjunction and implication, and by binding free variables by quantification. We can reduce this complexity in several standard ways. Specifically, we will assume that input expressions are built from literals (atoms or negated atoms) by the application of conjunction, disjunction and universal quantification. It is well known that for any PC expression \( \phi \) another expression \( \psi \) in this form can be found such that \( \phi \) is unsatisfiable iff \( \psi \) is, and \( \phi \) and \( \psi \) have the same number of occurrences of atoms. (\( \psi \) is got by replacing \( \phi \rightarrow \psi \) by \( \neg \phi \lor \psi \), moving \( \neg \) inwards by DeMorgan's laws, and replacing existential quantified variables by Skolem functions,
in that order. None of these operations change the number of occurrences of atoms. In chapter 4 we will further assume that the formula has been put into conjunctive normal form, but that is not yet necessary.)

Thus we have three choice ranges: components of conjunctions, components of disjunctions, and instances of universally quantified formulae. (I am here slurring over a very important issue, for it is by no means obvious that these normal forms for expressions are reflected in normal forms for proofs. In fact, however, they are. This fact has been called the "fundamental theorem of logic". It follows from the completeness results proved in Chapter 4.) Of these, the last is by far the most difficult to search. For, the number of possible instantiations of a formula with variables is usually infinite.

Fortunately, however, the work of Prawitz (1960) and J.A. Robinson (1965) provides a way of greatly reducing the number of instances which have to be considered. In this section this theory is expounded in a general setting.
As noted in section 3.4, refinement mappings can be considered as being incorporated into structural operations. We will temporarily adopt this useful convention, in order to simplify the discussion.

3.5.1 Propositional relations and substitutions

Some structural operations are essentially propositional in nature: they take no notice of the internal structure of atoms, but only of their interrelationships. They treat atoms as atomic propositions. Others are essentially to do with the internal structures of atoms, typically involving substituting terms for variables. We can make this important distinction precise by considering mappings between atoms.

Let $f$ be a map on the set $A$. If $S$ is a structure let $Sf$ denote the result of replacing every occurrence of every atom $A$ in $S$ by $f(A)$, and let $\mathcal{S}f = \{Sf: S \in \mathcal{S}\}$, etc. in the usual way. We assume that this operation is well-defined and that if $f$ is 1:1 on $A$,
then the map \( S \to S_f \) is 1:1 on \( \mathcal{S} \), where it is defined. \( S_f \) may not be a legal structure. The assumption cannot be proved since 'structure' is undefined, but it seems a very reasonable requirement. We will take it as a new axiom contributing to the meaning of 'structure'.

If \( R \) is a structural operation then we will say that \( R \) is propositional if
\[
R(S_1, \ldots, S_n) \equiv R(S_1 f, \ldots, S_n f) \text{ for every 1:1 map } f \text{ on } \mathcal{S}.
\]
An example of a non-propositional structural operation is provided by resolution between clauses, while ground resolution is an example of a propositional one.

For propositional operations we extend the definition of sensible (section 1.5.4) by treating atoms as nonlogical symbols. That is, we call a propositional operation nonsensible, if, for some \( S_1, \ldots, S_{n+1} \), \( R(S_1, \ldots, S_{n+1}) \) holds and either for some partition \( \alpha \) of \( \{1, \ldots, n\} \) onto \( \{1, \ldots, m\} \)
\[
\bigcap_{i} \left( \bigcup_{\alpha(j)=i} S_j \right) = \emptyset,
\]
\( \alpha(1)=1 \)
and sensible if it is not nonsensible. This is in accord with the intuitive idea of treating atoms as unanalysed wholes. Any propositional operation which does not obey this condition is equivalent to one which is nonsensible by the earlier definition. (To see this, say for example \[ \| S_{n+1} \| \not\subseteq \bigcup_{i=1}^{n} \| S_i \| \]: let \( A \in \| S_{n+1} \| \) but \( A \notin \bigcup_{i=1}^{n} \| S_i \| \). Then by mapping \( A \) into some atom \( B \) with a disjoint vocabulary, we can obtain an \( S'_{n+1} \) with \[ \| S'_{n+1} \| \not\subseteq \bigcup_{i=1}^{n} \| S_i \| \).)

We now need the relevant terminology to discuss instances and unifiers. Most of this is well known, although the present treatment is somewhat nonstandard.

A substitution \( \Theta \) is a function from variables to terms (such that either \( x=\Theta(x) \) or \( x \notin \Theta(x) \)). We will write \( x\Theta \) for \( \Theta(x) \). The constant
function on variables is called the \textit{null} substitution and is written $\mathcal{E}$. The substitution which maps $x_1$ onto $t_1$ and all other variables onto themselves will be written $(x_1/t_1, \ldots, x_n/t_n)$.

If $\Theta$ and $\mu$ are substitutions then $\mu \Theta$ is $\Theta \circ \mu$, so that $S(\mu \Theta) = (S \mu) \Theta$. Composition is associative by definition. If $t(A,S)$ is a term (atom, structure) then $t(\Theta(A, S))$ denotes the result of replacing $x$ by $x\Theta$ throughout $t(A, S)$. Thus a substitution defines a map in $A$ and hence in $S$. $S\Theta$ is an \textit{instance} of $S$. For every $S$, $S\Theta \Theta = S\Theta$.

If $S_1 \Theta = S_2$ and $S_2 \mu = S_1$ then $S_1$ and $S_2$ are \textit{variants}, and differ only in variable names. We will often treat variants as indistinguishable. If $S \Theta$ is a variant of $S$ then $\Theta$ is a \textit{variance substitution}.

Let $P$ be a predicate on substitutions. If there is a substitution $\sigma$ such that $P(\sigma)$ and for all $\Theta$, if $P(\Theta)$ then $\Theta = \sigma \lambda$ for some $\lambda$, then $P$ is \textit{pointed} and $\sigma$ is a \textit{point} of $P$. Clearly, points are unique to within variance, so we will often refer loosely to the point of $P$. 
Let $\mathcal{A}$ be a set of atoms and $\alpha$ a partition on $\mathcal{A}$. If the predicate
\[ P(\theta) \equiv (\forall A, B \in \mathcal{A}, \alpha(A) = \alpha(B) \Rightarrow A = B) \]
is pointed, then we will say that $\alpha$ is unifiable, and that the point of $P$ is the most general (mgu) unifier of $\alpha$. Not all partitions $\alpha$ are unifiable: for example, the following are non-unifiable partitions:
\[
(\left[ P(a), P(b) \right], \left[ P(x), P(y) \right]) \\
(\left[ P(x), Q(x) \right]) \\
(\left[ P(x, y), P(a, a) \right], \left[ P(y, x), P(b, b) \right])
\]

It follows from the work of Robinson (1965, 1970) that if there is a $\theta$ which unifies $\alpha$ then $\alpha$ is unifiable, and that mgu's of unifiable partitions are effectively and efficiently computable. Clearly, any substitution $\theta$ defines a partition $\alpha_\theta$ on $\mathcal{A}$ which is unifiable (since $\theta$ is a unifier), so that we may refer to $\theta$ as a partition.

The following easy lemma is a minor modification of a lemma in (Kowalski 1969).
Decomposition lemma. Let $\alpha$ be a unifiable partition of $\mathcal{A}$ with mgu $\sigma$, and let $\beta$ be a unifiable partition of $\mathcal{A}\sigma$, with mgu $\mu$. Then $\sigma\mu$ is the mgu of the partition $\alpha\beta$ on $\mathcal{A}$.

Proof. $\sigma\mu$ unifies $\alpha\beta$ since $\mu$ unifies $\mathcal{A}\sigma$.

Suppose that $\theta$ unifies $\alpha\beta$. Then $\theta$ unifies $\alpha$, and so $\theta = \sigma\lambda'$ for some $\lambda'$ which unifies $\beta$.

Therefore $\lambda' = \mu\lambda$ for some $\lambda$. But then $\theta = \sigma\mu\lambda$. So $\sigma\mu$ is the mgu of $\alpha\beta$.

QED.

A trivial inductive argument shows that any mgu of a partition $\alpha$ on $\mathcal{A}$ can be built up by compounding the mgu's $\sigma_1, \ldots, \sigma_k$ of partitions $\alpha_1, \ldots, \alpha_k$ where $\alpha = \alpha_1\alpha_2 \cdots \alpha_k$ and $\sigma_1$ is the mgu of $\alpha_1$ on $\mathcal{A}$, and $\sigma_i$ the mgu of $\alpha_i$ on $\mathcal{A}\sigma_1 \cdots \sigma_{i-1}$.
3.5.2 Liftable operations

Suppose we have a set $\mathcal{R}$ of structural operations defined for propositional calculus. Then they must be propositional operations (for in the propositional calculus, atoms have no internal structure). Hobinson's lifting lemma allows us in certain cases to define new structural operations for PC which 'mirror' the operations of $\mathcal{R}$ but have the instantiation search incorporated into them. The instance space is reduced from all instances to the most general instances necessary to make the mirrored propositional operations work.

In order to prove the lifting lemma I have found it necessary, for the first time, to make a detailed assumption about the form of structural operations.

A partial map $f: \mathcal{S} \rightarrow \mathcal{A}$, such that $f(S)$, if defined, occurs in $S$, will be called a selector. If $f(S) = A$, we will say that $f$ defines an occurrence of $A$ in $S$. There may be several occurrences of $A$ in $S$: we may have $f_1(S) = f_2(S)$ for distinct selectors $f_1$, $f_2$. We will now make
several assumptions about selectors and structures. The set of all selectors will be denoted by $\mathcal{F}$. 

(A1) $A \in \| S \|$ if $A = f(S)$ for some $f \in \mathcal{F}$.

(A2) If $\langle A_1, \ldots, A_n \rangle$ is a sequence of structures and $\{ f_1, \ldots, f_n \}$ a set of selectors then there is a structure $S$ with $f_1(S) = A_1$ and $f(S)$ undefined for all other $f \in \mathcal{F}$.

(A3) If, for all selectors $f$, $f(S_1) \neq f(S_2)$, then $S_1 = S_2$.

(A4) If $k$ is any map $A \rightarrow A$, then, for all $f \in \mathcal{F}$, $f$ is defined on $S$ iff it is defined on $k(S)$, and $f(k(S)) = k(f(S))$.

These assumptions correspond to (and are inspired by) similar assumptions often made about type-free data structures in programming languages: (1) says that the set of selectors is complete (or, if one prefers, (1) can be regarded as a definition of "occurring in" and hence of $\| S \|$); (2) says, essentially, that there is a constructor function for any meaningful pattern of selectors; (3) says that structures
have no hidden aspects; (4) says that components of structures may be simultaneously replaced by other components.

I believe these are independent. (A4 can be proved from A2, A3 and a finiteness assumption which we do not make.) They are quite strong assumptions, and it is possible to prove a version of the lifting lemma without them.

But the PC search space which results from this version is very redundant. It would be interesting to see whether the present version of the lifting lemma could be obtained under weaker assumptions. (I have expended considerable effort trying to do this, but to no avail.)

Notice that A4 means that there is a 1:1 correspondence between occurrences of atoms in a structure $S$ and its instance $S^\theta$, for any $\theta$. An instance of a structure must have the same "shape" as the structure. Thus sets of atoms cannot be regarded as structures under these assumptions. We will see, in section 3.5.3 below, that regarding structures as sets can be useful fiction, however.
Now, define a **linkage** to be a predicate of the form $\land (f_m(x_i) = f_n(x_j))$, where the $f_i$ are selectors. If $L(x_1, \ldots, x_n)$ is a linkage and $S_1, \ldots, S_n$ are structures, then $L(S_1, \ldots, S_n)$ is a conjunction of equalities between atoms in $\|S_1\| \cup \ldots \cup \|S_n\| = \emptyset$, say. We will call this conjunction a **linking condition**. We can treat a linking condition as an **object**, asking, for example, whether an equation occurs in it; or as a **statement**, asking for example, whether it is true. We will be swapping between these rapidly in what follows, but the meaning in each case should be clear from the context.

A certain amount of care over use and mention is needed, however.

I will use Quine's corner-quotation convention as follows: the expression $' \Gamma f_i(S_j)'$ denotes an expression obtained from $'f_i(S_j)'$ by replacing $'f_i'$ and $'S_j'$ by appropriate expressions. In general, if $<\text{string}>$ is an expression containing variables, then the result of writing $' \Gamma '$, followed by $<\text{string}>$, followed by $' \neg '$, is an expression which denotes an expression obtained by replacing variables in $<\text{string}>$ by suitable
expressions. For example, \( (i=j) \) denotes one of the expressions \( (1=1) \), \( (1=2) \), \( (2=1) \), \( (1=3) \), etc.

The linking condition \( L(S_1, \ldots, S_n) \) defines a partition \( \alpha \) on \( \mathcal{A} \) : \( \alpha(A) = \alpha(B) \) iff for some sequence

\[
\left[ f_{i_1}(S_{j_1}) \right], \ldots, \left[ f_{i_n}(S_{j_n}) \right], \quad f_{i_1}(S_{j_1}) = A, \quad f_{i_n}(S_{j_n}) = B,
\]

and

\[
\left[ f_{i_k}(S_{j_k}) = f_{i_{k+1}}(S_{j_{k+1}}) \right] \in L(S_1, \ldots, S_n).
\]

Let \( \sigma \) be the mgu of

\( \alpha \). \( \sigma \) will be called the mgu of \( L(S_1, \ldots, S_n) \).

It is clearly unique to within variance, since \( \alpha \) is a well-defined partition, by A4. Clearly also, if \( L(S_1 \theta, \ldots, S_n \theta) \) is true for some \( \theta \) then \( L(S_1 \sigma, \ldots, S_n \sigma) \) is true. Suppose that for every \( f \in F^\mathcal{N} \), if \( f(S_n) \) is defined then

\[
\left[ f(S_n) = f_i(S_j) \right] \in L(S_1, \ldots, S_n), \text{ where } j \neq n.
\]

Then we will say that \( L(S_1, \ldots, S_n) \) defines \( S_n \).

Now let \( R \) be a constructive operation. We will say that \( R \) is liftable if the following two conditions hold:

1. \( \forall S_1, \ldots, S_n, \ R(S_1, \ldots, S_n) \supset \exists \ L, \ L(S_1, \ldots, S_n) \) is true, and defines \( S_n \).
Let such an $L$ be called a key for $S_1, \ldots, S_n$.

(2) $\forall S_1, \ldots, S_n, (R(S_1^\lambda, \ldots, S_n^\lambda) \& (L(S_1, \ldots, S_n) \text{ is true})) \Rightarrow R(S_1, \ldots, S_n)$

where $\lambda$ is a substitution, and $L$ is a key for $S_1^\lambda, \ldots, S_n^\lambda$.

The intuition behind this definition is that $R$ is liftable when it can be defined roughly as follows: $R$ applies to $S_1, \ldots, S_{n-1}$ if some linkage holds true for them; and then $S_n$ is constructed from certain of the atoms occurring in $S_1, \ldots, S_{n-1}$ in a way which depends only upon their mode of occurrence in $S_1, \ldots, S_{n-1}$. The second condition indicates that the linkage is a sufficient as well as necessary condition: it says that we can make occurrences of the same atom (in $S_i^\lambda$) different (in $S$) and $R$ still applies if $L$ is still true. Note that $\lambda$ is a map between atoms, not between occurrences of atoms.
Now, let \( R \) be a liftable operation. Define the lifted operation \( \hat{R} \) as follows:
\[
\hat{R}(s_1, \ldots, s_n) \equiv \exists L \exists \sigma. L \text{ is a key for } s_1 \sigma, \ldots, s_n \sigma \quad \& \quad \sigma \text{ is the mgu of } L(s_1, \ldots, s_n) \quad \& \quad R(s_1 \sigma, \ldots, s_{n-1} \sigma, s_n).
\]

Clearly, if \( R(s_1, \ldots, s_n) \) then \( \hat{R}(s_1, \ldots, s_n) \) by taking \( \theta = \varepsilon \). However, \( \hat{R} \) may apply to structures which need to be properly instantiated in order that \( R \) will apply to them: and it then uses only the most general such instance.

We can now prove the important sensible Lifting Lemma. Let \( R \) be a liftable operation, and let \( \hat{R} \) be the lifted operation of \( R \). Suppose \( R(s_1 \theta, \ldots, s_n \theta, s') \). Then there is a structure \( S \) with \( s' = S \lambda \) for some \( \lambda \) and \( \hat{R}(s_1, \ldots, s_n, S) \).

**Proof.** Since \( R \) is liftable, there is a key \( L \) with:
\[
L = L(s_1 \theta, \ldots, s_n \theta, s') \text{ is true} \quad (1)
\]
Now let \( L' = L'(s_1 \theta, \ldots, s_n \theta) \) be the linking condition such that \( L' \) defines that same partition on \( (\| s_1 \| \cup \ldots \cup \| s_n \|) \theta = \emptyset \theta \) as does \( L \),
so that

\[(L \equiv L' \& L'')\) follows from the transitive and symmetry laws of equality, where every equation in \(L''\) has the form

\[\Gamma f_i(S') = f_j(S_k\theta)^T\]

for some \(k \leq n\).

Let \(\sigma\) be the mgu of \(L'\), so that

\[L'(S_1\sigma, \ldots, S_n\sigma)\) is true, \quad \text{by (1)}.

Then by construction, \(\sigma\) is the mgu of \(L\), since \(\|S'\| \leq \Theta\), since \(R\) is sensible.

Now, \(\theta = \sigma\lambda\) for some \(\lambda\). For each equation

\[\Gamma f_i(S') = f_j(S_k\theta)^T \quad \text{in} \quad L'\]

\[f_i(S') = f_j(S_k\theta) = f_j(S_k\sigma\lambda) = (f_j(S_k\sigma))^\lambda\]

by A4.

And \(f_j(S')\) is undefined for all other \(f_j \in \Gamma'\), since \(L\) defines \(S'\).

Let \(S\) be a structure obtained by replacing \(f_i(S')\) by \(f_j(S_k\sigma)\) in \(S'\), for each \(f_i\) for which there is such an equation in \(L''\). \(S\) is a structure by A2. Moreover,

\[f_i(S') = (f_i(S))^\lambda \quad \text{by construction, since \(L\) is true} = f_i(S\lambda) \quad \text{by A4}

for each \(f_i\) for which \(f_i(S')\) is defined; and \(f_j(S')\) and \(f_j(S\lambda)\) are both undefined for all other \(f_j\).

Therefore \(S' = S\lambda\), \quad \text{by A3.}
By construction, $L''(S_1 \sigma, \ldots, S_n \sigma, S)$ is true, since every equation in it has the form $f_i(S) = f_j(S_k \sigma)$. Therefore,

$$L(S_1 \sigma, \ldots, S_n \sigma, S) \equiv L'(S_1 \sigma, \ldots, S_n \sigma) \& L''(S_1 \sigma, \ldots, S_n \sigma, S)$$

is true. But then

$$R(S_1 \sigma, \ldots, S_n \sigma, S)$$

since $R$ is liftable (by the second condition on liftability); that is,

$$\hat{R}(S_1, \ldots, S_n, S).$$

QED.
Now let $R$ be a set of liftable operations, and let $\hat{R}$ be the set of lifted operations. If $\mathcal{D}'$ is a derivation relative to $R$ with premisses $S_1\theta, \ldots, S_n\theta$ and conclusion $S'$, then an obvious inductive argument from the lifting lemma shows that there is an isomorphic derivation $\mathcal{D}$ relative to $\hat{R}$ with premisses $S_1, \ldots, S_n$ and conclusion some $S$ with $S' = S\lambda$ for some $\lambda$. We will say that $\mathcal{D}$ lifts $\mathcal{D}'$.

The definition of $\hat{R}$ refers to the mgu of $L$. This, in fact, as explained earlier, is a slight abuse of terminology since any variant of a point is another point. If $\sigma_1$ and $\sigma_2$ are variant such mgu's then the results of $\hat{R}$ in the two cases will be variants (since $R$ is propositional). It is conventional therefore in describing lifted operations to allow arbitrary variants of the input structures to be used. We will follow this useful convention in what follows, so that $\hat{R}(S_1, \ldots, S_n)$ will always be taken to mean $\hat{R}(S_1\lambda_1, \ldots, S_{n-1}\lambda_{n-1}, S_n)$ where $S_i\lambda_1$ is a variant of $S_i$, and where $S_i\lambda_1$ and $S_j\lambda_j$ share no variables: we will say that the $S_i$ have been standardised apart. This standardisation is the most general
such convention since the mgu $\sigma$ can always merge variables together if required. Notice that the freedom to replace a structure by a variant corresponds to the inference rule of change of bound variables in a quantified formula. Thus, this convention will be valid only when all variables in a structure are bound in the structure. This is indeed the case in all TP systems known to me.

With this convention, the above result appears somewhat stronger, since the premisses of $D'$ can now be separate instances of the premisses of $D$: $S_1\theta_1, \ldots, S_n\theta_n$.

We will say that a set $R$ of constructive operations is **complete** for a language if, whenever $S_0$ is a set of structures corresponding to an expression of the language which has a proof, then the complete search space from $S_0$ relative to $R$ contains a node labelled with a proof of the expression.

Now suppose we have a set $R$ of liftable operations which are complete for the propositional
calculus. Then the set \( \mathcal{R} \) of lifted operations is complete for PC. This important result follows from the above corollary to the lifting lemma, and Herbrand's theorem. For suppose \( \phi \) is a PC assertion which has a proof. Then by Herbrand's theorem (proved in section 4. below), there is a set of instances \( \phi' = \{ \phi \theta_1, \ldots, \phi \theta_m \} \) such that \( \phi' \) has a propositional calculus proof.

Let \( \mathcal{B}_0 = \{ S_1, \ldots, S_n \} \) be the initial structures corresponding to \( \phi \); then the initial structures corresponding to \( \phi' \) are \( \{ S_1 \theta_1, \ldots, S_n \theta_m \} = \mathcal{B}'_0 \).

Now since \( \mathcal{R} \) is complete for propositional calculus and since \( S' \) has a propositional calculus proof, there is a derivation \( \mathcal{D}' \) relative to \( \mathcal{R} \) from \( \mathcal{B}'_0 \) of a proof \( S' \). Therefore by the above corollary (in the strong form), there is an isomorphic derivation \( \mathcal{D} \) relative to \( \mathcal{R} \) from \( \mathcal{B}_0 \) of a structure which has \( S \) as an instance; and this is a proof of \( \phi \).

This familiar argument (Robinson 1965) provides a methodology for designing search spaces for PC: we must find search spaces for propositional calculus, satisfying the assumptions (A1)-(A4), all of the operations defining which are liftable (they must of course be propositional). Any such collection of operations automatically defines a
collection of lifted operations which is complete for PC if the propositional operations are complete for propositional calculus.

If \( \mathcal{G}' \) is the complete search space from \( \mathcal{R} \) relative to \( \mathcal{G}_0 \), and \( \mathcal{G}_0 = \left\{ S_i \theta_i : S_i \in \mathcal{R}_0 \right\} \), then the complete search space \( \mathcal{G} \) from \( \mathcal{G}_0 \) relative to \( \mathcal{R} \) contains derivations lifting every derivation in \( \mathcal{G}' \). Thus all simple redundancy in \( \mathcal{G}' \) is mirrored in \( \mathcal{G} \). But \( \mathcal{G} \) may contain new redundancies also, since it may contain derivations which do not lift any derivation in \( \mathcal{G}' \). In the next section we discuss how such redundancies can arise through lifting a certain kind of compound operation.
3.5.3 Deletion operators and factoring

In this section I discuss a particular class of compound operations $R = R_1 \otimes R_2$ which arise very commonly in TP systems for PC, and which can be conveniently discussed together.

It often happens that we are interested in structures which contain no duplicate occurrences of atoms in certain substructures. (For example, if structures are sequences of atoms then we may want to not consider sequences in which an atom has more than one occurrence. Similarly, the semantic trees considered in chapter 4 are trees labelled with sets of atoms in which no branch contains an atom more than once.) However, the basic structural operations which one wishes to use are perhaps liable to introduce duplicated occurrences (for example, by building a new structure from two or more duplicate-free structures which have an atom in common), and so one introduces a deletion operation which, given a structure with duplications, removes an occurrence of an atom so as to remove the duplication. The structural
operation one uses in the search space is then $R_1 \otimes R_2$, where $R_1$ is the original operation and $R_2$ is the deletion operation, and the corresponding notion of admissibility is that the (pseudo)-structure should contain no duplications of a sort which $R_2$ would remove.

Suppose, for example, structures were sequences of atoms $<A_1, \ldots, A_n>$, in which we wished to remove duplicates: $<A,A> \rightarrow <A>$. Then if $R_2$ is this removal operation, $R = R_1 \otimes R_2$ is an operation which 'automatically' removes duplicates from its results. It would be somewhat plausible to construe $R$ as an operation upon sets of atoms in this case, and this has in fact usually been done in such cases in the literature.

Suppose that $R_2$ is a unary liftable operation such that each key for $S_1, S_2$ defines both $S_1$ and $S_2$ and has the form

$L(S_1, S_2) = \Gamma_f(S_1) = f_j(S_1) \& L'(S_1, S_2)$

where $f_i \neq f_j$ and $L'(S_1, S_2)$ is a conjunction of equations of the form $\Gamma_{f_p}(S_1) = f_q(S_2)$ which contains no expression $\Gamma_{f_p}(S_1)$ more than once. Then $R$ will be called a deletion operation and the
equation \( f_i(S_1) = f_j(S_1)^7 \) will be called a duplication.

Since \( L'(S_1, S_2) \) contains only one reference to each selection from \( S_1 \) or \( S_2 \), and since \( L(S_1, S_2) \) defines both \( S_1 \) and \( S_2 \), the condition \( L'(S_1, S_2) \) defines a 1:1 correspondence between all occurrences of atoms in \( S_2 \) and all occurrences of atoms in \( S_1 \) except possibly one of \( f_i(S_1) \), \( f_j(S_1) \): we suppose without loss of generality that \( f_i(S_1) \) is thus omitted, if any is. We will say that \( R \) deletes \( f_i(S_1) \) from \( S_1 \).

Corresponding to a deletion operation then there is an obvious admissibility condition: \( S \) is admissible iff no duplications hold true of it which would trigger an application of \( R_2 \), i.e. iff there is no \( S' \) with \( R_2(S, S') \).

If we have an \( S \) for which \( f_i(S) = f_j(S)^7 \) is true, where this is a duplication, (Not all such equations need be duplications: certain kinds of repeated occurrence may be harmless, e.g. occurrences of atoms on different branches of a semantic tree.) then we can construct an
S' for which R₂(S, S') holds, by A2. Thus if R₂(S₁, S₂) holds then \( f_i(S_i) = f_j(S_i) \) is true and therefore \( f_i(S_i h) = f_j(S_i h) \) is true for any map \( h: \hat{A} \rightarrow \hat{A} \); and therefore we can find an S' with R₂(S₁h, S₂). Now by construction and A3, S' = S₂h. Thus R₂(S₁h, S₂h). This fact will be important in the proof of:

**Lemma 3.1** If R₁ is liftable and R₂ is a deletion operator, then R₁ ⊗ R₂ is liftable.

**Proof.** Suppose R(S₁, ..., Sₙ₊₁). Then for some sequence Sₖ of pseudostructures, R₁(S₁, ..., Sₙ, Sₖ) & R₂(Sₖ⁻¹, Sₖ) & ... & R₂(Sₖ, Sₙ₊₁)

Since R₁ and R₂ are liftable, there are key linkages L₁ such that

L' = (L₀(S₁, ..., Sₙ, Sₖ) & L₁(Sₖ⁻¹, Sₖ) & ... & Lₖ(Sₖ, Sₖ₊₁)) is true, where we define Sₖ₊₁ = Sₙ₊₁ for notational convenience.

Moreover, L₁(Sₖ⁻¹, Sₖ₊₁) has the form

f_i(Sₖ⁻¹) = f_j(Sₖ⁻¹) & f_i(Sₖ) = f_k_1(Sₖ⁻¹) & L₁(Sₖ⁻¹, Sₖ₊₁)

where L₁(Sₖ⁻¹, Sₖ₊₁) contains only equations linking Sₖ⁻¹ and Sₖ₊₁; and L₁(...) defines Sₖ₊₁ for 0 ≤ i ≤ k.
Now suppose $f \in \mathcal{F}$ and $f(S'_i)$ is defined for some $i > 1$. Then there is a unique true equation
\[ r f(S'_i) = g_1(S'_{i-1}) \text{ in } L_{i-1}(S'_{i-1}, S'_i), \]
since this defines $S'_i$; therefore, $g_1(S'_{i-1})$ is defined. Similarly, there is a unique equation
\[ r g_1(S'_{i-1}) = g_2(S'_{i-2}) \text{ in } L_{i-2}(S'_{i-2}, S'_{i-1}), \]
and so on. Eventually we have a unique equation
\[ r g_{i-1}(S'_1) = g(S_p) \text{ for some } S_p. \]
Then the equation
\[ r f(S'_1) = g(S_p) \]
is true. We will say that $r g(S_p)$ corresponds to $r f(S'_1)$.

Let $M = M(S_1, \ldots, S_n, S_{n+1})$ be the conjunction of all such equation for $i = k+1$; then $M$ is a linkage condition which is true and which defines $S_{n+1}$.

For each $i$, $1 \leq i \leq k$, there is a duplication $r f_i(S'_i) = f_{j_i}(S'_i)$. Let $r g_1(S_p)$ and $r g_2(S_q)$ correspond respectively to $r f_i(S'_i)$ and $r f_{j_i}(S'_i)$; then the equation
\[ r g_1(S_p) = g_2(S_q) \]
is true. Let $M' = M'(S_1, \ldots, S_n)$ be the conjunction of all such equations for each $i$: then $M'$ is a true linkage condition.

Let $M_0 = M_0(S_1, \ldots, S_n)$ be the linkage condition which defines the same partition on $\| S_1 \| \cup \ldots \cup S_n \|$ as does $L_0(S_1, \ldots, S_n, S'_1)$. Let $M'' = r M_0 \& M'$ be the key for $S_1, \ldots, S_{n+1}$. 
Now suppose in addition that $T_1, \ldots, T_{n+1}$ are (admissible) structures with $T_1 = S_1$, where $h$ is a substitution, and suppose that $M'(T_1, \ldots, T_{n+1})$ is true. I will show that $R(T_1, \ldots, T_{n+1})$.

For each $i$, $1 \leq i \leq k+1$, and each $f$ for which $f(S_i)$ is defined, let $g_f(S_p)$ correspond to $f(S_i)^7$. Let $T'_i$ be the result of replacing $f(S_i)$ by $g_f(T_p)$ in $S'_i$, for each $f$ (well-defined by A2). Then, for any $f$,

$$f(T'_i) = (f(T'_i))\lambda \quad \text{by A4}$$

$$= g_f(S_p) \quad \text{by construction}$$

$$= f(S'_i) \quad \text{since L' is true.}$$

so $T'_i = S'_i$, by A3.

Let $f'(S_p) = f(S'_i)^7$ be an equation in $L_0(S_1, \ldots, S_n, S'_i)$. $f'(S_p)^7$ corresponds to $f(S'_i)^7$, so $f(T'_i) = f'(T_p)$ by construction:

$$f'(T_p) = f(T'_i)^7$$

is true. Moreover, for every equation $f(S_p) = f''(S_q)^7$ in $L_0(S_1, \ldots, S_n, S'_i)$, $f'(T_p) = f''(T_q)^7$ since $M''(T_1, \ldots, T_{n+1})$ is true. Thus, $L_0(T_1, \ldots, T_n, T'_1)$ is true. Similarly, $L_1(T'_1, T'_{i+1})$ is true for $1 \leq i \leq k$. Since $R_1$ and $R_2$ are liftable, we have $R_1(T_1, \ldots, T_n, T'_1)$ and $R_2(T'_1, T'_{i+1})$ for $1 \leq i \leq k$. 
Now, $T_{n+1} = S_{n+1}$, by assumption; so for all $f$, $f(T_{n+1})$ is defined iff $f(S_{n+1})$ is defined and $M(T_1, \ldots, T_n, T_{n+1})$ is true; hence, by construction, $f(T_{n+1}) = f(T_{k+1})$ for all $f$: so $T_{n+1} = T_{k+1}$ by A3. Now $T_{n+1}$ is admissible, since, if it were not, $T_{n+1} = S_{n+1}$ would not be either (for if a deletion applies to $S$ then it does to $S_h$ for any $h$); but no $T_i$ is admissible since $R_2$ applies to each one. Therefore, $T_i$ is a sequence of pseudostructures, so $R(T_1, \ldots, T_{n+1})$.

QED.

Now suppose $R = R_1 \otimes R_2$, as above. Then $\hat{R}$ exists. Hopefully, we should be able to piece together $\hat{R}$ from some combination of $\hat{R}_1$ and $\hat{R}_2$. This we can in fact do, with some care.

Suppose $R(S_1, \ldots, S_{n+1})$. Then for some mgu $\sigma$, $R(S_1 \sigma, \ldots, S_n \sigma, S_{n+1})$: that is,

$R_1(S_1 \sigma, \ldots, S_n \sigma, S_1') \& R_2(S_1', S_2') \& \cdots \& R_k(S_k', S_{k+1}')$

where $\| S_1' \| \subseteq \| S_1 \| \cup \cdots \cup \| S_n \|$, and $\sigma$ is the mgu of $L_0(S_1 \sigma, \ldots, S_n \sigma, S_1') \& L_1(S_1', S_2') \& \cdots \& L_k(S_k', S_{k+1}')$.

Now, by the decomposition lemma (section 3.5.1),
it follows that \( \mathcal{O} = \mathcal{O}_0 \mathcal{J}_1 \ldots \mathcal{J}_k \), where \( \mathcal{O}_0 \) is the mgu of \( L_0(S_1, \ldots, S_n, S_1') \), and \( \mathcal{J}_1 \) the mgu of \( L_1(S_1 \mathcal{O}_0 \ldots \mathcal{J}_1 - 1, S_1' \mathcal{O}_0 \ldots \mathcal{J}_1 - 1) \)

Thus

\[
R_1(S_1 \mathcal{O}_0 \ldots \mathcal{J}_k, \ldots, S_n \mathcal{O}_0 \ldots \mathcal{J}_k, S_1')
\]

where \( \mathcal{O}_0 \) is the mgu of \( L_0(S_1, \ldots, S_n, S_1') \). By the proof of the lifting lemma,

\[
R_1(S_1 \mathcal{O}_0, \ldots, S_n \mathcal{O}_0, s')
\]

where \( S_n \mathcal{O}_0 \ldots \mathcal{J}_k = S_1' \).

Thus, similarly, \( R_2(S_1'' \mathcal{O}_0 \ldots \mathcal{J}_k, S_1') \)

where \( \mathcal{O}_1 \) is the mgu of \( L_1(S_1 \mathcal{O}_0', S_2 \mathcal{O}_0) \).

By the proof of the lifting lemma again

\[
R_2(S_1'' \mathcal{O}_1, S_2')
\]

where \( S_2'' \mathcal{O}_1 \ldots \mathcal{J}_k = S_2' \).

Proceeding thus, we obtain

\[
R_1(S_1 \mathcal{O}_0, \ldots, S_n \mathcal{O}_0, s') \land R_2(S_1'' \mathcal{O}_1, S_2') \land \ldots \land R_2(S_k'' \mathcal{O}_k, S_{n+1})
\]

i.e. \( \hat{R}_1(S_1, \ldots, S_n, s') \land \hat{R}_2(S_1, S_2') \land \ldots \land \hat{R}_2(S_k, S_{n+1}) \).

Thus we can apply \( \hat{R} \) by applying \( \hat{R}_1 \) and \( \hat{R}_2 \) in sequence, 'mirroring' the applications of \( R_1 \) and \( R_2 \) which define \( R \). If, (as usually is the case) an admissible structure can have an inadmissible instance, then it is better to use the other formulation of \( R_1 \otimes R_2 \) as being an application
of $R_1$ preceded by sequences of applications of $R_2$. For in this case, instances of an admissible input structures may be preadmissible and hence require applications of $R_2$ before further operations can be applied. The above argument and lemma apply to this case also: the proofs need only tiresome terminological changes. I will ignore this complication in what follows.

In the propositional space, $R_2$ is applied until it is no longer applicable. However, we cannot make the same insistence for $\hat{R}_2$. For, some $S_1''$ for $i < k+1$ may be admissible, even though $S_1'' \sigma_1$ is not: for $\sigma_1$ may make a duplication true which is false of $S_1''$. Clearly, however, if $\hat{R}_2$ is $R_2$, then we 'must' apply it, as in the ground case: the refinement test is that no preadmissible structure labels an input node of a polyedge whose body is labelled with any other operation than $R_2$. Let us define

$$\tilde{R}_2(S_1, S_2) \equiv (\hat{R}_2(S_1, S_2) \land \neg R_2(S_1, S_2)).$$

That is, $\tilde{R}_2(S_1, S_2)$ if $S_1$ is admissible but has a preadmissible instance. Then the refinement test makes no mention of $\hat{R}_2$, which is therefore an operation which can be applied freely in the search
space. The result of lifting $R = R_1 \otimes R_2$ is thus two operations $\hat{R}_1 \otimes R_2$ and $\hat{R}_2 \otimes R_2$.

An application of $R_2$ represents in effect a choice of a particular instance of the input structure. The point is that lifting does not eliminate the instantiation choice range, but only replaces it by a much restricted choice range, viz. that of most general unifiers of certain key partitions of atoms. But this reduced choice range must be searched to preserve completeness.

Now suppose $S_1$ is a structure to which $R_2$ is applicable, say $\Gamma f^{(1)}_1 (S_1 \sigma_1) \equiv f^{(2)}_1 (S_1 \sigma_1)$ is true, for the mgu $\sigma_1$, but $f^{(1)}_1 (S_1) \not\equiv f^{(2)}_1 (S_1)$. Suppose also that $S_2$ is derived from $S_1$ and possibly some other structures by applying $R_1$. The linking condition in the application of $\hat{R}_1$ may (and usually will) contain $f^{(i_j)}_1 (S_1 \sigma_2) = f^{(j)}_1 (S)$ where $S$ is the result structure. Suppose now that some operation $R$ is applied to $S$ which involves instantiating it to $S \sigma_3$, where $\sigma_2 \sigma_3 = \sigma_1 \lambda$ for some $\lambda$. Then $f^{(1)}_1 (S_1 \sigma_2 \sigma_3) = f^{(2)}_1 (S_1 \sigma_2 \sigma_3)$. Thus we have a derivation which lifts no propositional derivation: for to generate the
instantiation $S \sigma_3$ we would have had to apply $R_1$ to a preadmissible structure $S_1 \sigma_2 \sigma_3$, which is forbidden by the refinement test.

If $R$ is itself an application of $\sim R_2$, we have the danger of simple redundancy. In fact, usually when a structure to which $\sim R_2$ is applicable occurs in a derivation, but $\sim R_2$ is not applied to it, then we have the opportunity of applying $\sim R_2$ to delete the occurrence of an atom which descends from the occurrence we could have deleted earlier; and all such applications would yield structures which could have been derived from the result of applying $\sim R_2$ earlier.

This situation does not arise if we lift the derivation space of $\{R_1, R_2\}$: for then, the putative derivation in which $R$ is applied to $S$ is simply illegal, i.e. not a structure, for it fails the refinement test. In general, if structures are sufficiently rich that all inadmissibility in a derivation is apparent in the conclusion of the derivation, then the refinement can be lifted successfully without introducing redundancy, by imposing a refinement test which removes all structures which could only have been derived by an admissible derivation.
An example of this is provided by 'm-factoring' (Kowalski 1970), which eliminates some redundancy of this sort, and is defined on derivations in the resolution space. The idea behind m-factoring is of general utility in these circumstances: it restricts applications of $\sim_2$ to those in which the duplication $f_1(S_1) = f_j(S_1)$ in the key is such that $f_1(S_1) = f_{i_1}(S_p)$ and $f_j(S_1) = f_{i_2}(S_q)$ for some $S_p, S_q$, inputs to the closest preceding application of $\sim_1$, with $p \neq q$. That is, it insists that one applies $R_2$ only to delete duplicate occurrences which do not descend from the same immediate premiss structure in the derivation. This corresponds to the propositional deletion regime in which all deletions are applied to each structure as soon as it is generated.

There are other ways of restricting applications of $\sim_2$ corresponding to different deletion regimes in the propositional system. Thus the method of 'distinguished-literal' factoring (Kowalski & Hayes 1969) corresponds to the deletion regime in which duplicates of an atom are deleted only when that atom is immediately resolved upon in the next polyedge of the derivation. This collapses
into structures consisting of a clause and a sector \( \langle C, f \rangle \), the latter being what distinguishes a literal. (The m-factoring scheme has the advantage, in the usual account of resolution, of corresponding to the 'automatic' deletion in the propositional space which is an inevitable by-product of describing structures as sets.)

3.5.4 Lifting choice functions and refinement tests

Suppose we have a TP system for propositional calculus, and a class \( \mathcal{R} \) of choice ranges arises from the structures in the search space of the system. If there is a lifted TP system for PC, then the structures in this lifted system are isomorphic to those in the original system, and so we have in the lifted system a corresponding class \( \hat{\mathcal{R}} \) of choice ranges. Thus the PC system has a choice range for every choice range in the propositional system, and in addition has the choice range of most general unifiers.
There is no effective choice function for this instantiation range, by the Ehrenfeucht-Rabin theorem: for if there were, we could lift a deterministic sensible TP system for the propositional calculus, and using the choice function get a deterministic sensible system for PC, which is impossible. However, if the propositional system uses a choice function for some ranges, then we might hope to be able to use a corresponding function for the corresponding ranges. This will only be effectively possible, however, if we can be certain that a choice made at one stage in a derivation will continue to be appropriate, in retrospect, after further instantiations. That is, if R is a choice range in , and f a choice function, then we must have f(R) = (f(R)) for any λ. (In fact, we only need: for any λ which is the composition of mgu's generated in a derivation. But in practice it seems impossible to capitalise on this slender advantage.) For, without this, we can have no confidence that a choice during the operations of the system will continue to be appropriate; and if it is not, then the system would have to backtrack and find another; and then the utility of the choice function is lost, since the system would in effect be searching the choice range.
One case of importance however is that in which
any choice function can be used. We will in this
case say that the range is selectable (following
the terminology of Kowalski & Kuehner 1971).
If \( \hat{R} \) is selectable then \( \hat{R'} \) is also selectable:
for, we need only that \( f(R \lambda) = f'(R) \lambda \) for
some \( f' \); and this is a vacuous condition by A4.
This case will in fact be important, as choice
ranges corresponding to the disjunction operator
are often selectable. The above trivial observation
shows that this selectability lifts intact to the
predicate calculus system.

As remarked earlier, throughout the discussion
of lifting I have assumed for simplicity that
refinement tests were absorbed into structural
operations:

\[
R'(S_1, \ldots, S_n) \equiv R(S_1, \ldots, S_n) \& K(S_n).
\]

However, it is often useful to consider them
separately. We need conditions on \( R \) and \( K \) which
ensure that \( R' \) is liftable. This is, fortunately,
straightforward.
Lemma 3.2 Suppose that $R$ is liftable and $K$ obeys the condition: $K(S^\lambda) \supseteq K(S)$ for all $S$ and $\lambda$. Then $R'$ is liftable.

**Proof.** Let $L$ be a key for $S_1, \ldots, S_n$ for $R$, and define $L$ to be a key for $R'$ also. Then we have:

\[ R'(S_1, \ldots, S_n) \supseteq R(S_1, \ldots, S_n) \supseteq (L(S_1, \ldots, S_n) \text{ is true}) \]

and:

\[ R'(S_1^\lambda, \ldots, S_n^\lambda) \supseteq (R(S_1^\lambda, \ldots, S_n^\lambda) \& K(S_n^\lambda)) \]

and so

\[ R'(S_1^\lambda, \ldots, S_n^\lambda) \& (L(S_1, \ldots, S_n) \text{ is true}) \supseteq R(S_1, \ldots, S_n) \& K(S_n) \text{ since } R \text{ is liftable.} \]

\[ \equiv R'(S_1, \ldots, S_n) \]

QED.

There are no doubt less restrictive requirements on $K$ which yield liftable $R'$ from liftable $R$, but this will suffice. More sophisticated conditions would seem to involve using a different key linkage for $R'$ than for $R$, which makes a general result difficult.
4. SEMANTIC TREES

4.0 Introduction

I investigate in this chapter a particular class of structures, which are basically truth-tables organised as trees. These semantic trees were introduced, as a technical tool in completeness proofs for resolution, by J.A. Robinson (1968), and developed further for this purpose by R. Kowalski and myself. (Kowalski & Hayes 1969). The results in sections 4.4.1 to 4.4.4 below are closely based on this joint work.

I distinguish two broad classes of TP system based on the way in which the system grows its trees. Conventional resolution is a collapse of the bottom-up systems, while "linear resolution" systems are collapses of top-down systems. It is conventional, as the name suggests, to regard linear resolution as a refinement of conventional resolution. The present account offers a more useful way of comparing linear and non-linear resolution systems, and greatly simplifies the proofs. Several other non-resolution systems
are collapses of top-down systems, including Prawitz' matrix reduction system (Prawitz 1969), Loveland's model elimination (Loveland 1969), and Kowalski & Kuehner's SL-resolution (Kowalski & Kuehner 1971).
4.1 Notation and Preliminary Lemmas.

4.1.1 Trees

A tree $T$ is a set of nodes together with a relation parent, such that:

(i) If $N_1$ and $N_2$ are parents of $M$ then $N_1 = N_2$; Define: $M$ is an ancestor of $N$ if $M$ is the parent of $N$, or $M$ is the parent of an ancestor of $N$.

(ii) There is a unique $N_0 \in T$, called the root, which is an ancestor of every other node.

(iii) The relation ancestor is a partial ordering.

We will think of trees growing downwards. The unique parent of $N$ will be denoted by $N'$.

If $M$ is an ancestor (parent) of $N$, then $N$ is a descendant (son) of $M$. If $N$ has no descendants then $N$ is a tip; otherwise $N$ is an interior node. We will write $N \prec_T M$ ($\ll_T M$) to mean that $N$ is the parent (an ancestor) of $M$ in the tree $T$. 
The set of sons of \(N\) is called the **fan at** \(N\). \(N_1\) and \(N_2\) are brothers iff they belong to a single fan. If every fan in a tree is finite then the tree is **finitely branching**. We will write \(f_N\) for the fan at \(N\).

A branch \(B\) is a (finite or infinite) sequence \(\langle M_1, \ldots, M_i, \ldots \rangle\) of nodes totally ordered by the parent relation. We will use \(B\), \(B_1\) etc. to denote branches. The finite sequence \(\langle M, \ldots, N \rangle\) is the branch from \(M\) to \(N\) and is denoted by \(B_{M,N}\). The branch \(B_{N_0,N}\) is the branch to \(N\); we will write \(B_N\) for brevity. The branch \(B_{M,N}\) where \(N\) is a tip, and the infinite branch \(\langle M, \ldots \rangle\), are branches from \(M\). A branch from the root is a branch of the tree.

Clearly, \(M\) is an ancestor of \(N\) just in case there is a branch \(B_{M,N}\). If neither of \(M, N\) is an ancestor of the other then \(M\) and \(N\) are **independent**.

A **labelled tree** \(\langle T, s \rangle\) is a tree together with a function \(s\) from nodes to **labels**, which may be arbitrary. We will write \(Ns\) for \(s(N)\).
An ordered tree is a tree together with a total ordering of each fan. We will regard a fan in an ordered tree as a sequence of nodes \( \langle N, \ldots \rangle \). Thus the ordered trees

\[
\begin{array}{c}
\text{and}
\end{array}
\]

are distinct. (Here and below we represent the ordering of fans by the left-right ordering on a diagram.)

A subset \( T' \subseteq T \), ordered by \( \ll_T \), where \( N \ll_{T'} M \) iff \( N \ll_T M \) and \( N, M \in T' \), is a subtree of \( T \) if, whenever \( N \ll_T M \) then there is a unique \( K \in T \) such that \( N \ll_T K \) and \( K \ll_T M \). If \( N \ll_T M \) iff \( N \ll_T M \) and \( N, M \in T' \), then \( T' \) is a connected subtree of \( T \). If \( T' \) is a proper subset of \( T \) then \( T' \) is a proper subtree of \( T \). More graphically, \( T' \) is obtained from \( T \) by deleting nodes, throwing away all but possibly one of the subtrees below those nodes, and glueing the pieces together as economically as possible.
We will want various means of defining subtrees. One useful way is by selecting a subset of each fan. The following easy lemmas establish this.

**Lemma 4.1.** Let \( \mathcal{F} \) be a function from fans in \( T \) into their subsets: \( \mathcal{F}(\mathcal{F}) \subseteq \mathcal{F} \). Then there is a subtree \( T' \) of \( T \) such that \( N \in T' \) only if \( N \in \mathcal{F}(\mathcal{F}) \) for some \( \mathcal{F} \) in \( T \).

**Proof.** We define a set \( T' \) inductively as follows.

(i) \( N_0 \in T' \).

(ii) If \( N \in T' \) and \( M \in \mathcal{F}(\mathcal{F}) \) then \( M \in T' \) and \( N < T'M \).

(iii) Suppose \( N \in T' \) and \( \mathcal{F}(\mathcal{F}) \) is empty. For each \( N_1 \in \mathcal{F}(\mathcal{F}) \), choose an \( N_2 \in T \) such that \( \mathcal{F}(\mathcal{F}) \) is empty for every \( M \in \mathcal{F}(\mathcal{F}) \) but \( \mathcal{F}(\mathcal{F}) \) \( \supseteq N_2 \). Put \( N_2 \in T' \), for each \( N_1 \in \mathcal{F}(\mathcal{F}) \) and \( N < T'N_2 \).

Let the ordering on \( T' \) be that induced by \( T \). Clearly, by construction, \( T' \) obeys the required conditions for a subtree. QED.

**Lemma 4.2** Let \( \mathcal{F} \) be a function from fans in \( T \) into their nonempty subsets. Then there is a unique connected subtree \( T' \) of \( T \) such that \( N \in T' \) iff \( N \in \mathcal{F}(\mathcal{F}) \) for some \( \mathcal{F} \) in \( T \).
Proof. (i) as above.

(ii) as above: \( f(\mathcal{Q}_N) \) is never empty. Clearly \( T' \) is uniquely determined by \( f \).

QED.

If \( \langle T, s \rangle \) is a labelled tree and \( T \) has a subtree \( T' \) then \( \langle T', s \rangle \) is a sublabelled tree. If \( T \) is ordered then \( T' \) is a subordered tree if, whenever \( \langle N_1, \ldots, N_i, \ldots \rangle \) is the fan at \( N \) in \( T' \), and \( \mathcal{Q} = \langle M_1, \ldots, M_i, \ldots \rangle \) is the fan at \( N \) in \( T \), then for some monotonic subsequence \( \langle M_{j_1}, \ldots, M_{j_i}, \ldots \rangle \) of \( \mathcal{Q} \), \( M_{j_i} \) is an ancestor of \( M_i \) in \( T \).

For example:

\[ (1) \]

\[ (2) \]

\[ (3) \]
In each case, the tree on the right is a subtree of the tree on the left: (3) and (4) are sublabelled and subordered; (5) is not sublabelled; (6) is sublabelled but not subordered. (1), (2) and (5) are connected, but not the others.
In contexts where the meaning is clear, I may omit the qualifications labelled and ordered, and refer simply to subtrees.

If \( N \in T \) then the subtree rooted at \( N \) is the subtree containing just \( N \) and all its descendants.

A frontier is a set \( F \) of nodes such that \( F \cap B \neq \emptyset \) for every branch \( B \). A subtree consisting of a frontier and all ancestors of nodes in the frontier is an initial subtree.

If \( \{ N_1, \ldots, \} = S \) is any set of nodes then the root of the smallest tree containing all of \( N_1, \ldots \) is the paterfamilias of \( S \).

In a binary tree, every node has either zero or exactly two sons.

We will call a function from trees to \( 2^{\mathcal{N}_0} \) a complexity measure if it treats isomorphic trees identically and if it is monotonic under subtrees, i.e. \( f(T_1) \leq f(T_2) \) if \( T_1 \) is a subtree of \( T_2 \). Two especially interesting measures are size, the number of interior nodes in the tree; and level.
or depth, the number of interior nodes in the longest branch of the tree.

Many of the results proved below have a simple intuitive content and are basically easy to understand, but some are obscured by the elaborate notation necessary to refer to trees. As an aid to the reader and to myself, I illustrate several of them with diagrammatical representations of trees, which should be more or less self-explanatory. It should, however, be pointed out that these diagrams give a totally misleading impression of the size of trees and especially of the rate at which their size increases with the increased depth: They suggest a linear increase of size with depth:

\[
\begin{align*}
\text{\begin{center}
\begin{tikzpicture}[scale=0.5]
\node (root) at (0,0) {\textbullet};
\node (left) at (-1,-1) {\textbullet};
\node (right) at (1,-1) {\textbullet};
\draw (root) -- (left);
\draw (root) -- (right);
\end{tikzpicture}
\end{center}}
\end{align*}
\]

whereas, of course, size may increase exponentially with depth in general.

For trees which are finitely branching, König's lemma holds.
Lemma 4.3. (König)

A finitely-branching tree $T$ has finite size iff every branch is finite.

Thus if a finitely-branching tree is infinite then it contains an infinite branch, so that $\text{size} = \text{depth} = \aleph_0$; and if not, then both size and depth are finite.

4.1.2. Tree surgery.

Let $T$ be a tree, $N \in T$, and let $M$ be a descendant of $N$ in $T$. Then the result of deleting the branch $B_{NM}$ is the subtree $T'$ of $T$ in which contains all descendants of $M$, but no other descendants of $N$, and in which $N \leq_{T} N_2$ if $M \leq_{T} N_2$, and otherwise $N_1 \leq_{T} N_2$ iff $N_1 \leq_{T} N_2$ and $N_1, N_2 \in T'$.
T' inherits fan orderings and labellings from T in the obvious way.

Let S be the set of subtrees of T rooted at the members of \( \mathcal{O} \) in T. \( M \in T' \) for some \( T' \in S \); let \( T'' \) be the result of removing all descendants of \( M \) from \( T' \), and let

\[
S' = \{ T \in S: T \not\in T' \} \cup \{ T'' \}.
\]

Then \( S' \) will be called the residue of the deletion. If T is ordered, then the residue is the obvious sequence of subtrees.

Now, conversely, let T be a tree, \( N \in T \), and let S be a set of trees disjoint from one another and from T, and let \( M \) be a tip of some \( T' \in S \). Then the result of inserting S between N and M is the tree \( T' = T \cup (U_S) \) in which

\[
M <_{T'N_1} \text{ iff } N <_{T'N_1}
\]

\[
N <_{T'N_1} \text{ iff } N_1 \text{ is the root of some } T'' \in S
\]

and otherwise \( N_1 <_{T'N_2} \text{ iff } N_1 <_{T''N_2} \) and \( N_1, N_2 \in T'' \).
If T and the members of S are ordered, and S is a sequence of trees, then T' is ordered in the obvious way.

Now let T be a tree, N, N' ∈ T and M a descendant of N. Then the result of moving the branch $R_{NM}$ to N' is the result of deleting the branch $R_{NM}$ from T and inserting the residue between N' and M:

N may be a descendant of N' in T, (but N' must not be a descendant of N):
More simply, if $N$ is a tip of $T$ and $T'$ is disjoint from $T$ then the result of adding $T'$ at $N$ is the tree $T'' = T \cup T' - \{N_0\}$, where $N_0$ is the root of $T'$, and $N_1 <_{T''} N_2$ iff $N_1 = N$ and $N_0 <_{T'} N_2$, or otherwise $N_1 <_{T'} N_2$ or $N_1 <_{T''} N_2$:

$T''$ is ordered (labelled) iff both $T, T'$ are.

Note that $N_0$, and hence its label, is missing in $T''$.

4.1.3 Predicate Calculus miscellanea.

A literal $L$ is an atom or the negation of an atom. The complement $\overline{L}$ of $L$ is the literal defined by: if $L = A$ then $\overline{L} = \neg A$;

if $L = \neg A$ then $\overline{L} = A$.

A clause is a sequence of literals. (This definition differs from the usual one. Treating
clauses as sequences rather than sets is in line with the discussion in Chapter 3.) We will often regard a clause as representing the universal closure of the disjunction of its literals. A set of clauses will similarly often be regarded as the conjunction of its members. Thus a set of clauses represents a statement in conjunctive normal form.

I will assume, throughout this chapter, that the input expression to the TP system is in conjunctive normal form, and that the set of initial structures is a corresponding set of clauses. The set of clauses will be called the clausal form of the input statement. (This assumption is almost universal in the literature on TP systems. I have followed it from a desire for simplicity and the easy life rather than from any deep belief in the usefulness of clausal form. The reduction assumed in section 3.5 above was, I believe, harmless. But the final step to conjunctive normal form is pernicious. The distribution of & over \( \lor \) leads to a proliferation of atom occurrences and much duplication of subexpressions, and breaks up
meaningful contexts into unrelated clauses. Most of the present theory can be transferred without great difficulty to the non-clausal case, as my student D. Wilkins (1973) has shown.

Corresponding to this assumption and the interpretation of clauses as disjunctions, a proof will mean henceforth a demonstration of unsatisfiability. If we interpreted clauses as conjunctions, sets of clauses as disjunctions, and reversed the roles of universal and existentially bound variables, then the present theory would support the positive notion of proof; but disjunctive normal form is, as a matter of pragmatic fact, less convenient as an expressive vehicle.

Note that clauses, considered as structures, have all their variables bound in them, and satisfy A1-A4 of section 3.5. (Selectors are 'first', 'second', etc.)

An assignment (or truth-assignment) $\mathcal{A}$ to a set or sequence $S$ of atoms is a set of literals
containing, for each \( A \in S \), exactly one of \( A \) or \( \neg A \).

An assignment represents an assignment of truth or falsity to each atom in \( S \) in the obvious way.

\( \bar{\alpha} \) is the assignment \( \{ \neg L : L \in \alpha \} \).

Let \( S \) be a set of clauses. The Herbrand universe \( H(S) = H \) of \( S \) is the set of terms which can be constructed from the function and constant symbols in \( S \) and variables. Thus all variables are in \( H \); all constants in \( S \) are in \( H \); and if \( t_1, \ldots, t_n \) are in \( H \) and \( f \) occurs in \( S \) then \( f(t_1, \ldots, t_n) \) is in \( H \).

The Herbrand base \( \hat{H}(S) = \hat{H} \) of \( S \) is the set of all atoms \( P(t_1, \ldots, t_n) \) where \( P \) occurs in \( S \) and \( t_i \in H(S) \).

Any interpretation \( M \) of \( S \) defines a truth-value for every atom in \( \hat{H}(S) \) and hence defines an assignment \( \alpha_M \) to \( \hat{H}(S) \). Such an assignment will be called a Herbrand interpretation. Any Herbrand interpretation \( \alpha \) defines a conventional interpretation \( M_\alpha \). The universe of \( M_\alpha \) is \( H \); a constant symbol denotes itself, a function symbol \( f \) denotes the function whose value for the arguments \( t_1, \ldots, t_n \) is the term \( f(t_1, \ldots, t_n) \)
and a predicate symbol $P$ denotes that predicate on $H$ which holds of $t_1, \ldots, t_n$ just in case $P(t_1, \ldots, t_n)$ is in $\mathcal{A}$. A clause $C$ is true in $M_A$ iff some literal in every instance $C \setminus \lambda$ over $H(S)$ is true; otherwise $C$ is false in $M_A$. A set $S$ of clauses is false in $M_A$, iff some member is false in $M_A$, otherwise $S$ is true in $M_A$. Thus, the empty clause $\langle \rangle$ is false in every $M_A$, while the empty set of clauses is true in every $M_A$. Clearly, $\mathcal{A}_{M_A} = \mathcal{A}$ so that $S$ has a model $M$ iff it has a Herbrand model $\mathcal{A}_{M}$, i.e. $S$ is unsatisfiable iff it has no Herbrand model.

Let $S$ be a set of clauses and $\mathcal{A}$ be an assignment to $B \subseteq A = ||S||$. Then the Davis-Putnam set of $S$ w.r.t. $\mathcal{A}$, $DP(S, \mathcal{A})$, is the set obtained by throwing away all clauses containing a literal in $\mathcal{A}$, and removing from the remainder all literals whose complements are in $\mathcal{A}$:

$$DP(S, \mathcal{A}) = \{ C : \exists D \in S. \ D \cap \mathcal{A} = \emptyset \land C = D - \overline{\mathcal{A}} \}.$$
4.2 Semantic Trees

4.2.1 Basics

Let \(<T, s> = T\) be a tree labelled with sets of literals. \(T\) is a semantic tree if:

(i) \(Ns = \emptyset\) iff \(N = N_0\)

(ii) If \(F\) is a fan in \(T\) then
\[ \forall_{N \in F} (\& Ns) \]
is a minimal tautology.

(iii) If \(B\) is a branch of \(T\) then
\[ \bigcup_{N \in B} Ns = B \]
is an assignment to a set of atoms.

By 'minimal' here is meant that \(\forall_{N \in S} (\& Ns)\), for any proper subset \(S \subset F\), is not tautologous. We do not assume that \(F\) is finite, or that \(B\) is finite. Notice that conditions (i) and (ii) guarantee that every \(F_N\) contains at least two nodes, unless \(N\) is a tip.

If \(T\) is a semantic tree in which \(B\) is an assignment to the set \(A\) of atoms, for every branch \(B\), then \(T\) is a semantic tree for \(A\).
Before proceeding we develop some useful properties of semantic trees. The set $N_\text{s}$ will be called the assignment at $N$. The set $B$ will be called the assignment on the branch $B$.

**Lemma 4.4** Let $N$ be a node in a semantic tree, and $L \in N_\text{s}$. Then for some brother $M$ of $N$, $\overline{L} \in M_\text{s}$.

**Proof** Suppose not. Let $\gamma$ be a truth-assignment to all the atoms except $\| L \|$. The assignment $\gamma \cup \{ \overline{L} \}$ falsifies $\& N_\text{s}$ and hence must make some other $\& M_\text{s}$ true. But this $\& M_\text{s}$ does not contain $\overline{L}$, by hypothesis, and hence is also made true by $\gamma \cup \{ \overline{L} \}$. Thus $M \in (\bigvee_{N \neq N} \overline{N}) \ (\& M_\text{s})$ is tautologous, contrary to the minimality of the original tautology.

QED.

**Lemma 4.5** Let $T$ be a semantic tree, $N \in T$. Then

(i) $N_\text{s} \cap B_N \neq \emptyset$ (ii) $N_\text{s} \cap \overline{B}_N \neq \emptyset$

**Proof** (i) No assignment contains a literal and its negation.

(ii) Suppose not: say $L \in N_\text{s}$ and $L \in \overline{B}_N$.

By lemma 4.4, $\overline{L} \in M_\text{s}$ for some brother $M$ of $N$. 

Now, \( B^*_H = B^*_H \) and \( (B^*_H \cap \overline{M_S}) \neq \emptyset \), contradicting (1). QED.

**Lemma 4.6** Let \( T \) be a semantic tree for \( A \), and \( \mathcal{A} \) be an assignment to \( A \). Then \( \mathcal{A} = \B \) for some branch \( B \) of \( T \).

**Proof** Let \( \mathcal{F} \) be a fan in \( T \). Since \( \forall N \in F (N \& N_S) \) is a tautology, \( \mathcal{A} \) makes at least one \( N_S \) true. That is, in every fan \( \mathcal{F} \) there is at least one \( N \) such that \( N_S \subseteq \mathcal{A} \).

Consider the set of nodes \( N \) such that \( M_S \subseteq \mathcal{A} \) for every \( M \in B_N \); this set is a subtree of \( T \), every branch of which is a branch of \( T \). Let \( B \) be such a branch: \( B \subseteq \mathcal{A} \) by construction; but \( B \) is an assignment to \( A \) since \( B \) is a branch of \( T \); hence \( B = \mathcal{A} \).

QED.

A semantic tree for \( A \) thus is a complete survey of all assignments to \( A \).
4.2.2 Failure and closure

Let $T$ be a semantic tree and $C$ be a clause. We will say that $C$ fails at a node $N$ of $T$ if there is a substitution $\sigma$ such that $\overline{C\sigma} \subseteq B_N$, i.e. $C\sigma$ is false in $B_N$.

If $T$ is a semantic tree for $\hat{H}(S)$ then for every branch $B$, $B$ is a Herbrand interpretation and it is easy to see that $C$ has the value false in this interpretation iff $C$ fails at some node of $B$.

This gives the important

Theorem 2. Let $S$ be a set of clauses and $T$ be a semantic tree for $\hat{H}(S)$. Then $S$ is unsatisfiable iff every branch of $T$ contains a node at which some clause $C \in S$ fails.

Proof. Suppose $S$ is unsatisfiable, and $B$ is a branch of $T$. $\overline{B}$ is an assignment to $\hat{H}(S)$ and hence a Herbrand interpretation. $S$ is false in this interpretation: hence some ground instance $C\theta$ of a clause $C \in S$ is false in $\overline{B}$: that is, $\overline{C\theta} \subseteq B$. But $\overline{C\theta}$ is finite since $C$ is finite, and hence $\overline{C\theta} \subseteq B_N$ for some $N$. 
Now suppose $S$ is satisfiable: let $M$ be a model of $S$. $M$ determines a truth-value for every atom in $\hat{H}(S)$ and hence determines a Herbrand model $A$ of $S$. $A = B$ for some branch $B$ of $T$. Hence no clause fails at any node of $B$; for if it did, it, and hence $S$, would be false in $B$. QED.

This theorem makes precise the intuitions described earlier. It has an immediate consequence, Herbrand's theorem (more properly called Skolem's theorem):

**Corollary (Herbrand/Skolem).** A set $S$ of clauses is unsatisfiable iff some set $S' = \{ C \theta : C \in S \}$, of ground instances over $\hat{H}(S)$ of clauses in $S$, is contradictory.

If some set $S'$ is a contradiction then $S$ is unsatisfiable. Now suppose $S$ is unsatisfiable. Let $T$ be a finitely-branching semantic tree for $\hat{H}(S)$. Every branch of $T$ contains a node $N$ where a clause $C_N$ fails. Let $S'$ be the set of corresponding instances $C \theta$. Consider the
subtree $T'$ of $T$ consisting of these nodes and all nodes above them. Every branch of $T'$ is finite, and $T'$ is finitely-branching: hence $T'$ is finite (König), and so $S'$ is finite. Clearly, $S'$ is contradictory by Lemma 3.

QED.

This proof is due to J.A. Robinson (1963)

Suppose a clause $C \in S$ fails at a node $N$ in a branch $B$ of $T$, then we will say that $B$ is closed (for $S$). If every branch of $T$ is closed (for $S$) then $T$ is closed (for $S$).

Let us define a closure to be a pair $\langle T, f \rangle$ where $T$ is a semantic tree in which every branch is finite, and $f$ is a total function from tips of $T$ to clauses, such that $f(N)$ is false in $B_N$. We may, when no confusion results, identify $\langle T, f \rangle$ with its underlying tree $T$, and hence with the set of its nodes.

If a semantic tree $T$ is closed for $S$ then some initial subtree $T'$ in which every branch is finite is also closed for $S$: that is, there is a closure $\langle T', f \rangle$ where $f(N)$ is an instance of a clause in $S$. 
A closure for a set \( S \) of clauses can be regarded as a proof of the statement of which the set \( S \) is the clausal form. It seems appropriate to try to find normal forms for such proofs.

The theorem above shows that if one semantic tree for \( S \) is closed then all are. On the face of it, this seems as strong a normal form as possible: for we can set out to build a semantic tree in any way we please, and, provided only that we check systematically for failure, success is guaranteed.

This simple scheme is however unworkable for two reasons: it would generate wastefully large closures; and it would not be liftable. The tree construction must be in terms of liftable operations, for otherwise we are in effect searching the infinite instantiation choice range.

More realistically, we can hope to find classes of closures which are defined by sensible liftable operations. Closures in such a class can be regarded as being in normal form. Before discussing examples, however, we investigate ways in which closures can be wastefully large.

In a closure \( \langle \langle T, s \rangle, f \rangle \), an occurrence of a literal in the assignment at a node in \( M \) may or may not contribute to the failure of \( f(N) \), for some \( N \) below \( M \). Suppose that \( N \) is a tip, \( M \) above \( N \), and \( L \in Ms \). Then we will say that \( L \) is relevant.
to $N$ at $M$ if $\overline{L} \in f(N)$: otherwise irrelevant to $N$ at $M$. If $L \in Ms$ is relevant to some tip $N$ below $M$ then $L$ is relevant at $M$, otherwise irrelevant at $M$.

If a closure $\langle \langle T, s \rangle, f \rangle$ contains a node $M$ at which $L$ is irrelevant then $L$ can be deleted from $Ms$ without affecting the failure of any $f(N)$. But the resulting tree will not be semantic. We will show that a certain subtree $T'$ of $T$ yields a closure $\langle T', f' \rangle$. First we define the exact relationship between these two closures.

Let $\langle \langle T, s \rangle, f \rangle, \langle \langle T', s \rangle, f' \rangle$ be closures such that

(i) $T'$ is a subtree of $T$;
(ii) $Ns' \subseteq Ns$ for all $N \in T'$;
(iii) for every tip $N$ of $T'$, there is a tip $M$ of $T$, below $N$ in $T$, such that $f'(N) = f(M)$;

then we will say that $\langle \langle T', s \rangle, f' \rangle$ is a subclosure of $\langle \langle T, s \rangle, f \rangle$.

If $T'$ is a proper sublabelled tree of $T$ then $\langle T', f \rangle$ is a proper subclosure of $\langle T, f \rangle$. If a closure has a proper subclosure then it is prunable.
Notice that if \( N \) is a tip of both \( T \) and \( T' \) then \( f(N) = f'(N) \); and otherwise, if \( N \) is a tip of \( T' \) but not of \( T \) then \( f \) is undefined on \( N \). We can thus assume that \( f = f' \), by harmlessly extending the domain of each function to include that of the other. This convenient abuse of notation will be assumed from now on.

For example:

\[ f(5) \text{ undefined} \quad \quad f'(5) = f(9) \]

Here, clauses which fail at nodes are written...
in square brackets. We will use this convention throughout the following.

The closure on the right is a proper subclosure of that on the left. This example illustrates most of the phenomena associated with the subclosure relation. Notice that the closure on the left, unlike the other, contains irrelevant assignments (at nodes 3, 6, 9, 10, 12 and 13).

The connection between prunability and relevance is established by the

Prune Lemma. Every closure has a subclosure which contains no irrelevant assignments.

Proof. Let \( \langle \langle T, s \rangle, f \rangle = \mathcal{C} \) be a closure. Define the labelling function \( s^o \) by:

\[
Ns^o = \{ L \in Ns : L \text{ is relevant at } N \}.
\]

Clearly \( Ns^o \subseteq Ns \). However, \( \langle T, s^o \rangle \) may not be a semantic tree.

Let \( \emptyset \) be a fan in \( T \), then

\[
\forall N \in \emptyset \quad \lor Ns^o \quad \text{is a tautology},
\]

since \( \lor N \in \emptyset \quad \& Ns \) is and \( Ns^o \subseteq Ns \).
Therefore for some nonempty subset $\mathcal{P}^1$ of $\mathcal{P}$,

$$\forall N \in \mathcal{P}^1$$

is a minimal tautology.

The mapping $\mathcal{P} \to \mathcal{P}^1$ defines (by lemma 4.2) a connected subtree $T^1$ of $T$. $<T^1, s^0>$ may not be a semantic tree, since some $\mathcal{P}^1$ may be a singleton $\{N\}$ where $N s^0 = \emptyset$. Now define $\mathcal{P}^2 = \emptyset$ if $\mathcal{P}^1$ is a singleton, $\mathcal{P}^2 = \mathcal{P}^1$ otherwise. The mapping $\mathcal{P}^1 \to \mathcal{P}^2$ defines (by lemma 4.3) a subtree $T^2$ of $T^1$. $<T^2, s^0>$ is a semantic tree, by construction.

Define $f'$ on tips of $T^2$ as follows. Every tip $N$ of $T^2$ is the ancestor of a unique tip of $T^1$, by construction. Every tip of $T^1$ is a tip of $T$. Define $N^-$ to be the unique tip of $T$ of which $N$ is an ancestor in $T^1$, and define $f'(N) = f(N^-)$.

Clearly, $f'(N)$ fails at $N$ in $<T^2, s^0>$, so that $<<T^2, s^0>, f'> = \mathcal{C}^1$ is a subclosure of $\mathcal{C}$. All assignments which are irrelevant in $\mathcal{C}$ are removed in $\mathcal{C}^1$, so that $T^2$ is a proper subtree of $T$ just in case $\mathcal{C}$ contains irrelevant
assignments. If $C^1$ contains irrelevant assignments, repeat the construction to give $C^2$, etc. Eventually a $C^n$ must contain no irrelevant assignments, since $T$ is finite.

QED.

Tighter normal forms are obtained if we invoke the prune lemma. For, since any closure which contains irrelevant assignments can be pruned to yield one which does not, it is natural to regard the latter as a normal form of the former.

It had, however, better be the case that such pruning preserves the class of earlier normal forms. For example, suppose we decide to consider only closures which were built from fans of the form

![Diagram](attachment:image.png)

then we would have to allow irrelevant assignments. For such a fan may be pruned, as in the example earlier, to a binary fan.
One very common set of closures which is preserved under prunings is that of all binary closures, i.e. those whose underlying semantic tree is binary. In subsequent sections we will consider this particular class in some detail. Binary closures are particularly natural because any class of closures which is closed under pruning must include binary closures: the only unprunable closure of \( \{ <A>, \neg A \} \) is:

```
\[ \overline{A} \quad \begin{array}{c}
A \\
A
\end{array} \]
```
4.3 Searching for closures

In the following sections, various methods (collections of constructive operations) of building closures from sets of clauses will be described.

Ideally, such a method would have the following properties.

(P1) It is liftable.
(P2) It generates only unprunable closures.
(P3) The search space is irredundant.
(P4) The search space has a low branching rate.

All the systems considered satisfy P1.

Most satisfy P2. (The only exception is Prawitz' system, described in 4.5.1 below.)

None of them satisfies P3, but they can sometimes be compared as to the degree of redundancy in their search spaces. The descriptions in this chapter will always be in terms of very rich structures (partial closures). Thus there will
be little or no simple redundancy but usually much structural redundancy. I will indicate various collapses in which the structural redundancy has been converted into simple redundancy: often, these will be isomorphic to well-known systems in the literature.

Condition P4 is deliberately vague, but one can compare some systems with others on this dimension also. The two broad classes of top-down and bottom-up methods (see below) have branching rates which are similar within each class, but the top-down branching rates seem to be an exponential order of magnitude less than the bottom-up branching rates.

Since each system is liftable, all the descriptions of the various systems will be for the propositional case only. This greatly simplifies the discussion. (Indeed, this simplicity was the motivation for getting over the notationally
horrible lifting theory once and for all in section 3.5.) In particular, I will, from time to time draw attention to various possible deletion regimes, and assume that they are lifted appropriately to the PC level, as discussed in section 3.5.3 earlier. Completeness follows immediately for the lifted systems, by the argument in section 3.5: Herbrand's theorem (for clauses) was established above.

In keeping with this restriction to the propositional calculus, we will now modify the definition of fails at: C fails at N will henceforth be synonymous with C is false in $\mathcal{B}_N$, i.e. $\overline{c} \notin \mathcal{B}_N$.

Each set of constructive operations is defined on some set of structures, representing partly-completed closures. There are several kinds of such structures.

Let T be a semantic tree and f be a partial function from tips of T to clauses such that some subclause of f(N) fails at N. Then $\langle T, f \rangle$
is a partial closure or pclosure. If \( f \) is total on tips of \( T \) then \( \langle T, f \rangle \) is a pclosure of type A; if \( f(N) \) fails at \( N \) wherever \( f \) is defined then \( \langle T, f \rangle \) is a pclosure of type B. A pclosure which is of type A and type B is a closure. If \( \langle T, f \rangle \) is a pclosure and \( f(N) \) is defined then we define \( f(N) \) to be the subclause of \( f(N) \) which fails at \( N \). Most of the systems described below will be defined on pclosures of type A or of type B.

The discussion of relevance and prunability in section 4.2.2 above applies without change to pclosures of type A. For pclosures of type B it is convenient to redefine 'relevance' so that \( L \in M_s \) is relevant at \( M \) if either \( M \) is a tip and \( f(M) \) is undefined, or \( L \) is relevant by the earlier definition. We thus allow the assignment at a tip to be relevant if that tip has no associated failing clause. This corresponds to the case where that tip will be an interior node in the final closure under construction: cf. section 4.5 below. With this slight modification the prune lemma applies to pclosures of type B also. For example, the following are prunable pclosures of types A and B respectively, with
their associated unprunable prunings.

Any method of building new closures from old, starting with clauses, must be based on some interpretation of clauses as closures, and some rules for building new trees from old ones. In the bottom-up methods described in section 4.4 below, we build trees, starting from the tips, by adding new nodes above appropriate sets of already existing trees. The closures which naturally correspond to this system are those of type A. In the top-down methods discussed in section 4.5, we build trees, starting at the root by adding trees corresponding to clauses below
the tips of already existing trees. Here the natural pclosures arising are those of type $B_i$.

It is convenient in what follows to occasionally regard semantic trees as ordered trees, so that each fan $\mathcal{F}_N$ is a sequence. All the other definitions and results above go through entirely unchanged: the prune lemma in particular still applies.

Any pclosure, considered as a structure, has its variables bound in it and satisfies (A1)-(A4). Selectors can be, for example, sequences of integers $\langle n_1, n_2, \ldots \rangle$ acting as a 'route map' through the tree: take the $n_1 \text{th}$ son of the root, the $n_2 \text{th}$ son of that, $\ldots$ etc. If the sets labelling nodes are not singletons then the lexicographic ordering can be used to define the first, second etc., literals at a node.
4.4 Bottom-up methods

Throughout this section, 'pclosure' will be understood to mean 'pclosure of type A', unless stated otherwise.

If C is a clause, let C' be the pclosure whose semantic tree has a single node N labelled with ∅ and where f(N) = C, so that f(N) = <>. Bottom-up methods treat initial clauses C as their pclosure counterpart C'. The node of C' becomes, as the construction proceeds, a tip of subsequent pclosures and eventually a tip of a final closure at which C fails.

In a pclosure, there are literals in clauses at tips which are not yet false. Let C be a pclosure and N a tip: I will call the set \{ L \in f(N) : L \in \tilde{f}(N) \} the excess at N, denoted by \text{ex}(N). The excess of C, \text{ex}(C) is \bigcup_{N \in C} \text{ex}(N). Thus \text{ex}(C') = \{ L : L \in C \}, and \text{ex}(C) = \emptyset \iff C is a closure.

The bottom-up methods described in this section all operate in the following way. A constructive
operation applies to a set \( \{ C_1, \ldots, C_n \} \) of pclosures, \( C_1 = \langle T_1, f_1 \rangle \), if there are certain subsets \( S_i \) of \( \text{ex}(C_i) \) such that \( \bigvee_i \&(S_i) \) is a minimal tautology of a certain form. Then the corresponding result is the pclosure \( C = \langle T, f \rangle \) where \( T \) is obtained by adding \( T_i \) below the tip \( N_i \) of the semantic tree consisting of a new root \( N \) and the fan \( \forall N = \{ N_1, \ldots, N_n \} \) where \( N_i \) is labelled with \( \&(S_i) \); and where \( f(N) = f_i(N) \) when \( N \) is a tip of \( T_i \).

\[
\begin{array}{ccc}
\{ 
\begin{array}{c}
C_1 \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\end{array}
\} \\
\cdots \\
\begin{array}{c}
C_n \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\forall S_i \\
\quad \\
\quad \\
\quad \\
\quad \\
\end{array}
\begin{array}{c}
\forall S_n \\
\quad \\
\quad \\
\quad \\
\quad \\
\end{array}
\end{array}
\]

Clearly, \( C \) is a pclosure, \( f_1(N) \subseteq f(N) \) and \( \text{ex}(C) = \bigcup_i (\text{ex}(C_i) - S_i) \).

Any such \( R \) is sensible, and it is liftable if the selection of subsets \( S_i \) and the choice of form of tautologies are both liftable, i.e. if an inference is 'R-warranted' whenever some instance of it is: for, given the sets \( S_i \), the requirement that \( \bigvee_i \&(S_i) \) be a minimal tautology is
stateable in terms of a linkage between selectors applied to the $C_1$. We will also assume that the class of legal tautologies is closed under prunings (in the obvious sense), in line with the discussion in section 4.3 above.

Any such $R$ determines a corresponding class of pclosures which contains all one-node pclosures and all pclosures in which the minimal tautology at every fan is such as would sanction an application of $R$. If $C = \langle \langle T, s \rangle, f \rangle$ is any such pclosure and $N \in T$ then the pclosure $C_N = \langle \langle T_N, s' \rangle, f \rangle$ where $T_N$ is the subtree rooted at $N$, $Ns' = \emptyset$, and $Ms' = Ms$ for all other $M \in T_N$, is also a member of the class. Note that: $\text{ex}(C_N) = \mathcal{P}_N \cup \text{ex}(C)$ where $\mathcal{P}_N$ is the branch in $T$, $C_N = C$ iff $N$ is the root of $T$; if $N$ is a tip of $T$ then $C_N = (f(N))'$; and $Ns \subseteq \text{ex}(C_N)$, since $C$ is unprunable.

There is an obvious 1:1 correspondence between derivations from one-node pclosures relative to $R$, and unprunable pclosures $C$ in the class. $\mathcal{D}$ corresponds to its result, and $C$ corresponds to
the derivation obtained by replacing each fan \( f_N \) in \( C \) by the edge \( \langle f_N, M, N \rangle \) where \( M \) is a new (body) node labelled with \( R \), \( N \) is labelled with \( C_N \) and each \( M \in f_N \) is labelled with \( C_M \). We assume that inputs to polyedges are sets rather than sequences, in accordance with the symmetry of \( R \). Clearly, \( C \) is the result of this derivation.

Thus the search space from one-node pclosures relative to \( R \) is isomorphic to its derivation space and has therefore no simple redundancy. Unfortunately it has rather a lot of structural redundancy, as we will shortly see.

The completeness of all bottom-up methods follows from the following Lemma 4.4.1. Let \( C = \langle T, s \rangle, f \rangle \) be an unprunable pclosure in the class corresponding to \( R \), and let \( F \) be a frontier of \( T \). Then there is a derivation relative to \( R \) of \( C \) from \( \{ C_M : M \in F \} \).

Proof. Let \( T' \) be the initial subtree above \( F \). The proof is by induction on the size of \( T' \). If \( T' \) is a single node \( N \) then \( N \) is the root of \( T \), so \( C_N = C \). Otherwise, \( T' \) contains an interior
node N such that every \( M \in \mathcal{F}_N \) is a tip (for otherwise, there is an infinite branch of interior nodes \( M \) with an interior node in \( \mathcal{F}_{M'} \)). By construction, \( R \) applies to \( \{ C_M : M \in \mathcal{F}_N \} \), with \( C_N \) as a result. Thus there is a (single edge) derivation \( \sim \) of \( C_N \) from \( \{ C_M : M \in \mathcal{F}_F \} \) relative to \( R \). Let \( F' = (F - \mathcal{F}_N) \cup \{ N \} \): then the initial subtree above \( F' \) is a proper subtree of \( T \), so by induction hypothesis there is a derivation \( \sim' \) of \( C \) from \( \{ C_M : M \in \mathcal{F}_F \} \). Let \( D \) be the derivation obtained by identifying the result node of \( E \) with the input node of \( D' \) labelled with \( C_N \). Then \( D \) is a derivation of \( C \) from \( \{ C_M : M \in \mathcal{F}_F \} \).

QED.

Now, let \( F \) be the set of tips of a closure \( C \) for a set \( S \) of clauses: then by the lemma, there is a derivation relative to \( R \) of \( C \) from \( S' = \{ C' : C \in S \} \); this is the completeness result we required.

However, the lemma shows that any pclosure of the class corresponding to \( R \) for \( S \) can be obtained by using \( R \). Thus, a complete search space relative to \( R \) has far more derivations in it than are needed.
4.4.1 Resolution methods

R is defined only upon the sets \( \text{ex}(C_i) \). The rest of the structure of \( C_i \) is irrelevant to R. Moreover, the recognition of a terminal structure—a closure—can be made only upon \( \text{ex}(C) \), since \( C \) is a closure iff \( \text{ex}(C) = \emptyset \). The complete search space can therefore be collapsed by identifying pclosures with the same excess. However, the structures in the space would be sets if we followed this course directly, violating A4. To preserve the lifting theory of section 3.5 I will, rather, map pclosures onto clauses.

Let \( \text{cl} \) be a map from pclosures to clauses such that if \( C \) is a one-node pclosure then \( \text{cl}(C) \) is the clause labelling the single node of \( C \), and for any \( C \), \( \{ \text{cl}(C) \} = \text{ex}(C) \). Clearly we can define such a \( \text{cl} \) recursively on pclosures as follows: let \( \{ N_1, \ldots, N_n \} \) be the fan at the root of \( C \), suppose \( \text{cl}(C_{N_i}) \) is defined, then \( \text{cl}(C) = k(C) \), where \( C \) is the result of concatenating \( \text{cl}(C_{N_i}) \)-\( N_i \)'s and \( k \) is some function which deletes certain duplicated occurrences of literals: \( k \) may be the identity function. (This depends on some definite policy for ordering fans
of semantic trees, and I assume that this is done in some arbitrary but systematic way.) If $C = \langle \langle T, s \rangle, f \rangle$ is a closure then $cl(C_N)$ fails at $N$. In particular, if $N$ is the root then $cl(C_N) = \langle \rangle$

Now, let $R$ be the constructive operation on clauses defined by $R_c(C_1, \ldots, C_n) \equiv \exists C. R(C_1, \ldots, C_{n-1}, C) \land C_n = cl(C)$ We will call $R_c$ a resolution operation corresponding to $R$. To any derivation $\mathcal{D}$ relative to $R$, there clearly corresponds a derivation $\mathcal{D}'$ relative to $R_c$, in which non-body nodes are labelled with $cl(C)$ where $C$ labels a corresponding node in $\mathcal{D}$. If distinct $C_1$ and $C_2$ both label nodes of $\mathcal{D}$ and $cl(C_1) = cl(C_2)$ then the corresponding nodes are identified in $\mathcal{D}'$, and we arbitrarily choose one of the derivations (relative to $R_c$) of $cl(C_i)$ as the appropriate subderivation of $\mathcal{D}'$. However, there are derivations relative to $R_c$ which correspond to no derivation from $R$. Let $\mathcal{D}$ be a derivation relative to $R_c$; then there is an associated labelled tree $T$ obtained by mapping edges $\langle i(E), b(E), o(E) \rangle$ to fans so that copies of $i(E)$ are the fans at all the nodes corresponding to $o(E)$; and labelling
each copy of an input node with the key conjunction \( S_1 \) of the clause associated with the inference. This labelled tree clearly obeys all the conditions for a semantic tree except that it may contain contradictions along a branch. If we associate clauses labelling input nodes of \( D \) with the corresponding tips of \( T \) then the resulting structure \( C \) is just like a pclosure except that its underlying tree may not be semantic.

If, however, \( T \) is a semantic tree (and hence \( C \) a pclosure) then this construction yields a pclosure whose corresponding derivation \( D' \) relative to \( R \) collapses into \( D \) when we replace \( C \) by cl(\( C \)). Thus we have

**Lemma 4.4.2.** The complete search space relative to \( R_c \) from a set \( S \) of clauses properly contains the collapse, of the complete search space relative to \( R \) from \( S' = \{ C' : C \in S \} \), generated by the congruence \( C_1 \leftrightarrow C_2 \equiv (\text{cl}(C_1) \) is a permutation of \( \text{cl}(C_2) \)).

**Proof.** Obvious from the above remarks and the definition of \( R_c \).

\[ QED. \]
Notice that in this collapse, all closures map into the empty clause \(<\rangle\), dramatically illustrating the structural redundancy in the \(R\) space.

The refinement mapping which restricts the complete \(R_c\) space to the collapse of the \(R\) space will be called the **minimality** refinement of \(R_c\). In order to make this into a refinement test we would have to preserve rather more of the structure of the pclosures \(c\), viz. the sets of assignments on their branches. Clearly, \(R_c\) and the minimality refinement are liftable.

Historically, the development was in the reverse direction from the present account: resolution came first, followed by the idea of minimality, followed (in this dissertation) by the definitions of structural operations upon trees. There are other refinements of \(R_c\), incompatible with minimality, which are of some interest. One in particular (binary Pl-deduction) will be discussed below. (There are numerous other refinements of \(R_c\) rules in the literature, however, which apparently do not fit into the semantic tree framework. The most interesting of these is Boyer's method of **locking** (Boyer 1971). I have made several attempts to
incorporate locking into the present framework, but so far to no avail.)

4.4.2 Merging and factoring

The clause $\text{cl}(C)$ is supposed to represent the set $\text{ex}(C)$, but may contain duplicate occurrences of literals. The definition of $R_c$ refers to the removal of a set $S_i$ of literals from a clause $C_i$, and of course if $C_i$ contains two occurrences of some $L \in S_i$ then they must both be removed. It is convenient to represent a structural operation therefore in the form $R_c = R_1 \otimes R_2$, where $R_2$ is a deletion operation, corresponding to the map $k$ above, and $R_1$ is the case of $R_c$ in which only a single occurrence of the relevant literals in the input clauses is selected by the key linkage. There are several possible such deletion operations, each corresponding to an inductively defined $\text{cl}$ function. I will describe the two most sensible.

The first simply deletes all duplicate occurrences of literals in clauses as soon as a clause is
produced by $R_1 \circ R_2 (C_1, C_2)$ is true if $C_2$ results by removing the first duplicated occurrence of a literal in $C_1$. Applying $R_2$ to exhaustion then effectively makes clauses $cl(C)$ isomorphic to their corresponding sets $ex(C)$, since we are ignoring order. Lifting $R_c = R_1 \circ R_2$ as described in section 3.5.3, we obtain the rules $\tilde{R}_1 \circ R_2$ and $\tilde{R}_2 \circ R_2$, in which the latter is the \textit{m-factoring} rule of Kowalski (1970).

The second deletion regime is to make only such deletions as are strictly necessary in order to allow $R_1$ to function properly. Here we enrich structures slightly in order to be able to 'record' the deletions which have taken place: structures are pairs $<C_i, F_i>$ where $C_i$ is a clause and $F_i$ a set of selectors. We insist, in applying $R_1'$, that the set $S_i$ involved in the tautology is $\{ f(C_i) : f \in F_i \}$. The output is any $<C, F>$ where $C$ is the clause (without deletions) produced by $R_c$ and $F$ is a set of selectors, appropriate to $C$, such that $f_1(C) \neq f_2(C)$ for $f_1 \neq f_2 \in F$. (Of course, this assumes that the notion of a set of selectors appropriate to $C$ is meaningful.
Different bottom-up systems yield different such notions of appropriateness, as explained later.)

The deletion operation corresponding to this is \( R_2', \) where 
\[
R_2'(\langle S_1, F_1 \rangle, \langle S_2, F_2 \rangle) \text{ iff } F_1 = F_2 \text{ and } S_2 \text{ is the result of deleting some } g(S_1) \text{ from } S_1 \text{ with } g \notin F_1 \text{ but } g(S_1) = f(S_1) \text{ for some } f \in F_1.
\]
This lifts to give \( \hat{R}_1 \otimes R_2' \) and \( \hat{R}_2 \otimes R_2' \) where \( R_2' \) is (almost) distinguished-literal factoring (Kowalski & Hayes 1969) and \( R_1 \) is (almost) the distinguished-literal version of the resolution rule \( R_c. \) ('Almost' is necessary since in the usual description, based on sets, of resolution operations, all deletions are automatic. Thus the usual operations would be \( \hat{R}_1 \otimes R_2 \) and \( \hat{R}_2 \otimes R_2. \) The present account is somewhat more coherent.)

4.4.3 Minimal Binary Resolution and A-orderings

Consider the class of binary pclosures. This is closed under contractions. The corresponding form of tautologies is 
\[
\bigvee (\& \{ L \}, \& \{ \overline{L} \}),
\]
i.e. \( L \lor \overline{L}. \) The linkage condition in the corresponding structural operation \( R_b \) is thus 
\[
f_1(C_1) = f_2(C_2) \text{ where } f_1(C_1) \text{ occurs positively in } ex(C_1) \text{ and } f_2(C_2) \text{ occurs negatively in } ex(C_2).
\]
The corresponding clausal resolution operation
is Robinson's original binary ground resolution rule (Robinson 1966), and the factoring methods all have their conventional names also. In the second (distinguished-literal) deletion method, 'appropriate' sets of selectors $F$ are simply all singletons $\{ f \}$ such that $f$ is defined on the clause.

Notice again that if we lift the $pclosure$ operation $R$ rather than the clausal operation $R_c$, then minimality and irredundant factoring are automatic, but there is considerable structural redundancy.

The wastefulness in the binary $pclosure$ space, all of which, by lemma 4.4.2 above, is mirrored in the binary Resolution space, is best illustrated by the example of full sets. The full set on a sequence of atoms $\langle A_1, \ldots, A_n \rangle$ is the set of $2^n$ distinct clauses $\langle L_1, \ldots, L_n \rangle$, where each $L_i$ is $A_i$ or $\neg A_i$. Full sets are unsatisfiable. The number $P_n$ of distinct binary closures for the full set of $n$ atoms obeys the recursion equation $P_1=1$ (obviously) and $P_n=n(P_{n-1})^2$; for, we can pick any $A_i$ to label the fan at the root of a closure,
and each immediate subtree can then be constructed independently in $P_{n-1}$ ways. The solution to these equations is $P_n = \prod_{i=0}^{n-1} (n-i)^2$, which increases explosively with $n$. For example, $P_2=2$, $P_3=12$, $P_4 \approx 500$, $P_5 \approx 1\frac{1}{4}$ million and $P_9$ is more than a googol.

The size of a closure for the $n^{th}$ full set is $2^{n-1}$: in general, if $S$ is a set of clauses and $|S|$ has $n$ members, then any unprunable closure for $S$ has size no more than $2^{n-1}$. This bound is therefore also a bound upon the size of minimal Resolution derivations, although in general the Resolution derivation corresponding to a pclosure is smaller than it, since nodes may become identified in the collapse.

Now, these different closures differ only in the orderings of the assignments $\rho$ on branches of the underlying semantic tree. This suggests we could decrease structural redundancy (and hence simple redundancy in a collapse) by arranging for the constructive operation to be sensitive to this ordering, in line with the programme sketched in section 3.3 above.
This is very easy: let ≤ be any partial ordering on \( \mathbb{A} \), extended to an ordering of literals by treating each literal as its corresponding atom, and consider the class of binary closures in which \( N_1 \ll N_2 \) iff the atom in the assignment at \( N_1 \) is \( \leq \) the atom in the assignment at \( N_2 \). This class is closed under contractions, and corresponds to the refinement of \( R_b \) in which we insist that the literal \( f(\mathcal{C}) \) in the linkage condition is not \( \leq \) any other literal in \( \mathcal{E}(\mathcal{C}) \): let us call this operation \( R_\leq \).

If \( \leq \) is total, there is only a single such pclosure for any full set of clauses, which is promising. Unfortunately, \( R_\leq \) is not in general liftable, for we may have \( L_1 \leq L_2 \) but \( L_1 \lambda \neq L_2 \lambda \).

It is sufficient for liftability that \( \leq \) obey the condition: \( L_1 \leq L_2 \supset L_1 \lambda \leq L_2 \lambda \) for all \( L_1, L_2 \) and \( \lambda \). For, suppose \( R_\leq (C_1 \lambda, C_2 \lambda, C \lambda) \), with the key \( f_1(C_1 \lambda) = f_2(C_2 \lambda) \& L \), where \( L \) defines the construction of \( C \lambda \) from \( C_1 \lambda \) and \( C_2 \lambda \). Then \( f_1(C_1 \lambda) \lambda = f_2(C_2 \lambda) \lambda \), and for all \( f \neq f_1 \), \( f_1(C_1 \lambda) \lambda \neq f(C_1 \lambda) \lambda \). Hence
\[ f_1(C_1) \not= f(C_1), \] by the assumption. Thus the selection of \( f_1(C_1) \) from \( C_1 \) satisfies the ordering restriction.

Such an ordering is called an \( A \)-ordering (Slagle 1967, Kowalski & Hayes 1969). Unfortunately \( A \)-orderings are very partial, since they must respect the entire lattice structure of instantiations. The corresponding clausal resolution operation is the binary \( A \)-ordering system reported in (Kowalski & Hayes 1969).

By construction, \( A \)-ordering is compatible with minimality. However, an \( A \)-ordered pclosure may not be the smallest pclosure for \( S \): not every pclosure can be constructed by \( R \leq \). The size of smallest minimal \( A \)-ordered derivations is of course bounded by the upper limit \( 2^n - 1 \) on the size of minimal derivations.

4.4.4 \( M \)-Clashes

Let \( M \) be an interpretation for the vocabulary of a set of clauses. \( M \) determines the Herbrand interpretation \( \mathcal{C}_M \). A semantic tree will be
called an \textit{m-clash tree} if every fan has the form

\[
\begin{array}{c}
N_0 \\
\& (L_i) \\
N_1 \ldots N_n
\end{array}
\]

where \( \{L_1, \ldots, L_n, \ldots\} = \alpha_m - \overline{B}_N \). Thus \( N_0 \) is a tip since \( B_{N_0} \) is an assignment to \( \{ ||L|| : L \in \alpha_m \} \). The (tip) node \( N_0 \) is called a \textit{nucleus node}, the nodes \( N_1, \ldots, N_n, \ldots \) are \textit{satellite nodes}. \textit{m-clash} trees are usually infinitely branching: but if \( S \) is an unsatisfiable set of clauses and \( T \) is an \textit{m-clash} tree then there is an unprunable closure \( C = \langle T', f \rangle \) where \( f(N) \in S \) and \( T' \) is a finitely branching subtree of \( T \), by theorem 2 and the prune lemma. We will call \( C \) an \textit{m-closure}. Clearly, the set of \textit{m-closures} is closed under prunings.

Now, for any interior node \( N \) in an \textit{m-closure} \( C = \langle T, s \rangle, f \), \( B_N \subseteq \overline{\alpha_m} \); and in any fan \( F_N \), the nucleus node \( N_0 \) is a tip and the assignment at \( N_0 \) is a subset of \( \alpha_m \). Thus \( \text{ex}(C_N) \subseteq \alpha_m \) for \( N \) a satellite tip or interior node; and for \( N_0 \) a nucleus tip, \( \text{ex}(C_{N_0}) = \{ L : L \in \mathcal{F}_N \} = N_0 \cup C_2 \).
where \( N_0 s \subseteq \mathcal{A}_m \) and \( C_2 \subseteq \mathcal{A}_m \).

The structural operation corresponding to this class of pclosures is \( R_m \) where we have:

\[
R_m (C_0, C_1, \ldots, C_n, C) \equiv \\
\text{ex}(C_1) \cup \ldots \cup \text{ex}(C_n) \cup \text{ex}(C) \subseteq \mathcal{A}_m \\
\& (\text{ex}(C_0) \cap \mathcal{A}_m) = \& \{L_1, \ldots, L_n\} \\
\& \bar{I}_i \in \text{ex}(C_i) \\
\& \bar{C} \text{ is formed as in } R, \text{ where the key tautology is } \forall \{\& \{\bar{I}_i\}, L_1, \ldots, L_n\}.
\]

The fact, that \( \text{ex}(C) \subseteq \mathcal{A}_m \) for every conclusion \( C \) of a derivation relative to \( R_m \), seems to correspond to various heuristically plausible ways of using 'models' of a domain of discourse to prune a search space. This topic, much discussed in the TP literature, lends plausibility to the \( M \)-clash rule. Notice that, given a (single-node) pclosure \( C \), we can determine whether \( C \) is to be used as a satellite or as a nucleus, and, if the latter, then which literals are to be removed in the application of \( R_m \).
Unfortunately, again, $R_m$ is not in general liftable. For a literal $L \lambda$ may be true in $M$ but $L$ false in $M$, so that $L \lambda \in \mathcal{O}_M$ but $L \notin \mathcal{O}_M$.

We can now proceed in two ways. Firstly, by weakening the operation $R_m$ to give an $R'_m$ which insists only that some instance of the literals removed from nuclei are true in $M$. This is the course adopted in (Slagle 1967) and (Kowalski & Hayes 1969).

Now $R'_m$ is liftable, but the search space relative to $R'_m$ unfortunately contains new redundancies, in general. The second course, which I will adopt here, is to consider only interpretations $M$ such that $L \lambda \in \mathcal{O}_M \supset L \in \mathcal{O}_M$ for all $L$ and $\lambda$. It is easy to see that such interpretations are predicate letter (pl) interpretations, in which every predicate letter denotes either the wholly true predicate or the wholly false predicate. Thus the truth or falsity in $M$ of an atom $A$ depends only upon the predicate letter of $A$. If $M$ is a pl interpretation then $R_m$ is liftable, trivially. Predicate letter interpretations $M$ have the desirable property that the truth-values of atoms in $M$ are easy to compute.
The clausal resolution operation corresponding to $R_m$ is Slagle's (1967) 'semantic resolution' rule, which has Robinson's (1967) hyper-resolution as a special case. The restricted notion of interpretation used here corresponds precisely to Meltzer's (1966) 'renaming' generalisation of hyper-resolution. The deletion operations both apply here also. The first method applies directly. To apply the second (distinguished-literal) method, we distinguish between satellite and nucleus clauses. An appropriate set of selectors for a satellite clause $C$ is any singleton $\{f\}$ with $f(C)$ defined. An appropriate set of selectors for a nucleus clause is any set $\{f_1, \ldots, f_n\}$ such that for all $L \in C$, $L \in C_m$ iff $L = f_i(C)$ for some $i$, $1 \leq i \leq n$, and $f_i(C)$ is defined for all $i$, $1 \leq i \leq n$.

The size of unprunable $\mathcal{M}$-closures is bounded, just as in the binary case. An unprunable $\mathcal{M}$-closure for the full set on $n$ atoms has size $Q_n$, where $Q_1 = 1$ and $Q_n = n \cdot Q_{n-1} + n$: for the $\mathcal{M}$-closure below each satellite son of the root has size $Q_{n-1}$, and there are $n$ such satellites. (I have not
been able to find a neat solution to this recursion equation using elementary functions, but clearly,
\[ n! < Q_n < n^n. \] This is, therefore, as in the binary case, an upper limit on the size of any unprunable \( \mathcal{M} \)-closure for \( S \) where \( \| S \| \) has \( n \) elements, since any such \( \mathcal{M} \)-closure is a pruning of an \( \mathcal{M} \)-closure for a full set. And this is also therefore an upper limit on the size of minimal \( \mathcal{M} \)-clash Resolution derivations: although this time, the upper bound is not sharp since for \( n \geq 3 \) the \( \mathcal{M} \)-closure for a full set is guaranteed to collapse into a smaller minimal \( \mathcal{M} \)-clash Resolution derivation.

Tighter bounds on the sizes of \( \mathcal{M} \)-clash derivations can be obtained by decomposing \( \mathcal{M} \)-clash derivations into non-minimal binary \( \mathcal{M} \)-resolution derivations, as described below.

4.4.5 Binary A \( \mathcal{M} \)-resolution

In this section I describe a binary structural operation upon clauses which is incompatible with minimality and therefore cannot be regarded as
corresponding to any refinement of $R_b$. Nevertheless, semantic trees are a useful tool in analysing its properties. Throughout this section I will assume that the first deletion method is being used, so that clauses can be considered to 'automatically' have no duplicated occurrences of literals. The structural operation is

$$R_w(C_1, C_2, C) = (R_b(C_1, C_2, C)$$

$$
& C = c_1(C)
$$

$$
& C_1 \subseteq \alpha_m
$$

$$
& C_1 - C \leq M \text{ for all } M \in C_1 \cap C
$$

where $\alpha_m$ is a Herbrand interpretation and $\leq$ an $A$-ordering.

Let us call a clause $C$ with $C \subseteq \alpha_m$ a satellite, any other clauses a nucleus, as in the $M$-clash case above. The largest satellite subclause of a nucleus $C$ will be called its residue $re(C)$. Thus if $C$ is a nucleus then

$$\{L : L \in C_1\} = \{L : L \in re(C)\} \cup \{L : L \in C_2\}$$

where $C_2 \subseteq \alpha_m$.

**Lemma 4.4.3** If $S$ is unsatisfiable then there is a derivation $D$ of $\langle \rangle$ from $S$ relative to $R_w$.

**Proof.** Let $\langle T, s \rangle$ be the binary $A$-ordered semantic tree (i.e. $N_1 \ll N_2$ iff $N_1s \leq N_2s$), and let
\( C_s = \langle \langle T', s \rangle, f \rangle \) be the smallest unprunable closure in which \( T' \) is a subtree of \( T \) and \( f(N) \in S \).

The proof is by induction on the size of \( T' \). If \( T' \) is a single node \( N \) then \( f(N) = \langle \rangle \in S \).

Otherwise, there is a unique branch \( B \) of \( T \) with \( B \subseteq C_m \): let \( N \) be the tip of \( T \) on \( B \), let \( M \) be the brother of \( N \), and let \( T_M \) be the subtree of \( T \) rooted at \( M \). Since \( C \) is unprunable, \( \overline{Ns} \) occurs in \( f(N) \). By construction, \( f(N) \subseteq C_m \) so \( f(N) \) is a satellite, and \( \overline{Ns} \neq M \) for any other \( M \in f(N) \). Moreover, \( Ms = \overline{Ns} \) is relevant at \( M \) and therefore, for some tips \( N_1, \ldots, N_n \) of \( T_M \), \( \overline{Ms} = \overline{Ns} \) occurs in \( f(N_i) \). Thus for each \( i \), \( R_w(f(N), f(N_i), C_i) \) for some unique \( C_i \) which contains no occurrence of \( \overline{Ms} = \overline{Ns} \). Let

\[ C' = \langle \langle T', s \rangle, f' \rangle \text{ where } f'(N_i) = C_i \text{ and otherwise } f' = f. \]

Then the assignment \( Ms \) at \( M \) is irrelevant in \( C' \), and so \( C' \) can be pruned, by removing \( B_{N'M} \) from \( T' \), yielding \( C'' = \langle \langle T'', s \rangle, f' \rangle \).

By induction hypothesis, since \( T'' \) is a proper subtree of \( T' \), there is a derivation \( D' \) from \( \{ f'(N): N \text{ a tip of } T'' \} \) relative to \( R_w \) of \( \langle \rangle \). Let \( \overset{\sim}{D} \) be the result of adding the appropriate polyedge before each input node of \( D' \) labelled with \( C_i \). Then \( \overset{\sim}{D} \) is a derivation of \( \langle \rangle \) from \( S \) relative to \( R_w \).

QED.
There are analogs of Lemma 4.4.1 for the earlier clausal resolution operations which are based on an argument similar in some ways to the above (c.f. theorem 2 of Kowalski & Hayes 1969).

This proof strongly suggests that the minimality refinement is incompatible with $R_w$, and that smallest derivations relative to $R_w$ can be larger than minimal derivations. Both of these are true, see as one can find by examining full sets.

Consider the full set $S_2$ on $\{A, B\}$, with $\mathcal{C}_m = \{\overline{A}, \overline{B}\}$ and $A \leq B$. Then the construction in the proof of the lemma gives the derivation $D_2$ from the unprunable closure shown. (All body nodes are labelled with $R_w \otimes R_2$ where $R_2$ is the first deletion operation.)
This is the smallest derivation of $\langle \rangle$ relative to $R_w$. There is no minimal derivation of $\langle \rangle$ relative to $R_v$.

Now consider the full set $S_3$ on $\{A, B, C\}$ with $\mathcal{M}_n = \{\overline{A}, \overline{B}, \overline{C}\}$ and $A \leq B \leq C$. Then the construction in the proof of the lemma gives the derivation $D_3$: 

![Diagram of derivation process]
and again $D_3$ is the smallest derivation of $<>$ from $S_3$. Notice that $D_3$ has the form

$$D_2 \rightarrow D_1 \rightarrow D_0$$

where $D'_1$ is isomorphic to $D_2$. This construction iterates, so if $X_n$ is the size of $D_n$, we have $X_1 = 1$ and $X_n = 2X_{n-1} + 2^{n-1}$, which has solution $X_n = n \cdot 2^{n-1}$.

If $S$ is any set of clauses on $n$ atoms then there is a derivation of $<>$ from $S$ relative to $R_1 \bigodot R_2$ which is smaller than $X_n$, by following the construction in the lemma and pruning where necessary: thus $X_n$ is a tight upper bound on the size of a smallest derivation of $<>$ relative to $R_1 \bigodot R_2$. Notice that the ratio $X_n : S_n$, where $S_n = 2^{n-1}$ is the size of the smallest minimal derivation of $<$, tends to $n/2$ as $n$ increases. Note also that if we made derivations into trees, by allowing different nodes with the same labels,
then we would obtain a grossly misleading size measure for the derivations $D_n$; in fact we would have $X_n = (2^{n-1} + 1)X_{n-1} + 2^{n-1}$.

Let $C$ be a fully deleted nucleus clause and $L_1 \in \text{re}(C)$. Then the result $C'$ of applying $R_w$ to $C$ and a satellite $C_1$ is a satellite iff $L_1$ is the only literal which occurs in $\text{re}(C)$, and otherwise $\text{re}(C') = \text{re}(C) - \{ L_1 \}$. Let $\langle C_1, \ldots, C_n \rangle$ be a sequence of satellites such that:

$$R_w \ominus R_2(C, C_1, C_1') \& R_w \ominus R_2(C_1', C_2, C_2') \& \ldots \& R_w \ominus R_2(C_{n-1}', C_n, C_n'),$$

where $C'$ is a satellite. Then $C' \subseteq C$ and hence we have $R'_{M} \ominus R_2(C, C_1, \ldots, C_n, C')$, where $R'_{M}$ is the clausal version of $R'_{M}$. Contrariwise, given $R'_{M} \ominus R_2(C, C_1, \ldots, C_n, C')$, then there are $C_1, \ldots, C_n$ satisfying the above. Thus there is a correspondence between 'linear' derivations from a nucleus and satellites to a satellite relative to $R_w \ominus R_2$, and $M$-clash polyedges with the same labels. (This phenomenon was first pointed out by Robinson (1968)). This has two important consequences.

(a) In the $M$-clash polyedge, each satellite obeys the A-ordering refinement. Thus we have
shown completeness of the $\wedge \mathcal{M}$ clash operation of Slagle (1967): This completeness argument is somewhat simpler than the argument in (Kowalski & Hayes 1969).

(b) More importantly, since the same $\mathcal{M}$-clash corresponds to derivations from $C$ and the $C_i$ in any order, this shows that the literals in $\mathsf{re}(C)$ for a nucleus $C$ form a selectable choice range; we can arbitrarily pick a literal in $\mathsf{re}(C)$ and insist that all applications of $R_w$ use only that literal in a key. This selectability lifts to the PC level, as explained in section 3.5.4 above. Notice that the $\mathcal{M}$-clash derivation has smaller size than the binary derivation, and yet that $X_n < Q_n$ for $n > 2$, so that $X_n$ is a better bound on the size of $\mathcal{M}$-clash derivations than $Q_n$, as promised.

There is no absolute upper bound on the size of $R_w$ derivations, since we can insert arbitrarily long branches of the form:

![Diagram](image)
Minimality cannot be used to control this, and the correspondence with \( \mathcal{M} \)-clashes does not help either since the \( \Lambda \)-\( \mathcal{M} \) clash operation is also not compatible with minimality (the full set \( S_2 \) is a counterexample.). However, there is a weaker notion of minimality which can be imposed upon \( R_w \) derivations.

In the proof of the lemma 4.4.3, suppose that the tip node \( N \) on \( P \) is a son of the root of \( T' \), and say that \( N_s = L \). This will be the case iff the clause failing on \( P \) is the unit clause \( \langle L \rangle \) (assuming all deletions have been performed). Then \( L \) is minimal in the \( \Lambda \)-ordering \( \leq \) : that is, \( L \leq M \) for any \( M \) occurring in a clause labelling a tip of \( T' \) in \( C_s \). Moreover, \( L \) does not occur in the tree \( T'' \) constructed by deleting \( N \), and hence does not occur in the derivation \( D' \) of \( \langle \rangle \) from \( \{ f'(N) : N \text{ a tip of } T'' \} \).

Thus the derivation \( D \) constructed in the proof, and all its subderivations, obey the following condition: let \( N \) be an input node in \( D \) labelled with a unit clause \( \langle L \rangle \). (So that \( L \) is minimal in the \( \Lambda \)-ordering relative to the clauses labelling all input nodes of the derivation
of $N$); then $L$ does not occur in any clause labelling a node on the path from $N$ to the conclusion of $\mathcal{D}$.

It follows that if $L_1$ is minimal in the $\prec$-ordering then $L_1$ is resolved upon at most once in any branch; and hence that if $L_2$ is penultimate in the $\prec$-ordering then it is resolved upon at most twice (once before $L_1$ and once after), and so on. Thus the maximum length of a branch in a $\mathcal{D}$ obeying this condition is $1+2+\ldots+n = \frac{n(n+1)}{2}$. The way this weak minimality condition operates can be seen by examining the derivation $\mathcal{D}_3$ above.

This binary resolution system $R_w \otimes R_2$, with selectivity on residues of nuclei and this weak minimality condition, seems to be one of the most restrictive bottom-up systems in the literature. It would be most interesting to find a way of relating it to the other restrictive bottom-up system, locking.
4.5 Top-down methods

Top-down methods build semantic trees by adding tree fragments below the tips of partially constructed semantic trees, rather than above the root. Typically, these tree fragments are composed from input clauses or parts of input clauses.

The derivation of a pclosure in a top-down system is thus typically a linear sequence of additions of clauses or clause fragments to a pclosure, and is not isomorphic to the pclosure in the sense that bottom-up derivations are. Moreover, top-down structural operations typically pay more attention to the tree structure of the pclosure than do bottom-up operations, since in adding literals at the tip of a branch $B$, the assignment $B$ must be taken into consideration.

These two features—linear derivations and context sensitivity in adding clauses—are the hallmark of top-down methods.

Throughout this section, 'pclosure' will mean 'pclosure of type B', unless stated otherwise, and we will adopt the slightly altered notion of
relevance mentioned in section 4.3 above, i.e. $L \in \mathcal{N}_s$ is relevant if $N$ is a tip and $f(N)$ is undefined.

Most of this section is devoted to top-down methods, which generalise and improve the 'linear resolution' collection of systems in the literature, especially $\mathcal{S}\mathcal{L}$-resolution. First however, I consider a simpler system which is very similar to Prawitz' method of matrix reduction (1969).

4.5.1 Free top-down growth of pclosures

Given a set of clauses, we can grow a binary semantic tree closed for $S$ as follows. (1) Choose a literal $L$ occurring in $S$; (2) form the tree $T$: 

```
    L
   / \  \
M   L
```

(3) pick a tip $N$ of this tree at which no clause of $S$ fails (if there is one); (4) pick a literal $M$ different from $L$ which occurs in $S$; (5) add the tree 

```
    M
   / \  \
M   M
```
below N to form a new semantic tree T; (6) goto (3).
If step (3) fails because there is no tip at which no clause fails, then T is closed for S. Such a procedure will clearly always work eventually, no matter which choices are made in steps (3) and (4). However, the resulting closure may be extremely prunable. The system of free top-down growth attempts to diminish the likely irrelevance of the assignments on the tree T by insisting in step (4) that both \( \bar{L} \) and \( L \) occur in clauses of S which could conceivably fail on some extension of the branch \( B_N \), i.e. which are not already made true by \( B_N \).

Let \( C = \langle\langle T, s \rangle, f \rangle \) be a pclosure for S and suppose \( f(N) \) is undefined. Consider the Davis-Putnam set \( DP(S, B_N) \). Every \( C \in DP(S, B_N) \) corresponds to a \( D_C \in S \) such that \( C \) is a subclause of \( D_C \) and \( (D_C - \{ L : L \in C \}) \subseteq B_N \); moreover, \( C \cap B_N = \emptyset \) so that \( C \) consists of literals which do not occur so far on the branch to N.

The following algorithm scheme defines the system of free top-down growth from S:

(1) set \( C = \langle\langle T, s \rangle, f \rangle \), where \( T = \{ N \} \), \( N_s = \emptyset \), and \( f \) is totally undefined unless \( <> \in S \), when \( f(N) = <> \).
(2) Choose a tip $N$ of $C$ with $f(N)$ undefined. If there is no such tip $N$, exit with success.

(3) Choose a literal $L$ such that $L$ occurs in $C_1 \in \text{DP}(S, P_N)$ and $\overline{L}$ occurs in $C_2 \in \text{DP}(S, P_N)$. If there is no such literal $L$, exit with failure.

(4) Add the tree

\[
\begin{array}{c}
\otimes \\
L & \overline{L} \\
N_1 & N_2
\end{array}
\]

down below $N$ in $\langle T, s \rangle$. If some clause $C$ in $S$ fails at $N_1$, then $f(N_1) = C$; otherwise $f(N_1)$ is undefined.

(5) goto (2).

In steps (2) and (3), the choices can be made in any arbitrary way: both are selectable choice ranges.

Clearly, this method is correct (if it exits with failure in step (3) then $\text{DP}(S, P_N)$ is satisfiable, say $\mathcal{A}$ is a Herbrand model: then $P_N \cup \mathcal{A}$ is a Herbrand model for $S$; and complete (all semantic trees are closed if $S$ is unsatisfiable; $S$ contains only finitely many atoms and therefore the procedure must terminate each
branch, eventually, and choose a different one in step (2).

However, if we regard $C$ as a structure, and the addition of the three-node tree in step (4) as a structural operation, then this operation is not sensible. For, this three-node tree shares no vocabulary with $C$. The growth of $C$ is affected only negatively by the its contents.

This nonsensibleness explains the apparent lack of search in the procedure, understood this way.

It is also reflected in the fact that the closure grown by the system may well be prunable, in spite of the elaborate precautions in step (3). For example, faced with the set

$$\{\langle AC \rangle, \langle AC \rangle, \langle B \rangle, \langle BD \rangle, \langle A \rangle\},$$

the procedure might grow the closure:

```
A  \overline{A}
  \overline{A}
  B  \overline{B}
 C  \overline{C}
  \overline{C}
  \overline{C}
```
We could attempt to retrieve sensibleness by insisting, in step (3), that the literal L chosen occurs (positively or negatively) in a clause to whose failure the assignment $D_N$ already contributes. On the first pass through the procedure there is no such clause, and the system would be obliged to search through the set of possible first clauses. This modified system satisfies theorem 1, therefore, and is sensible and liftable. In section 4.5.2 I will discuss a system similar in some ways to this modification. Notice that the tree built by the free top-down growth can be thought of as an amalgamation of all the trees in the search space of this sensible modification. However, I will not pursue this topic here.

The free top-down system is closely similar to the matrix reduction method of Prawitz (1969). Unfortunately an exact comparison is not possible at the PC level as Prawitz uses a different lifting method not based upon general unifiers. At the propositional level, free top-down growth is isomorphic to a special case of Prawitz's system.
Pravitz grows a binary tree of matrices, which we can think of in the present terminology as sets of clauses. The basic tree-growing primitive is splitting; from the matrix \( M = \{ \ldots C_1 \ldots C_2 \ldots \} \), where \( L \in C_1 \) and \( \overline{L} \in C_2 \), the two matrices
\[
M_1 = \{ \ldots C_1 - \{L\} \ldots \langle \overline{L} \rangle \ldots \}, \quad M_2 = \{ \ldots \langle L \rangle \ldots \langle \overline{L} \rangle \ldots \}
\]
are formed, labelling (new) sons of the (tip) node of the tree labelled by \( M \). If a matrix contains clauses \( \langle L \rangle \) and \( \langle \overline{L} \rangle \) it is a terminal matrix and no sons of it are generated. Pravitz also has a second primitive, simple reduction: from \( M = \{ \ldots C \ldots \langle L \rangle \ldots \} \), where \( \overline{L} \in C \), generate the single son matrix:
\[
\{ \ldots C - \{ \overline{L} \} \ldots \langle L \rangle \ldots \}
\]
(This clearly is the special case of splitting where one son matrix is terminal.) When all tips of the tree are terminal, the original matrix has been shown to be unsatisfiable. Pravitz does not explicitly discuss deletion operations; but I will assume that all clauses are fully deleted, which seems to correspond most closely to his description of the matrix reduction system.

Let us call the literals \( L \) and \( \overline{L} \) in the unit clauses introduced in the splitting, hypothesis literals of the matrices in which they occur.
The following special case of the matrix reduction system will be called directed matrix reduction. It is obtained by adding two restrictions:
(a) Immediately after splitting, all simple reductions against the hypothesis literal introduced in the splitting are performed. Call the resulting matrices the reduced sons of the matrix which was split.
(b) Splitting or simple reduction are never applied to a clause C in M containing a hypothesis literal L. We will call such a clause dead. Now, let \( \hat{M} \) be the set of clauses, got by removing singletons of hypothesis clauses and all dead clauses from the matrix M. Then it is easy to see that if \( \hat{M}_1 \) is a reduced son of \( \hat{M} \), then
\[
\hat{M}_1 = \text{DP}(\hat{M}, \{ L_1 \})
\]
where \( L_1 \) is the hypothesis literal introduced in the splitting. Thus if \( \hat{M}_0 \) is the original matrix and \( \hat{M}_{i+1} \) a reduced son of \( \hat{M}_i \), then
\[
\hat{M}_n = \text{DP}(\hat{M}_0, \{ L \})
\]
where \( \{ L \} \) is an assignment if \( L \) and \( \bar{L} \) both \( \in \{ L \} \), then for some \( \hat{M}_i \), \( i < n \), we must have L an hypothesis literal of \( \hat{M}_i \) and \( \hat{M}_{i+1} \) got from \( \hat{M}_i \) by splitting on a C containing L, which violates condition (b).

Moreover, if M is terminal but no ancestor
of $M$ is terminal then $M$ must contain $\langle L \rangle$ and $\langle \bar{L} \rangle$ where precisely one of $L$, $\bar{L}$ is a hypothesis literal (say $L$). For both cannot be hypothesis literals; and if neither are, then they would both have occurred in the parent matrix of $M$. That is, we must have a clause $C=\langle \bar{L} \rangle$ in $M$ such that the original clause $C'$ in $M_1$ from which $C$ descends (in the obvious sense) is false in $\alpha_M$, the set of hypothesis literals of $M$.

We have therefore a correspondence between pclosures $C$ for $S$ produced by free top-down growth and trees $T$ of matrices produced by directed matrix reduction. If we ignore the intermediate simple reduction matrices immediately after a splitting, and have binary trees $T'$ labelled with reduced son matrices, then the correspondence $C \leftrightarrow T'$ is 1:1, and $C$ and $T'$ are isomorphic. If $m(N)$ is the matrix at the node $N$ in $T'$, and $C=\langle \langle T',s \rangle,f \rangle$, then we have:

$$
\mathcal{D}_N = \alpha_{m(N)} \\
\text{DP}(S, \mathcal{D}_N) = m(N)
$$

and if $N$ is a tip, then $Nf$ is a clause $C \in S$ such that the descendant of $C$ in $m(N)$ is a singleton $\langle \bar{L} \rangle$ with $L \in \mathcal{D}_N$. 
I will not attempt to extend this correspondence to the PC level, as neither free top-down growth nor matrix reduction is liftable.

4.5.2 The jigsaw method

In this section I will describe a top-down system which in many respects resembles free top-down growth but is more closely controlled by the form of the particular input clauses in S, and is therefore sensible, liftable and guaranteed to generate only unprunable closures. It generalises the various systems of 'linear resolution' in the TP literature, including Loveland's Model Elimination method (1969).

It is convenient at this point to assume that (binary) semantic trees are ordered. This does not affect any of the theory so far: in particular, it does not affect the prune lemma. It does however make the description of these top-down systems somewhat neater. We will thus refer to the right and left sons of a node in a semantic tree, the rightmost branch of a semantic tree, etc..
Given a clause $C = \langle L_1, \ldots, L_n \rangle$, let $\text{lin}(C)$ be the structure $\langle \langle T, s \rangle, f \rangle$, where $\langle T, s \rangle$ has the form:

![Diagram of a tree structure with nodes labeled $L_i$ and $T$]

and where $f(N) = C$ and $f$ is undefined elsewhere. $\text{lin}(C)$ is clearly an unprunable closure iff $C$ is fully deleted.

The jigsaw method begins with the trivial pclosure $\langle \langle \overline{N_s^f}, s \rangle, f \rangle$ where $N_s = \emptyset$ and $f(N)$ is undefined, and builds pclosures by using the structural operation $R_j = R_j \otimes R_D$, where:

$$R_j(\mathcal{C}_1, C, C') \equiv (\mathcal{C}_1 = \langle \langle T, s \rangle, f \rangle \text{ is a pclosure; } \exists N. N \text{ a tip of } T \text{ with } f(N) \text{ undefined; }$$

$$\& C \text{ is a clause with } C \cap \mathcal{P}_N = \emptyset$$

$$\& \overline{N_s} \in C;$$

$$\& C' \text{ is the result of adding } \text{lin}(C - \{\overline{N_s^f}\}) \text{ below } N \text{ in } \mathcal{C}_1'.)$$
RD($C', C$) ≡ $C$ is the result of removing a branch $\mathcal{P}_{\mathcal{N}'} \mathcal{N}$ from $C' = \langle \langle T, s \rangle, f \rangle$, where $N' < N$ and $N_s = Ms$ for some $M \in \mathcal{P}_{\mathcal{N}'}$.

Clearly, $R_D$ is a deletion operation in the sense of section 3.5.3 above, and if $R_J^f \otimes R_D(C_1, C, C)$ then $C$ is a pclosure. Clearly, also, $R_J^f$, and hence $R_J^f$, is liftable and sensible.

When $C$ results from an application of $R_D$, we will say that the node $M$ which mediates the deletion is a deletor for $N'$ in $C$. In diagrams we may draw dotted lines from nodes to their deletors.

Note that since $C_1$ is a pclosure, the deletions performed by $R_D$ in an application of $R_J^f \otimes R_D$ all apply to nodes in the branch of $C'$ imported from $C$. If we lift $R_J^f$ by the methods discussed in section 3.5.3, where applications of $R_D$ are
restricted to such branches also, then irredundant lifting is automatic, for inadmissibility can be immediately detected by examining the branch structure of the pclosure. There is no need to apply any deletion operations to input clauses, since all necessary deletions will be performed by $R_D$ when the clause is used to grow a pclosure. In the lifted space, this observation shows that no factoring is necessary.

It may be however considered desirable in practice to fully delete input clauses and then, in applying $R_D$, only consider deletions which are mediated by assignments in $\mathcal{O}_N$, where $N$ is the tip of $C_1$ in the immediately preceding application of $R_1$. On lifting, we get complete factoring of input clauses and a correspondingly restricted application of $\tilde{R}_D$. This has the marginal advantage in practice that the effort involved in computing an mgu in a factoring input clause is expended only once, and the result stored for future use. This time/space tradeoff probably makes the second method more efficient, as storage is cheap in good implementations.
However, the difference is marginal and it is more convenient in describing the system to leave all the deletion to $R_D$. In what follows I will ignore this minor complication.

Notice that a result of an application of $R_J$ is unprunable if its first input is, so that all derivations from the trivial pclosure relative to $R_J$ have unprunable conclusions.

The set of tips $N$ of $C$ with $f(N)$ undefined forms a choice range for the jigsaw method, which is in fact selectable. This means that the jigsaw method has a search space with a particularly low branching rate. This fact, and an extended set-of-support refinement, follow from lemma 4.5.1 below.

Given a semantic tree $<T',s>$ for an unsatisfiable set $S$ of clauses, and a node $N$ in $T$, I will define a set of clauses $S_N$ associated with $N$ recursively as follows:
(a) If \( N \) is the root, then \( S_N \) is a minimal unsatisfiable subset of \( S \).

(b) Suppose \( S_M \) is defined and \( M < N \). Then \( S_N \) is a minimal unsatisfiable subset of \( DP(S_N, N_s) \).

Here, 'minimal unsatisfiable' means that if any clause is removed then the set is satisfiable.

Clearly, \( S_N \subseteq DP(S, \mathcal{P}_N) \), and there is a closure \( \langle T, f \rangle \) for \( S \) where \( T' \) is a subtree of \( T \) and every clause failing at a node below \( N \) is \( D_C \) for some \( C \in S_N \). (Just attach a semantic tree for \( \| S_N \| \) below \( N \).) If \( \langle \rangle \in DP(S, \mathcal{P}_N) \), then \( S_M = \{ \langle \rangle \} \) for all \( M \gg N \), since \( \langle \rangle \) is unsatisfiable. \( S_N = \{ \langle \rangle \} \) iff \( D_C \subseteq S \) is false in \( \mathcal{P}_N \). For all \( C \in S_N \), \( D_C \cap \mathcal{P}_N = \emptyset \). The idea for this construction comes from (Kowalski & Kuehner 1971).

The construction of \( S_N \) is not, in PC, an effective process, since determining satisfiability of a set of clauses is not even semi-effective. Even in propositional calculus, to compute \( S_N \) is a task of the same order of difficulty as proving the unsatisfiability of \( S \) in the first place: so one would not permit an operation or
choice function in a TP system which depended upon such an operation (c.f. the discussion in section 1.5.3). Nevertheless, I will now flout my own rules and describe a system which does use this construction. It must be understood that this description is not intended to denote a real TP system: the choices must be replaced by searches through the appropriate choice range in the actual system. It is, however, useful fiction for the proof of the lemma below. For the moment, then, let us pretend to be Gods.

Consider the following algorithm scheme:

1. Set $C = \langle \langle T, s \rangle, f \rangle$, where $T = \{N_s^2\}$, $N_s = \emptyset$, and $f$ is totally undefined unless $\langle \rangle \in S$, when $f(N) = \langle \rangle$.
2. Choose a tip $N$ of $C$ with $F(N)$ undefined. If there is no such tip $N$, exit with success.
3. Choose a clause $C \in S_N$ with $\overline{N_s} \in D_C$; then $R_j(\overline{C}, D_C, C')$ for some $C'$. If there is no such clause $C$, exit with failure.
4. Set $C = C'$.
5. goto (2).
A TP system defined by this algorithm will be called a selective jigsaw system. This algorithm should be compared with that defining the free top-down system.

**Lemma 4.5.1** A jigsaw system is complete.

**Proof.** We have only to show that the algorithm does not exit with failure in step (3); for it must terminate somewhere since \( \| S \| \) is finite. Suppose therefore the algorithm exits in step(3), so that for some node \( N \) in \( \mathcal{C} \), there is no clause \( C \) in \( S_N \) with \( \overline{N_s} \in D_C \).

By construction, \( N_s \in C \) for some clause \( C \in S \), failing at the leftmost tip below the parent \( N' \) of \( N \). Let \( M \) be the ancestor of \( N' \) at which \( \text{lin}(C'-\{\overline{N_s}\}) \) attached in an earlier application of \( R_j \), and let \( N_1, \ldots, N_k \) be the right sons of the nodes on the branch \( \mathcal{B}_{MN'} \).
I will show that $S_M$ has an unsatisfiable proper subset, which contradicts the assumption that $C$ was constructed by using the algorithm.

By construction, $C=D_{C'}$, for some $C'=S_M$. Now, add a semantic tree for $\| S_{N_1} \|$ below each $N_1$. Since each $S_{N_1}$ is unsatisfiable, these semantic trees are all closed for $S_M$. Moreover, $N_1$ is irrelevant since no clause in $\{ D_C : C \leq S_{N_1} \}$ contains $N_1$. We can therefore prune to give a closure for $S_M$ at no tip of which $C'$ fails:

Thus $S_M - C'$ is unsatisfiable.

A realistic system based on this algorithm can thus select freely in step (2) and this
selectivity lifts intact to the PC level, as explained in section 3.5.4 above. However, it has to search for appropriate clauses in step(3), in general. If, however, the TP system has access to an oracle of some kind, it can make use of it in step (3). For example, it might be possible to restrict attention to a limited number of candidate clauses, on the grounds that the rest of the clauses form a satisfiable set. This is the set-of-support restriction. The lemma shows that it can be invoked at every stage in the growth of a pclosure, provided proper attention is paid to the context of the tip being processed.

In the course of building a pclosure (of type B) by the jigsaw method, pclosures of type A will be produced when a part of the tree gets closed off. Let \( N \) be an interior node of a pclosure \(<<T, s>, f>>\), and suppose \( f(M) \) is defined for every tip \( M \) below \( N \). Then \( C_N = <T_N, s>, f>\), where \( T_N \) is the subtree of \( T \) rooted at \( N \), is a pclosure of type A, where \( \text{ex}(C_N) = \{ \text{M}: \exists K. M \in T_N \} \). Since all clauses labelling tips of such a pclosure are
in $S$, clearly, it could be derived from $S$ by the application of the binary bottom-up operation $R_b$.

In the course of building a pclosure, therefore, such type A pclosures could perhaps be recognised, and clauses made from their excesses could be added to the stock of available clauses to be used in applications of $R_j$. Such a clause will be called a lemma, following Loveland. The generation and use of lemmas shortens derivations but increases the branching rate in the search space, and their usefulness is not therefore entirely clear. In section 4.5 below I describe a variation on the jigsaw method which has the advantages of lemma generation without the disadvantages.

The jigsaw operation $R_j$ is defined only upon a particular branch $C_N$ of the input pclosure; the rest of the structure of the pclosure is irrelevant to the application $R_j$. This structural redundancy can be eliminated by moving to a collapse in which pclosures are replaced by branches, if we make selections of nodes in some sufficiently systematic way. There are two systems in the literature which are got by the two obvious
such schemes: if we always choose the leftmost tip, we get SL-resolution; if we always choose the rightmost tip, we get model elimination. In the next two subsections I will outline these two collapses, indicating the translations into the published descriptions of these two systems.

I will include only sufficient detail to enable the reader to see the mappings for himself, in order to save space. In diagrams a shaded triangle indicates a pclosure of type A, i.e. a closed semantic subtree, and a black dot the root of such a tree.

4.5.2.1 SL-resolution

I assume the reader is familiar with the terminology of (Kowalski & Kuehner 1971). I will indicate a technical term from that paper by placing single quotation marks around it.

If we always choose the leftmost tip N with f(N) undefined, then a typical pclosure has the form:
Regarding the branch $B_N$ from the root to $N$ as a sequence, $\langle N_0, \ldots, N_n=N \rangle$ ordered by $<$, then the corresponding 'admissible chain' $C(N)$ is the sequence $\langle L_1, \ldots, L_n \rangle$, where $L_n=N_s$ is a 'B-literal' and otherwise $L_i=N_1's$ and is an 'A-literal'; if $N_i$ is a right son of $N_{i-1}$; and $L_i=N_1's$ and is a B-literal, if $N_i$ is a left son of $N_{i-1}$. Thus

$B_N = \{ L: L \text{ is an A-literal of } C(N) \} \cup \{ L: L \text{ is a B-literal of } C(N) \text{ other than the last literal} \} \cup \{ L: L \text{ is the last B-literal of } C(N) \}$.
Closed subtrees to the left of the active branch $B_N$ correspond to chains which have been 'truncated' and are no longer represented in the chain. It is easy to see that any admissible chain can be redrawn as a branch of the form above. The closed trees attached to the left of the active branch can be retrieved by examining the chains in the derivation of the chain.

The operation $R_J$, rediscribed in terms of chains, is the operation of 'extension'; the operation $R_D$ becomes the operation of 'reduction'. The operation of 'truncation' corresponds simply to backtracking up the tree when a left son becomes closed: the lemma generated (optionally) during truncation is the pclosure of type $\mathbb{A}$ whose root is that node.

The published description of SL-resolution uses the second approach to deletion discussed earlier, in which input clauses are factors and $R_D$ is not applied between nodes in the most recently introduced tree fragment. (In fact, I got the idea from the SL-resolution authors.)
The main result of the SL-resolution paper - that any SL-derivation of the empty chain corresponds to a minimal resolution derivation of $\langle \rangle$ of the same 'r-m-size' - is trivial in the present framework. For consider a pclosure $C = \langle T, f \rangle$ generated by $R_J \otimes R_D$: we could, by lemma 4.4.1, have generated $C$ by $R_b \otimes R_2$. Clearly, by inspection, the number $n$ of applications of $R_J$ or $R_b$ in the respective derivations is the size of $T$, while the number of applications of $R_D$ or $R_2$ is

$$\sum_{N \text{ a tip of } T} \text{length}(f(N)) - n,$$

since each application of $R_D$ or $R_2$ deletes precisely one occurrence of a literal.

4.5.2.2 Model Elimination

I assume the reader is familiar with the terminology of (Loveland 1969). Again, technical terms of that paper will be quoted.

If we always choose the rightmost tip $N$ with $f(N)$ undefined, then a typical pclosure has the form:
The description of the 'admissible chain' corresponding to the active branch $\mathcal{R}_N$ is a little more complicated here. Let $\mathcal{R}_N = \langle N_0, \ldots, N_n = N \rangle$, as before. If $N_i$ ($i > 0$) is a right son of $N_{i-1}$, let $\langle N_{i,1}', \ldots, N_{i,j_i} \rangle$ be the leftmost branch below but not including the (left) brother of $N_i$. 

\[ N_{i-1} \quad N_i \quad N_{i,1} \quad N_{i,j_i} \]
If the left brother of $N_i$ is a tip then this is $\langle \rangle$. Let $\{N_{i_1}, \ldots, N_{i_k}\}$ be the subset of $B_N$ consisting of all left sons. Now, the admissible chain $C(N)$ is

$$\langle N_{i_1}, l_{i_1}, \ldots, N_{i_1}, l_{i_1}, N_{i_2}, l_{i_2}, \ldots, N_{i_n}, l_{i_n}, N_{s}\rangle - \{N_{i_1}, \ldots, N_{i_k}\}$$

where each $N_{i_j}$ is a 'B-literal'; $N_{s}$ is a B-literal; and otherwise each $N_{i_j}$ is an 'A-literal'. Thus $C(N)$ is got by ignoring left sons on $B_N$, and prefixing the label of each right son by the labels of the left sons of its left brother, in ascending order.

Since left sons are ignored, we no longer have $B_N$ explicitly displayed in $C(N)$. This lack is rectified by Loveland's 'lemma' device, as explained below. Closed subtrees to the right of the active branch correspond to chains which have been 'contracted' and are no longer present in the chain; again, they can be retrieved by examining the chains in the derivation of the chain. It is easy to see that any admissible chain can be redrawn as a branch of the form above.

The operation $R_J$, redescribed in terms of chains, is the operation of 'extension'; the
operation $R_D$, in which the deletor is a right son, is the operation of 'reduction'. Again, 'contraction' is simply a backtracking device. However, the 'lemmas' formed during contraction play a vital role, since they make left sons on $P_N$ available to the inference system. Applications of $R_D$, in which the deletor is a left son, correspond to extensions against a lemma (the 'first literal' of a lemma is a right son on $P_N$), followed by reductions of the rest of the lemma against appropriate right sons on the branch. (The other literals of a lemma are all the A-literals which acted as deletors when the lemma was formed. The 'scope' device is a method of deleting deletors.) It is of some interest, in comparing model elimination with SL-resolution, to note that model elimination's use of 'lemmas' thus corresponds to SL's 'merging'. Kowalski and Kuehner's claims for the superiority of SL-resolution over model elimination should be re-examined in this light.

Loveland does not factor input clauses, but leaves all deletion to $R_D$, as we do.
4.5.3 Retrospective reordering

The jigsaw method will not generate any unprunable pclosures, and sometimes it will not generate a smallest closure, as the following example, due to Donald Kuehner, shows:

\[
\begin{align*}
\{N\} \\
\{N, M, Q\} \\
\{N, P, R\} \\
\{L, Q\} \\
\{L, R\} \\
\{L, P\} \\
\{L, M\}
\end{align*}
\]

A smallest closure for this set of clauses is, for example:

```
with size 6. This cannot be produced by the jigsaw method, as one can see by examining the relevancies
```
in it. Any pclosure produced by jigsaw is built up by assembling pieces of the form:

\[ \begin{array}{c}
\text{N}_1 \\
\text{N}_2 \\
\vdots \\
\text{N}_n \\
\text{[C]}
\end{array} \]

where the assignments \( \text{N}_i \)'s are all relevant to \( C \).

The above closure does not have this form, no matter how it is ordered.

It is easy to see that any closure of this form can be constructed by a selective linear system, by simply choosing in step (3) the clause \( C \) when the node \( \text{N}_0 \) is chosen in step (2). I will say that such a pclosure is in jigsaw form. The clause failing at the tip of the leftmost branch of a closure \( C \) in jigsaw form will be called the main clause of \( C \).

We can eliminate this constraint upon the form of derivable closures, at no cost in increased branching rate, by reordering the nodes
of the semantic tree after it has been generated.

Suppose $N_1, N_2$ are right sons in a pclosure $C = \langle \langle T, s \rangle, f \rangle$, where $N_1 s = N_2 s$ and $N_1$ and $N_2$ are independent. Let $M_1$ be the brother of $N_1$, let $N$ be the paterfamilias of $\{N_1, N_2\}$, and let $N_0$ be the son of $N$ which has $N_1$ as a descendant. Consider the branch $\mathcal{P}_{N_0, N_1}$. If there is a deletor $K$ of some node in $T_{N_1}$ in this branch, we will say that $K$ is forced by $M_1$:

![Diagram of semantic tree branches with forced deletors]

$K$ may force some other node(s) in $\mathcal{P}_{N_0, N_1}$; and so on:
Let \( \rightarrow \) denote the transitive closure of the forcing relation, so that \( M_1 \rightarrow M_1 \) and \( M_1 \rightarrow K_1 \), etc. Now, let \( <K_0, K_1, \ldots, K_k> \) be the sequence nodes such that \( \{ K_0, \ldots, K_k \} = \{ K : M_1 \rightarrow K \} \) and \( K_1 < K_{i-1} \) (so that \( K_0 = M_1 \) is the lowest branch). Let \( T_0 = T \), and \( T_1 \) be the result of moving the branch \( B_{K_1 K_i} \) to \( N \), and then deleting all branches \( B_{M_i M} \) from the tree rooted at \( K \), where \( M_i = K_s \). Clearly, the branch \( B_{M_2 M_2} \) is deleted in this process. Let \( T' \) be \( T_k \):
Then it is easy to see that \( \langle T', s \rangle \) is a semantic tree. Moreover, if \( N \) is a tip of \( T \) with \( f(N) \) defined, and \( N \in T' \), then \( f(N) \) fails at \( N \) in \( T' \) also. Thus \( C' = \langle T', s \rangle , f \rangle \) is a pclosure, in which the node \( N_2 \) has been eliminated. The size of \( C' \) is less than the size of \( C \). Clearly, the operation whose output is such a \( C' \) when its input is such a \( C \), is a deletion operation in the sense of section 3.5.4: we will call it \( R_R \), and will say that the node \( N_2 \) has been merged into the node \( N_1 \). We will say that the nodes \( K_1, \ldots, K_k \) are forced by the merge. (In practice, there are often very few forced nodes in most mergings.)

Consider the modification of the selective jigsaw algorithm in which we insert the following between steps (2) and (3):

(2a) Choose whether to apply \( R_R \) to merge \( N \) against some \( N_1 \), if possible. If \( R_R \) is applied, goto (2).

A system which corresponds to the resulting algorithm will be called a merging system. Any selective jigsaw system is a merging system.
The completeness argument of lemma 4.5.1 applies virtually unchanged to merging systems. Clearly, the new algorithm must terminate since each pass through 2a and 2 reduces the size of \( C \). In the argument of lemma 4.5.1, if \( Ns \) is relevant to the failure of \( C \) then argue as before: if not, it must have been introduced by an application of \( R \). Say we have a clause \( C \in S_N \) in the earlier pclosure such that \( D_C \not\ni Ns \). Then we can use \( C \) at \( N \) in the current pclosure also, since the argument at \( N \) in the current pclosure is a subset of that in the earlier pclosure:

Thus, merging systems are complete. However, unlike selective jigsaw systems, merging systems are capable of generating simplest closures, provided appropriate selections are made. This follows from the
construction in the proof of the following lemma.

**Lemma 4.5.2** Let \( C \) be an unprunable closure for \( S \), and \( C \) a clause failing at a tip of \( C \). Then there is a pclosure \( C' \) in jigsaw form such that a closure of size not greater than \( C \) can be obtained by applying \( R_R \) to \( C' \).

**Proof** by induction on the size of \( C \). If \( C \) has size 1 then the result is trivial.

Reorder the fans of \( C \) if necessary so that \( C \) fails at the leftmost tip \( N \); this does not change the size. Now let \( B_N \) be the leftmost branch of \( C \). If every literal in \( B_N \) is relevant at \( N \), apply the induction hypothesis to the trees rooted at brothers of nodes in \( B_N \), selecting as main clause one to which the assignment at the root is relevant, and put the pieces back together: the result is in jigsaw form.

Now suppose that \( M \in B_N \) and \( Ms \) is irrelevant to the failure of \( C \) at \( N \). Since \( C \) is unprunable, \( Ms \) must be relevant to some node in the tree rooted at the brother of some node on \( B_{MN} \); let \( N_1 \) be the highest such node on \( B_{IN} \).
Let $C_1$ be the result of moving the branch $P_{M'N_1}$ to $N_1$, and then deleting all branches $P_{K'K}$ from $T$, where $K$ occurs in $P_{M'N_1}$:

Clearly $C_1$ is not larger than $C$.

However, $C_1$ may fail to be a closure because $M_s$ was relevant to the failure of a clause $C'$ at a tip $K$ of the tree rooted at the brother of some node in $P_{N_1N}$. For each such $K$, add the three-node tree $M_s$.
below \( K \). The result is a pclosure \( C_2 \) such that \( R_R \) applies between each such \( K_1 \) and the brother of \( M \); and if all such applications of \( R_R \) are performed, the result has the same size as \( C \). If there are no such nodes \( K \), let \( C_2 \) be \( C_1 \).

Repeat this construction; eventually we have a pclosure \( C_n \) such that the leftmost branch contains no assignments irrelevant to the failure of \( C \). Apply the inductive hypothesis to the trees rooted at brothers of nodes on this branch, choosing main clauses appropriately, and put the pieces together: the result is \( C' \).

QED.

Clearly, we can follow this construction by making appropriate choices in the algorithm. For example, the closure described at the beginning of this section corresponds to the following under the construction in the proof:
Ignoring the tip which is deleted by $R_R$, this pclosure has size 6; as the construction in the proof indicates.

Applications of $R_R$ in which the node $N_1$ is the root of a closed subtree, correspond to applications of $R_J$ with the appropriate lemma. However, $R_R$ is selectable in a merging system: if the system chooses to use $R_R$ to eliminate a literal, it does not need to consider other ways of removing the literal. Thus there is no increase in branching rate as a result of using $R_R$. Moreover, $R_R$ can be applied when the node $N_1$ is not yet the root of a closed tree, in order to rearrange the pclosure.

This selectability does not lift intact to the PC level, unfortunately, since the lifted operation $\widehat{R}_R$ may involve a nontrivial instantiation of the pclosure. $R_R$ itself is still selectable at the general level, however, - and use can be made of its selectability in order to decrease potential redundancy in the lifted search space, by the methods of section 3.5.3.
REFERENCES


Norton (1966) ADEPT - a heuristic program for proving theorems in group theory. Memo MAC-TR-33, Project MAC, MIT.


