DECIDABILITY, COMPLEXITY AND AUTOMATED REASONING IN RELEVANT LOGIC.

By
Jacques M. Riche

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY OF THE AUSTRALIAN NATIONAL UNIVERSITY

Department of Philosophy and Automated Reasoning Project Research School of Social Sciences September 1991
Unless otherwise acknowledged, this thesis is my own work.
Acknowledgements

I would like to thank my supervisors Dr John K. Slaney and Dr Robert K. Meyer whose ideas, advices and discussions have greatly contributed to the research reported in this work. I also want to thank my advisers, Prof M. A. McRobbie, Executive Director of the Center for Information Science Research and former Director of the Automated Reasoning Project, Dr Richard Sylvan of the Department of Philosophy and Dr Errol Martin of the University of Canberra.

I am grateful to all people with whom I could discuss my work for their support and interest during these years of research, Dr André Fuhrmann, Dr Rod Girle, Dr Peter Lavers, Dr Edwin Mares, Dr Igor Urbas, Hajime Sawamura, Dr Paul Thistlewaite, particularly, whose original research was the starting point of this work, and many other occasional visitors of the Automated Reasoning Project.

Also to John Barlow and Dr Gustav Meglicki for their patience with a ruthless system user, and to all the staff of the ARP, Dorothy, Jenny, Jocelyn, Kellie, Carolyn, Regina, who contributed in making the ARP a pleasant place to spent most of these years.

For my discovery of the Antipodean Logics and Philosophy, my special thanks to Dr Richard Sylvan, and to Prof Jack Smart, who, with Liz Smart, helped us in adapting to a new world. For making the Australian adventure possible, our greatest thanks to Prof Paul Gochet whose constant encouragements and friendship over the years have always been greatly appreciated.

Finally, and most of all, I want to thank my wife Catherine Gelinne for her long patience and willingness to abandon everything to satisfy my own research interests, and become my excellent typist. Last but not least, my parents, Gabriel and Lucienne Riche, who once allowed me to engage in the Holzwege der Philosophie and eventually find the logical way out. To them, I dedicate this work.

Finally, Relevant Logic being now a local export product, I am very grateful to the Australian National University for an ANU PhD Scholarship, and to Prof M. A. McRobbie and Fujitsu Australia for a Scholarship in Information Sciences that supported this research.

I also thank the Argonne National Laboratory for providing access to its machines.
List of Trademarks.

The X Window System™ is a trademark of the Massachusetts Institute of Technology.
Sun Workstation® is a registered trademark of Sun Microsystems, Inc.
SunTools™ and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.
Connection Machine® and C*® are registered trademarks of Thinking Machines Corporation.
Unix® and C are registered trademark of AT&T Bell Laboratories.
Sequent Symmetry® is a registered trademark of Sequent Computer Systems, Inc.
Butterfly® is a registered trademark of BBN Advanced Computers, Inc.
Cray® is a registered trademark of Cray Research Inc.
Abstract

This dissertation is a contribution to computational logic and automated reasoning in Relevant Logics. The first three chapters are theoretical and investigate the decision problem of several systems of these logics in an algebraic perspective. It is shown that the termination condition of the decidability procedure is equivalent to some known results in the theory of ordered sets and in the theory of commutative rings. The complexity of these logics and of their decision procedure is then investigated and some new results are obtained. These results also hold in various other fields of computational logic where the same termination condition is used. This is particularly the case in Logic Programming which is discussed in Chapter three, making the transition with the rest of the thesis devoted to an application of these investigation in Automated Theorem Proving. The relations between Automated Theorem Proving and Logic Programming are discussed, and the decision procedure is implemented in a Prolog theorem prover for the system LR of Relevant Logics. The main issues discussed concern the computational feasibility with respect to the complexity of the logics and the current inefficiencies of Logic Programming. It is shown that, to some extent, the resources of parallel and massive parallel processing can help in overcoming some of these inefficiencies and part of the complexity of the logics. Even in the present stage of technological development, these computational resources cannot supplant the need for deeper insight into the logic and further discovery of heuristic procedures. But they are required to find and execute these procedures efficiently.
## Contents

Acknowledgements iii  
Abstract v  

1 Proof Theory and Decision Problem. 6  
1.1 Summary. 6  
1.2 LR and its proof theory. 6  
1.2.1 Hilbert system. 7  
1.2.2 Gentzen System. 8  
1.3 Decidability. 9  
1.4 Kripke, Dickson and Meyer Lemmas. 10  
1.4.1 Relevant Divisibility. 11  
1.4.2 Commutative Monoids Semantics. 12  
1.4.3 Finite Model Property. 13  
1.5 Dickson's Lemma and Well-Orderings. 14  
1.5.1 Well-Orderings. 15  
1.5.2 Well-Quasi-Orderings. 15  
1.6 Higman's Theorem. 16  
1.6.1 Finite Basis Property. 17  
1.6.2 Some Universal Algebra. 17  
1.7 Higman's Theorem. 20  
1.8 Kruskal's Theorem. 23  
1.8.1 The Tree Theorem. 24  
1.8.2 Labelled Trees. 25  
1.8.3 Higman ≡ Kruskal. 25  
1.9 Finitary Unconstrained Algebras. 26  

2 Constructivity and Complexity. 28  
2.1 Summary. 28
2.2 Hilbert's Finite Basis Theorem ........................................... 29
2.3 Constructive Approach. .................................................. 31
  2.3.1 Seidenberg Construction. ........................................... 32
  2.3.2 Computing with Polynomials. ..................................... 35
2.4 Constructive decision procedure. ...................................... 37
2.5 Complexity. .............................................................. 39
  2.5.1 The complexity of the word and membership problems. ....... 39
  2.5.2 Complexity of the Logic. ........................................... 40
  2.5.3 Lowering the complexity. .......................................... 41
2.6 Kripke's Conjecture. .................................................... 44
  2.6.1 A New Upper Complexity Bound. ................................ 44
  2.6.2 Reverse Mathematics. .............................................. 46

3 Prolog and Parallel Processing. ........................................ 48
  3.1 Logic programming and Prolog. ...................................... 49
  3.2 The Prolog Model. ..................................................... 51
    3.2.1 Prolog III. ........................................................ 54
    3.2.2 The Infinite Trees Model of Prolog. ......................... 56
    3.2.3 Reduction algorithm. ........................................... 57
    3.2.4 Decidability and Termination. ................................ 58
  3.3 Parallel Logic Programming. ........................................ 60
    3.3.1 The Aurora Or-Parallel Prolog. ................................ 62
    3.3.2 The Warren Abstract Machine. ................................ 63
    3.3.3 The SRI Model. .................................................. 64
    3.3.4 Scheduling. ..................................................... 65
    3.3.5 Parallel Programming Style. .................................. 66
    3.3.6 Experimenting with Other Approaches. ...................... 68
  3.4 Logic programming and Automated Theorem Proving. ............. 69
    3.4.1 Intelligent Backtracking. .................................... 71

4 Automated Theorem Proving in LR. ..................................... 75
  4.1 Introduction. ........................................................... 75
  4.2 Dealing with Complexity in ATP. .................................. 76
  4.3 A Gentzen Style Theorem Prover ................................... 78
  4.4 Where Complexity Strikes. .......................................... 79
    4.4.1 Complexity in Action. ........................................ 80
    4.4.2 Proof Theory. .................................................. 82
Introduction.

This dissertation, a contribution to computational logic and automated reasoning in the context of Relevant Logic, builds upon, and is a continuation of previous work by R. K. Meyer, M. A. McRobbie and P. B. Thistlewaite whose book *Automated Theorem-Proving in Non-Classical Logics* presents their extensive research in this field.

One main and early use of automated theorem proving techniques in Relevant Logic was related to the question of the decidability of the Logic $R$ which had been a long standing open and difficult problem.

$L_R$, a logic close to $R$, was known to be decidable, and there were reasons to think that $R$ was undecidable. It was then thought that a theorem prover based on $L_R$ could help in solving the decidability problem of $R$. Given the genuine characteristics of Relevant Logics, traditional methods used in automated theorem proving like resolution could not be used, and a specific-purpose theorem prover, “Kripke” developed by P. Thistlewaite was used.

In order to verify what had been conjectured for a long time, that the logic $R$ is undecidable, the starting idea was to rely on a technique used in another context by Routley and Meyer, consisting in coding the word problem for semigroup into $R$, the former being undecidable, the later would inherit the property. To that end, the usual semigroup operations are translated into the $R$ connectives. The problem was then to find a connective corresponding to the semigroup multiplication operator which is associative, but not commutative nor idempotent, that is, to find a free associative connective in $R$.

The possible candidates for this connective are defined in a list of formulae which, if they are proved in $L_R$, would mean that the corresponding connective is not the free connective wanted for $R$. A set of sixteen possible candidates definitions of an associative connective in $R$ was obtained. They actually define an associative connective in $L_R$, and hence, this connective is not free. Some of these formulae appeared to be too hard to prove automatically. Nevertheless, the theorem prover allowed to partition these formulae into classes of provably equivalent formulae in $L_R$ and helped M. McRobbie to prove them by hand.
using his tableau technique.

Eventually, Urquhart's proof of undecidability of $R$ interrupted this research. But the problem of defining an associative connective, that is to automatically prove the mentioned hard problems, remained as a challenge and a sample of what can be thought of as a hard problem against which to benchmark a theorem prover, or a goal to achieve in building such a system.

The present work in Computational Logic is mainly concerned with the decidability problem of the logic, its constructive aspects, complexity and computational tractability. To set the scene, we briefly characterizes some leading ideas at the background of this research.

The Entscheidungsproblem, the decision problem in logic was introduced by Hilbert as "the fundamental problem of mathematical logic". Given a formula in a logic, is the formula valid or logically true in that logic?

One may wonder whether decidability is still an important issue. As Pratt [Pr90] remarks, considering that mathematics founded on the undecidable Zermelo-Fraenkel set theory does not seem to be a major concern for most mathematicians, one could think that decidability is not an issue. Of course, one could argue that there are other basis than ZF on which to found mathematics, or that there are alternative ways of viewing them. And it is what [Pr90] suggests, a general, unique, foundation may not be needed, because any given argument may be considered inside small and localized theories. Mathematics would then be a family of domain specific theories. And the shift of perspective could be justified by viewing theories as a way "to organize thought to be constructive without being oracular".

The Entscheidungsproblem was eventually proved undecidable by Church and Turing who showed that the validity or, equivalently, the satisfiability problem, is undecidable. This did not prevent the interest to turn toward related questions aimed at delimiting the realm of decidability, like discovering which classes of sentences, if any, of first-order logic are decidable, or which are the minimal undecidable classes of sentences. It is interesting to note that Ramsey's famous theorem in Combinatorics that we will encounter later on, was proved in order to show the decidability of some special class of first-order logic sentences with prefixed quantifiers.

The interest into decision questions was revived with the advent of computer science, where the complexity of decision procedures and of algorithms is a major problem. But more than decidability itself, what is now a fundamental issue is tractability or feasibility. Quoting V. Pratt again, who, noting the current tendency of viewing programs as proofs
and proofs as computations, proposes that "the proper notions of constructivity in logic are its computational complexity and its human surveyability". The criterion for judging the merits of any theory would then be its tractability. And the criterion for tractability would be "the threshold of polynomial and exponential time". But this should appropriately be seen as a research program until many of the problems which pave the way, $P \neq NP$ being not the least, are solved.

This dissertation can be divided into two main parts. One part, theoretical, investigates the decision problem of the logic and its complexity. The other part, empirical, concerns the development of a Prolog Automated Theorem Prover for the logic $LR$.

Chapter one investigates the decision procedure of the logic $LR$. After a review of the proof theory of the logics and its decision procedure based on Kripke's lemma, starting with R.K. Meyer's original work on improved decision procedures for Relevant Logic which provides, at the same time, a commutative monoid semantics for some relevant systems and some unmatched insights into Kripke's lemma, mainly through his infinite division principle which he shows to be equivalent to Kripke's lemma and to Dickson's lemma in number theory, these lemmas, the termination condition of the decision procedure, are shown equivalent to Higman's finite basis and to Kruskal's Tree theorems in the theory of well-quasi-orders. This last theorem on well quasi-ordered trees has received a lot of attention in Finite Combinatorics and its finitization by H. Friedman is related to Meyer's construction of finitary unconstrained algebras.

Finiteness is thus guaranteed by Dickson's lemma or any of its equivalent formulations. But practicality or feasibility is not. Dickson had remarked that his result could also be obtained by Hilbert's finite basis theorem which provides the equivalent finiteness condition in the theory of polynomial rings.

Chapter two first investigates the question of the constructivity of the decision procedure and its complexity. Early work of G. Hermann attempted to give a constructive proof of Hilbert's theorem, and any later developments relied on her original work. In the seventies, Seidenberg gave a complete constructive proof of Hilbert's theorem. Given the equivalence of the word problem for commutative semigroup and monoids with the membership problem in polynomial ideals, from a constructive proof of Hilbert's theorem, we can obtain a constructive solution of the word problem for commutative semigroups and monoids, and so, an alternative constructive decision procedure of some of the logics for which a commutative monoids semantics is available.

But, recalling Pratt's suggestion, constructivity is not enough. What matters, is what can actually be computed. We have a finiteness or termination condition, complexity theory tells us what is feasible and what is not.
Complexity results for Relevant Logics are entirely due to A. Urquhart who provided some bounds for the logics and their decidability proof. An improved upper complexity bound is easily obtained as well as other results in the case of a bounded system. This upper complexity bound applies in other fields of Computer Science, like Unification Theory, Constraint logic Programming, Rewriting Systems, Gröbner basis, where a similar termination condition is used. Finally, recent results in the foundations of mathematics allow us to throw some light on the strength of the theorems discussed and, at the same time, to settle Kripke's conjecture about the provability of his decision procedure.

The complexity of the logic \( LR \) had been suspected for a long time on empirical evidence. Now that theoretical results on the complexity of the logic have been found, the question remains to know where, and why the complexity arises, and to what extent it can actually be controled in practical cases, that is, what is feasible.

The rest of the thesis is concerned with the development of a Prolog Theorem Prover for the logic \( LR \) and various attempts to answer these questions.

In chapter three, some aspects of Logic Programming and its relation to automated theorem proving are considered. Prolog, the programming language used in the implementation of the theorem prover described in the next chapter, is sometimes considered as inefficient and inappropriate to develop such a tool. Nevertheless, in the long run, the development of efficient compilers and parallel processing may lead to a reconsideration of such critiques. Considering the natural relationship between Prolog and automated theorem proving, insights gained in one field could benefit to the other. One example is the implementation of one form of intelligent backtracking in the theorem prover. Reciprocally, from a theoretical point of view, noting that in Prolog III, the prototype of Constraint Logic Programming, the termination condition of the algorithm is exactly similar to the termination condition of the decision procedure of our logic, the complexity results obtained in the logic apply to it. Moreover, various ways to deal with complexity in the logic could enlight some efficiency issues in Constraint Logic Programming.

It is important to note this relationship since the execution of the proof theory of the logic by the theorem prover can be seen as mimicking the execution of the Prolog engine.

Chapter four shows how an efficient formulation of the proof theory of the logic \( LR \) provided in [TMM88] can, to some extent, constrain its complexity. A sound, complete and correct implementation of the proof theory in the theorem prover is first described. Then various improvements to the theorem prover, like the implementation of some proved properties of the logic, the preprocessing of data, the implementation of heuristics search and intelligent backtracking are explained.

A common problem to Prolog and to most automated theorem provers, is to control the generation and the use of information produced at runtime. An obvious condition to
impose on such systems is that any computation which has been performed once should not be done again. This imposes to keep the information generated in a way which does not impede the execution of the program.

Chapter five explains the generation of various databases used to improve the efficiency of the theorem prover. Among these, a search for efficient LR matrices generated by J. Slaney’s program “MaGIC” is emphasized. Subsumption is probably the best way available to control efficiently the irredundancy of the information generated. Performing the subsumption test sequentially on large databases is an expensive operation. Using the resources of a massively parallel machine allows to perform the operation in constant time and on the entire databases at once.

Finally, chapter six presents a selection of results which support the claims made in the previous chapters. Large databases and parallel processing did not allow to solve all the hard problems. But experiments have shown were the complexity lies and have suggested ways to solve them. Whatever the computational resources available, deeper insights into the logic and better heuristics are needed, and massive parallelism allows to perform efficiently intelligent heuristic search on large knowledge bases.
Chapter 1

Proof Theory and Decision Problem.

1.1 Summary.

Chapter one first reviews the proof theory of the logics and the decision procedure based on Kripke’s lemma. It emphasizes R. K. Meyer’s work on improved decision procedures for some systems of Relevant Logics and relate his original insights on the procedures to some results in the theory of ordered sets. Starting with Meyer’s infinite division theorem, which is equivalent to Kripke’s lemma and Dickson’s lemma, we examine Higman’s theorems on well-quasi-orderings and Kruskal’s Tree theorem. Dickson-Kripke-Meyer lemma is a consequence of both of them which are shown to be equivalent.

1.2 LR and its proof theory.

Let a language \( L = (P, C, F) \), where \( P \) is a countable set of propositional variables, \( C \), a set of connectives \( \{ \neg, \&, \vee, \rightarrow \} \) and \( F \) a set of well-formed formulae abbreviated \( \text{wff} \) be defined recursively as follows: (i) any \( p \in P \) is a \( \text{wff} \), (ii) if \( A \) and \( B \) are \( \text{wff} \), then \( \neg A \), \( A \& B \), \( A \lor B \) and \( A \rightarrow B \) are \( \text{wff} \).

A logical system is defined as the least set in \( L \) such that any proposition is an instance of some fixed set of axiom schemata introduced below or is derived by application of the rules.
1.2.1 Hilbert system.

Axioms.

A0. \( A \rightarrow A \)  
Identity

A1. \( A \rightarrow B \rightarrow .B \rightarrow C \rightarrow .A \rightarrow C \)  
Sufffixing

A2. \( A \rightarrow B \rightarrow .C \rightarrow A \rightarrow .C \rightarrow B \)  
Prefixing

A3. \( (A \rightarrow .A \rightarrow B) \rightarrow .A \rightarrow B \)  
Contraction

A4. \( A \& B \rightarrow A \)  
Simplification

A5. \( A \& B \rightarrow B \)  
Simplification

A6. \( (A \rightarrow B) \& (A \rightarrow C) \rightarrow .A \rightarrow B \& C \)  
\&-introduction

A7. \( A \rightarrow A \lor B \)  
Addition

A8. \( B \rightarrow A \lor B \)  
Addition

A9. \( (A \rightarrow C) \& (B \rightarrow C) \rightarrow .A \lor B \rightarrow C \)  
V-introduction

A10. \( \sim \sim A \rightarrow A \)  
Double Negation

A11. \( A \rightarrow \sim B \rightarrow .B \rightarrow \sim A \)  
Contraposition

A12. \( A \rightarrow .A \rightarrow .A \)  
Reduction

A13. \( A \rightarrow .A \rightarrow B \rightarrow .B \)  
Assertion

A14. \( (A \rightarrow .B \rightarrow C) \rightarrow .B \rightarrow .A \rightarrow C \)  
Permutation

A15. \( A \& (B \lor C) \rightarrow .(A \& B) \lor (A \& C) \)  
Distribution

A16. \( A \rightarrow .B \rightarrow A \)  
Positive Paradox

Deduction Rules Schemata.

R1. From \( \vdash A \) and \( \vdash A \rightarrow B \) infer \( \vdash B \)  
Modus Ponens

R2. From \( \vdash A \) and \( \vdash B \) infer \( \vdash A \& B \)  
Adjunction

It should be noted that other choices of axioms are allowed for the systems we will refer to, and that this set of axioms may be redundant. Various systems of relevant logic that we will encounter are axiomatized as follows. T: A1 – A13, A16; E: A1 – A14, A16. LR: A1 – A15, R: A1 – A16 and Classical Logic: A1 – A17. The positive fragment of a system excludes the axioms containing a negation, and the implicational fragment includes the axioms containing only the implication connective.

The propositional constants \( t \) and \( T \), defined respectively as the conjunction of all logical truths (i.e. the weakest sentence of a set of sentences which implies any other member of the set) and the disjunction of all propositions (the dual of conjunction), can be conservatively added with the following axioms. For R and LR, A18. \( A \rightarrow T \), and A19. \( A \rightarrow .t \rightarrow A \).

The intensional connectives, fission, + and fusion, \( \circ \), respectively defined as \( A + B =_{def} \sim A \rightarrow B \) and \( A \circ B =_{def} (A \rightarrow \sim B) \), can also be added conservatively or introduced as
CHAPTER 1. PROOF THEORY AND DECISION PROBLEM.

primitive symbols with appropriate axioms. (See [AB75]).

In the family of relevant logics, \( LR \) which owes its name to its lattice-like operators \& and \( \lor \), is closely related to the main logic \( R \) although in the absence of distributivity, it may lack some of its interesting properties. But it is the price of its decidability (See [TMM88],[AB75]). \( LR \) minus contraction may also be seen as the non-exponential fragment of the fashionable Linear Logic. The main concern of this work being automated theorem proving, we will be more interested in the computational aspects of the logic, its decidability and it complexity than in its strictly logical properties.

A first source of complexity comes from the use of \textit{multisets} of subformulae i.e. sets of formulae in which repetition is allowed, rather than sets as data types. In a Gentzen system \( A \vdash B \) means that the multi set of formulae \( B \) is \textit{derivable} from the multiset \( A \), and repetition of subformulae is allowed. And in a right-sided Gentzen system, \( \vdash B \) means that the multiset \( B \) is derivable from the null multiset.

1.2.2 Gentzen System.

A second and main source of complexity of the logic stems from the axiom or structural rule of contraction (Girard’s “way to infinity”) and the absence of the rule of weakening. In a right sided Gentzen formulation of the logic, where \( \alpha, \beta, \gamma \) range over multisets, these structural inference rules are written

\[
\frac{\vdash \alpha, A, A}{\vdash \alpha, A} \text{ Contraction} \quad \frac{\vdash \alpha, A}{\vdash \alpha, A} \text{ Weakening}
\]

The proof theory implemented in the theorem prover that we will examine later corresponds to the following system \( L5 \) of [TMM88].

\textbf{Axiom:} \( p, \sim p \)

\textbf{Operational Rules:}

\[
\frac{\vdash \alpha, A, B}{\vdash \alpha, A + B} \text{ Fission} \quad \frac{\vdash \alpha, A, \alpha, B}{\vdash \alpha, A \& B} \text{ Conjunction}
\]

\[
\frac{\vdash \beta, A}{\vdash \alpha, A \lor B} \quad \frac{\vdash \beta, B}{\vdash \alpha, A \lor B} \text{ Disjunction} \quad \frac{\vdash \beta, \alpha, \gamma, B}{\vdash \alpha, A \circ B} \text{ Fusion}
\]

Some strict rules govern the partition and distribution of members of \( \alpha \) into the premises of the intensional connectives rules as we will see in due course.

In applying the rule to prove some given formula \( \alpha \), the \textit{proof search tree} of \( \alpha \) is constructed by application of the rules in reverse so that the possible premises which could have \( \alpha \) as conclusion are generated and assigned to the nodes of the tree immediately following \( \alpha \). Then the possible premises of the first two premises of \( \alpha \) and so on, until the tip of each branch, i.e an axiom, \([A, \sim A]\), is reached.
1.3 Decidability.

The zero-degree fragment of $R$ and $E$, which consists in formulae containing only the connectives $\land, \lor, \neg$ is strictly equivalent to classical propositional logic with truth-values as characteristic model.

The first-degree fragment in which the formulae are of type $A \rightarrow B$, where $A, B$ are of degree zero, has a simple decision procedure: there is a four-valued matrix which is a characteristic model for the fragment.

The decidability problem for the implicational fragments of $E$ and $R$ is more difficult. The first result for these systems was obtained by Kripke [Kr59] who proposed a Gentzen system modified in such a way that it admits only one formula in the consequent. The key argument in the decidability proof which is now known as Kripke's lemma does not appear in [Kr59] but is proved in [Be65].

Since it is this decision procedure which is implemented in the theorem prover and which is investigated in the next chapter, we briefly summarize the main steps of the procedure, following essentially [Be65]. (See also [AB75], [Du86] and [TMM88]).

The basic idea of the decision procedure is to restrict the application of the rule of contraction in a Gentzen formulation of the system by building its effect into the operational rules. Contraction of the conclusion of such a rule is allowed as long as its effect could not have been already obtained by first contracting the premises.

A sequent $\beta$ reduces to a sequent $\gamma$ if $\gamma$ can be obtained from $\beta$ by a series of applications of the rules of contraction and permutation. Then, Curry's lemma guarantees that changing the rules leaves the proof theory invariant: if a sequent $\alpha$ is provable in the system, then there is a proof of $\alpha$ containing no $\beta$ such that $\beta$ reduces to $\alpha$. This follows from the fact shown by Curry [Cu50], that if $\beta$ reduces to $\alpha$ and $\alpha$ is provable in $m$ steps, then $\beta$ is provable in no more than $m$ steps. This is what the Curry's property expresses: a proof of a sequent has the Curry Property iff for all sequents $\beta, \gamma$ in the proof tree, if $\gamma$ is a successor of $\beta$ on some branch from $\beta$ to a tip of the tree, then $\beta$ is not strongly contained in $\gamma$. I.e. $\beta$ cannot be reduced to $\gamma$. This shows that for any derivable sequent, there is an irredundant derivation, i.e. a derivation such that on no branch, there is a $\beta$ below a $\gamma$ of which it is a contraction.

A tree is finitely branching if each node has a finite number of children and if every branch of the tree is finite. A tree is finite if it has a finite number of nodes. By König's lemma, a finitely branching infinite tree has an infinite branch.

Let $S$ be a finitely branching system. Any search for a proof of a formula $\alpha$ in $S$ can be represented as a tree: the proof search tree of $\alpha$. If the tree has a proof of $\alpha$ as a subtree then it is called, following [Be65], a complete proof tree.
CHAPTER 1. PROOF THEORY AND DECISION PROBLEM.

For an arbitrary system, there is an effective procedure which applied to \( \alpha \) yields a complete proof search tree, but in general, it is not finite. If it is finite, then the system is decidable since a finite proof search tree has only a finite number of finite subtrees, one of which is a proof of \( \alpha \), i.e. what is ordinarily called the proof tree. Hence, if there is a procedure yielding a finitely branching complete proof search tree with all branches finite, the system is decidable.

First, the proof search tree is complete because if it is constructed in the way briefly explained above and in accordance with the Curry’s property, then if there is a proof of \( \alpha \), then there is a proof of \( \alpha \) satisfying the lemma.

Secondly, since the rule are finitary, there can only be a finite number of premises. Hence, the tree is finitely branching.

We must make sure that the tree has the finite branch property. Define two sequents as cognate, \( \alpha \simeq \beta \) if the same wffs occur in \( \alpha \) and \( \beta \) and call a cognition class of a sequent \( \alpha \) the class of sequents which are cognate. Since the system has the subformula property, (i.e. if \( A, B, C \) are formulae, \( A \) is a subformula of \( A \); if \( C \) is a subformula of \( A \) or \( B, C \) is a subformula of \( A \rightarrow B \), and similarly for the other connectives in an extended system), then any wff occurring in the derivation of a given sequent is a subformula of some wff occurring in the sequent. Hence, by the subformula property, there is only finitely many classes of cognate sequents in the derivation of any sequent.

It remains to show that if only a finite number of members of each cognition classes occur in any branch then each branch is finite.

A sequence \( \alpha_0, \alpha_1, \ldots \) is irredundant if for no \( i, j, j > i, \alpha_j \) reduces to \( \alpha_i \). By Curry’s Property, every sequence of cognate sequents in any branch is irredundant. By Kripke lemma, such sequence is finite.

We now turn to the foundations of this lemma which, as we will see from R. Meyer’s work and other results in the theory of ordered sets, amounts to the well-quasi-ordering of a quasi-ordered set.

1.4 Kripke, Dickson and Meyer Lemmas.

Decidability is guaranteed by Kripke’s lemma, but as Meyer [Me73b] remarks, this decision procedure does not provide any deep insight into the systems as one would expect of a decision proof. Moreover, even though systems of natural deduction are often considered to be more interesting than axiomatic systems because, in concentrating on the consequence relations rather than on derivable theorems, they provide a better understanding of the nature of inference and deduction, or because they give the meaning of logical symbols, for example through the introduction and elimination rules for the connectives, nevertheless,
they are mere deductive techniques while the subject matter of logical analysis are theories and structures into which relevant logics give deeper insights [Me73a]. And this insight is what [Me73a,b,c] provide. It should be noted that the following overview is far from doing justice to the original work of Meyer.

The importance of his contribution is twofold. On one side, it not only enlightens the nature of the termination condition of the decision procedure - an in depth study of Kripke’s lemma or the equivalent Dickson’s lemma is found nowhere in the logical literature where a similar termination property is used-, but taking the commutative monoids as the core of the logical systems studied, it reveals their underlying algebraic structure. The outcome on the other side, is a theory of propositions [Me73a,b] which provides a natural criterion of relevance.

The positive integers seen as the free commutative monoid with primes as free generators, noted \(< N_+, \cdot, 1 >\), and with multiplication as monoid operation are proved relevantly to be characteristic for \( R_f \), the implicational fragment of \( R^1 \) taken as paradigmatic of other equivalent systems. (For example, the argument extends to \( R_{\rightarrow \&} \) into which \( LR \) translates and \( R_{\rightarrow \&_0} \) is embeddable into \( R_{\rightarrow \&} \)). Moreover, by selecting finite subsets of \( N_+ \), finite model properties are obtained for these systems.

### 1.4.1 Relevant Divisibility.

The “use” criterion of relevance imposes that in a deduction the antecedent be effectively used in the derivation of a consequent. Considering ordinary divisibility, Meyer notes that, obviously, it is a fallacy of relevance since 2 divides 6 when 3 is a factor irrelevant to the factors of 2. For this reason, he introduces a notion of relevant divisibility: a number divides another one if all factors of the divisors are used in performing the division. Formally, in a commutative monoid \( M =< M, \cdot, 1 > \) where \( M \) is a set and \( \cdot \) is an associative-commutative operation on \( M \) and 1 is the identity, relevant divisibility \( \mid_r \) is the smallest binary reflexive relation s.t. for all \( a \in M \), \( a_i \mid a_j \), \((0 < i \leq j < \omega)\) and if \( a \mid r c \) and \( b \mid r d \), then \( ab \mid r cd \), with the additional decomposition property that \( a \mid r b \) iff there is some decomposition of \( a \) and \( b \) into the same factors with exponents of the factors being at least as great in the decomposition of \( b \) as in that of \( a \). By relevant division lemma, in any commutative monoid \( M, \mid_r \) is reflexive and has the decomposition property; and, in addition, any free commutative monoid is partially-ordered under \( \mid_r^2 \).

One may wonder what the relationship with the logical systems is. But under these intuitions, relevant implication behaves like relevant divisibility on \( N_+ \), and propositions behave like sets of natural numbers. The primes can be seen as the collection of mutually

---

1axiomatized by the Identity, Permutation, Prefixing and Contraction axioms.

2See addenda, Note 1.
irreducible irrelevant propositions, and the monoid operation, ·, as an intensional conjunction acting like the fusion operation does on propositions: if \( a, b \) are propositions, \( a \cdot b \) is the proposition entailing exactly the propositions which may be derived from both \( a \) and \( b \). And these intuitions justify the use of relevant divisibility (which remains consistent with classical divisibility (1 0): \( i \mid j \) iff \( i \mid a \) and \( i, j \) have the same prime factors) to build relevance into the commutative monoid semantics.

### 1.4.2 Commutative Monoids Semantics.

Let \( S \) be the sentential variables of \( R_I \) and \( F \) the set of formulas built from \( S \) in the usual way. Let \( M \) be an arbitrary commutative monoid. Then a valuation \( V \) in \( M \) is a function \( v: S \times M \rightarrow \{t, f\} \). A possible interpretation \( I \) is a function \( I: F \times M \rightarrow \{t, f\} \).

Let \( Aa \), where \( A \in F \) and \( a \in M \), abbreviate \( I(A, a) = t \).

Generalizing from \( M \), the free commutative monoid, \( N^+ \) is proved characteristic for \( R_I \).

Let \( I \) be an \( R_I \)-interpretation in a denumerable commutative monoid \( M \). Then there is an equivalent \( I' \), i.e. an interpretation verifying exactly the same formulae, in the monoid \( N^+ \), the homomorphic image of \( M \). If \( M \) is finitely generated by a \( n \)-member subset \( M' \) of \( M \), then there is an \( R_I \)-interpretation \( I" \), equivalent to \( I \), in the finitely generated submonoid \( N_n \subset N^+ \). As a corollary, if \( I \) is hereditary, \( I' \) and \( I" \) are hereditary.

Taking \( N^+ \) as characteristic presents a difficulty because the vectors can be arbitrarily long. This problem is solved by the \textit{finite generator property}. Let \( n \) be the \textit{index} of a formula \( A \), i.e. the number of its subformulae, then \( A \) is a theorem of \( R_I \) iff \( A \) is valid in all \( n \) generators \( M \). This property thus transforms infinitely long vectors with no finite bound into vectors of uniform length \( n \) as sequences corresponding to a given formula \( A \). Hence, the property transforms the cardinality of the primes required for a refutation of a non-theorem \( A \).

Kripke's lemma and the finite generator property then suffice to prove decidability. Actually, decidability is expected since commutative monoids have a decidable word problem and, as we will see in the next chapter, in some way, the decidability procedure for \( R_I \) amounts to solving the word problem in commutative monoids.

The termination condition of the decision procedure, Kripke's lemma, is then proved equivalent to Dickson's Lemma and to the Infinite Divisor Property.

Let \( N_n \) be our free commutative monoid generated by the first \( n \) primes. Then,

**Dickson's lemma** (D): Let \( Q_n \subseteq N_n \) and suppose that for all \( a, b \in Q_n \), if \( a \mid r b \) (as well as \( a \mid a b \)) then \( a = b \). Then \( Q_n \) is finite.

**Kripke's lemma** (K): Let \( a_i \) be any sequence of members of \( N_n \) and suppose that for
all \( i, j \), if \( i < j \) then \( a_i \not| a_j \). Then \( a_i \) is finite.

**Infinite Division Principle (IDP):** Let \( Q_n \) be any infinite subset of \( \mathbb{N}_n \). Then there is an infinite subset \( R_n \) of \( Q_n \) and a member \( a \) of \( R_n \) s.t. for all \( b \in R_n \), \( a \mid | b \).

Obviously, IDP implies \( D \) since if \( Q_n \) is infinite, then some \( a \mid | \) infinitely many members of \( Q_n \), otherwise, \( Q_n \) is finite. And similarly, IDP implies \( K \).

As before, let \( \mathbb{N}_n \) be the subset of \( \mathbb{N}_+ \) whose members, abbreviated \( p^n_i \), can be represented in prime decomposition as products of the first \( n \) primes, and let \( < \mathbb{N}_n, | > \) be the partially-ordered set partially ordered by \( | \). We note that by definition of \( \mathbb{N}_n \), \( < \mathbb{N}_n, | > \) is isomorphic to \( < \mathbb{N}^n, \leq > \) by the mapping of each element of \( \mathbb{N}_n \) into the sequence of its exponents in \( \mathbb{N}^n \). That is, \( \leq \) is defined on \( \mathbb{N}^n \) by the product construction and agrees with divisibility as characterized before on \( \mathbb{N}^n \). And similarly for \( | \) and \( \leq_r \).

Since Kripke's lemma in its number-theoretic form is about order induced by divisibility on the positive integers, the theory of partial order not only throws some light on its significance, but it also guarantees the truth of the lemma and of its equivalent formulations for the free commutative monoid \( < \mathbb{N}_k, \cdot, 1 > \).

Let \( X \) be a partially ordered set (more accurately, we should write \( < X, \leq > \)). The partial ordering relation \( \leq \) is defined as usual, \( a \geq b \iff b \leq a \), and \( < \) is such that \( a < b \iff a \leq b \) and \( b \not< a \).

\( X \) satisfies the IDP if for all \( A \subseteq X \), there is a \( a \in A' \subseteq A \subseteq X \) s.t. \( A' \) is infinite and for all \( a' \in A' \), \( a \leq a' \).

**1.4.3 Finite Model Property.**

Extending the preceding results, \( R_I \) is shown to have the finite model property, \( (FM P) \), a stronger property than decidability, in the \( M \) semantics. \( R_I \) has the finite model property iff (i) there is an effective enumeration of the finite commutative monoids, (ii) every pertinent interpretation of \( R_I \) relative to a given formula \( A \) of \( R_I \) is given effectively and there are finitely many such interpretations in a finite \( M \), (iii) for a given non-theorem \( A \) of \( R_I \), there is some pertinent interpretation of \( R_I \) refuting \( A \) in some finite \( M \) where, in (ii) and (iii), pertinent means hereditary.

Another problem arises when the monoid \( \mathbb{N}_+ \) generated by the primes is taken as characteristic for \( R_I \) because arbitrarily high exponents are allowed on any particular \( p^n_i \) of \( \mathbb{N}_+ \) where \( p \) is a prime. This is solved by placing bounds on the exponents which are relevant to a refutation of a formula \( A \), i.e. in shrinking \( < \mathbb{N}^k, +, 0 > \) to \( i^k = < i^k, \oplus, 0 > \) where \( k \in \mathbb{N}_+ \). \( < i, \oplus, 0 > \) is the additive commutative monoid where \( i = \{ n: 0 \leq n < i \} \) and \( \oplus \) is defined as follows: for \( 0 \leq m, n < i \), if \( m + n \geq 1 \) then \( m \oplus n = i - 1 \), otherwise,

\[ \text{See addenda, Note 3} \]
m ⊕ n = m + n. That is i is the 1 generator commutative monoid \(< \mathbb{N}, +, 0 >\) bounded at \(i - 1\), and the elements of \(i^k\) are the \(k\)-places sequences of natural numbers \(< i \) on every coordinate substituted to the sequences \(> i \) of \(\mathbb{N}^k\).

The substitution of \(i^k\) to \(\mathbb{N}^k\) is guaranteed by the natural homomorphism \(h: < \mathbb{N}^k > \rightarrow < i^k >\) whose effect on the \(j\)th coordinate of \(a \in \mathbb{N}^k\) is s.t. if \(a_j \geq 1\), \((h(a))_j = i - 1\), else \((h(a))_j = a_j\).

In this way, the coordinates of elements that are greater than \(i - 1\) are finitized and bounded to \(i - 1\).

There is an effective procedure which applied to a given non-theorem \(A\) of \(R_I\) of index \(k\) finds a refutation of \(A\) in \(\mathbb{N}^k = < \mathbb{N}^k, +, 0 >\) or in the isomorphic \(< \mathbb{N}^k, \cdot, 1 >\): let \(a \in \mathbb{N}^k\) and call \(a\) critical for the formula \(A \in R_I\) if \(I(A, a) = f\) on an interpretation \(I\) and for all \(c \in \mathbb{N}^k\), if \(c \leq a\) then \(I(A, c) = t\). That is, \(a\) is critical iff it is minimal in the ordering \(\leq_c\) in the subset of elements of \(\mathbb{N}^k\) at which \(A\) is false.

By Dickson-Kripke-Meyer’s lemma, for all \(A \in R_I\) and for all interpretation \(I\) in \(\mathbb{N}^k\), the set of critical elements for \(A\) is finite. The shrinking lemma allows then to transform a refutation of \(A\) in \(\mathbb{N}^k\) into a refutation of \(A\) in \(i^k\). For all subformula \(B\) of \(A\) on all \(a \in \mathbb{N}^k\), \(I(B, a) = I(B(h(a)))\) where \(h(a) \leq_c a\) and \(h\) shrinks the large coordinates to \(i - 1\).

### 1.5 Dickson’s Lemma and Well-Orderings.

In its most common formulation Dickson’s lemma [Di13] says that a set of pairwise incomparable elements, i.e. elements such that \(x \not\leq y\) and \(y \not\leq x\), is finite. Given its historical importance, we first state the lemma before adding some more clearer equivalent formulations to those we have already seen.

**Lemma 1.1** Any set \(S\) of functions of type

\[
F = a_1^{a_1} a_2^{a_2} \cdots a_r^{a_r}
\]

contains a finite number of functions \(F_1, \ldots, F_k\) such that each function \(F\) of \(S\) can be expressed as a product \(F_i f\) where \(f\) is of the form (1) but is not necessarily in the set \(S\).

**Definition 1.2** Let \(S\) be a set of integers. Then \(S\) has the divisor property if every infinite subset of \(S\) contains two distinct numbers one of which divides the other.

Applying this definition to the lemma, it now says that if \(S\) has the divisor property and if \(S'\) is the set of all members of form \(a_1 a_2 \cdots a_n, n > 0, a_i \in S\), then \(S'\) has the divisor property. As we will see, in this form, it is then a special case of Higman’s theorem [Hi52] and can be reformulated as follows:

1. See addenda, note 4.
Lemma 1.3 If a set $N$ of integers does not contain any infinite subsets no element of which divides any other element, then neither does $P(N)$ the set of integers which can be written as products of elements of $N$.

That is, if for any sequence in the set of integers $N$, there exist $i, j$ $i < j < \omega$ such that $a_i \not| a_j$, then the set of all products has the same property.

1.5.1 Well-Orderings.

Definition 1.4

A partial order, $(PO)$, on a set is a reflexive, transitive and antisymmetric relation $\leq$. An set $S$ is well-partially ordered, $(WPO)$, if every non-empty subset $S' \subseteq S$ has a finite number of least elements. If the least element is unique, then $S$ is well-ordered.

Let $A$ be a PO set of sequences $a_i \in A$ s.t. every infinite subset $a_{i_k}$ contains a finite subset $b_1, b_2, \ldots, b_l$, and such that every $a_{i_k}$ is no less than at least one of the $b$'s. Then, $A$ is WPO. Let $V$ be the set of all vectors of finite length $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ constructed from the $a_i$'s. Then, defining the relation $\leq$ on $V$

$$(a_{i_1}, a_{i_2}, \ldots, a_{i_n}) \leq (a_{j_1}, a_{j_2}, \ldots, a_{j_l})$$

if there exists an isomorphic mapping $a_{i_1} \rightarrow a_{j_1}$ such that every $a_i \leq a_j$, i.e. each $a_i$ is less than or equal to its mate $a_j$, or more generally $[Ne49],[Hi52]$, defining a mapping function $f(x)$ of $\{1, 2, \ldots, k\}$ into $\{1, 2, \ldots, l\}$, i.e. $\mathbb{N}^k \rightarrow \mathbb{N}^l$, such that $f(x)$ is monotone increasing and $a_{i_j} \leq a_{j_k}$, then $[Ra54],[Er59]$,

Theorem 1.5 The set of such vectors is a well-partially ordered set.

Hence, Dickson's lemma can be formulated in yet another equivalent way:

Corollary 1.6 Dickson's lemma, is a case of a partial order in $\mathbb{N}^\omega$ defined by $a \leq b$ iff $a|b$. That is, the set of integers is well-partially ordered by divisibility. 2

1.5.2 Well-Quasi-Orderings.

Definition 1.7 A binary relation $\leq$ on a set $S$ is a quasi-ordering, $QO$, if it is reflexive and transitive.

As we have seen, a partial order is a $QO$ which is also antisymmetric. Antisymmetry prevents the situation in which both $a \leq b$ and $a \geq b$ hold. That means that a $QO$ does not account for these equivalences. Nevertheless, without loss of generality, we may restrict

1. where $a, b$, are vectors, finite products of $N$. 2
ourselves to $QO$ rather than $PO$ since $a \leq b$ and $a \geq b$ define an equivalence relation on $S$ and a $QO$ on $S$ induces a $PO$ on its equivalence classes by imposing an ordering on the equivalence classes of $[a] \leq [b]$ if and only if $a \leq b$.

The basic proof technique used in the theory of well-quasi-orderings relies on the notion of minimal bad sequences.

**Definition 1.8** Let $a_i = a_1, a_2, \ldots, a_n, \ldots$ be an infinite sequence of elements of a $QO$ set $A$. Then, $a_i$ is called *good* if there exist positive integers $i, j$ such that $i < j$ and $a_i < a_j$. Otherwise, the sequence $a_i$ is called *bad*.

**Definition 1.9** Let $A$ be a $QO$ set. Then, $A$ is well-quasi-ordered, $WQO$, if every infinite sequence of elements of $A$ is good. Equivalently, $A$ is $WQO$ if it does not contain an infinite descending chain (i.e. $a_0 > a_1 > \cdots > \cdots$), nor an infinite anti-chain (i.e. a set of pairwise incomparable elements).

By a reasoning similar to that which established theorem 1.5, if $A$ is $QO$, the set of all finite sequences of elements of $A$ is $WQO$, i.e.

**Definition 1.10** Let $A_n$ be the set of finite sequences of natural numbers $\leq n$, $A_1 = (a_0, \ldots, a_k) \in A_n$ and $A_2 = (b_0, \ldots, b_m) \in A_n$. Then $A_n$ is $WQO$, i.e. $(a_0, \ldots, a_k) \leq (b_0, \ldots, b_m)$ if there is an embedding function $f$ from $\{0, \ldots, k\}$ into $\{0, \ldots, m\}$, (an increasing mapping $f: k \rightarrow m$) such that $a_i \leq b_f(i)$.

For example, if we consider the natural numbers, we have $(1, 2, 3, 4) < (2, 1, 2, 4, 5, 2)$ but $(1, 2, 3, 4, 5) \not< (2, 1, 2, 4, 5, 2)$.

### 1.6 Higman's First Theorem.

The main theorem that we will prove in the next section is the following: *an abstract algebra with a finite set of operations has the finite basis property in a divisibility order if any generating set has*. It is the main theorem proved by Higman [Hi52]. Before coming to this theorem, we examine his other theorem on $WQO$ and some of their properties equivalent to the finite basis property. Doing so, we will add some equivalent properties to $D, K, M$ and throw some more light on their algebraic meaning.

**Definition 1.11** If $A$ is $QO$ and $B \subseteq A$, then $Cl(B) = \{a \in A | \exists b \in B, b \leq a\}$ is the *closure of $B$*. 

Definition 1.12 If \( A \) is \( QO \) and if every subset of \( A \) is the closure of a finite set, then \( A \) has the finite basis property.

1.6.1 Finite Basis Property.

Theorem 1.13 (Higman) Let \( A \) be a \( QO \) set. Then the following conditions on \( A \) are equivalent:

- (i) every closed subset of \( A \) is the closure of a finite subset;
- (ii) the ascending chain condition holds for the closed subsets of \( A \);
- (iii) if \( B \) is any subset of \( A \), there is a finite set \( B_0 \) such that \( B_0 \subset B \subset Cl(B_0) \);
- (iv) every infinite sequence of elements of \( A \) has an infinite ascending subsequence;
- (v) if \( a_1, a_2, \ldots \) is an infinite sequence of elements of \( A \), there exist integers \( i, j \) such that \( i < j \) and \( a_i \leq a_j \);
- (vi) there exists neither an infinite strictly descending sequence in \( A \), nor an infinite ascending antichain, that is, an infinity of mutually incomparable elements of \( A \).

(i) is the finite basis property, (v) defines the \( WQO \). As Higman remarks, (i), (ii), (iii) are equivalent by usual properties of closure operations and the condition that \( B_0 \subset B \) if \( a \in Cl(B) \) and \( a \in Cl(B_0) \). (iv), (v) and (vi) are obviously the equivalences proved in [Me 73b].

1.6.2 Some Universal Algebra.

Before giving a proof of Higman's second theorem, we explain condition (iii) of theorem 1.13. This will allow us to introduce some notions of operator algebras and free algebras that will be necessary in this chapter and in Chapter three. Classical references on these algebras are [Gr68], [Bi35], [Co81] and [Du89].

Operator Algebras.

Consider an algebra, \( A = (A, O) \), given by a set of elements \( A \) and a set of operators \( O \), \( A \cap O = \emptyset \) and call \( A \) an operator-algebra, or, following [Co81], an \( \Omega \)-algebra. The elements \( A_k \) of the algebra obtained by application of the \( n \)-ary operations in \( O \) are defined inductively as follows: \( A_0 = A, A_{k+1} = \{ z \in A | z \in A_k \text{ or } x = o a, a \in A_k, o \in O_n \} \), then, \( C_O(A) = \bigcup_{k=0}^{\infty} A_k \) is the closure of \( A \), i.e. it is the set of all subalgebras containing \( A \).

1. Henceforth called \( O \)-algebra.
CHAPTER 1. PROOF THEORY AND DECISION PROBLEM.

With $A \subseteq \mathcal{A}$ and $\mathcal{A} = C_0(A)$, $A$ is a generating set of $\mathcal{A}$. That means that in 1.13, condition $(iii)$, $B_0$ is the finite basis generating $\mathcal{A}$.

**Definition 1.14** An algebra $\mathcal{A} = (A, O)$ is ordered if $A$ is $QO$ and for all $o \in O$, $a, b \in A$ and $a_i, b_i$ finite sequences of elements of $A$,

- (i) if $a_i \leq b_i$, then $o(a_i) \leq o(b_i)$
- (ii) whenever $a \leq b_i$, if $a \leq o(b_i)$, then the $QO$ is a divisibility order.

(i) together with (ii) define a divisibility ordering.

**Free Word-Algebras.**

We may extend our definition of an operator algebra to that of a free algebra [Birk35]. Let $\mathfrak{A}$ be a class of $O$-algebras $(A, O)$. An $O$-algebra $\mathcal{A} = (A, F_A)$ where $F \in O$ is a free algebra freely generated by $A$ if, for any $O$-algebra $\mathcal{B} = (B, F_B)$ and any mapping $\phi: A \rightarrow B$, there exists a homomorphism $\psi: \mathcal{A} \rightarrow \mathcal{B}$ s.t. the restriction of $\psi$ to $A$ equals $\phi$, i.e. $\psi \upharpoonright A = \phi$.

**Theorem 1.15 (Birkhoff)** Let $\mathcal{A}$ be a class of $O$-algebras. Then for any set $S$, there is a $\mathcal{A}$-free algebra on $S$ and every algebra in $\mathcal{A}$ is the homomorphic image of a free $\mathcal{A}$-algebra.

Birkhoff provides a method to construct the word algebras based on congruences and direct products, but we will more simply follow [Co81] (see also [Hu85],[Le86]), and define a free algebra on a set $A$ as follows.

In $\mathcal{A} = (A, F)$, consider the set of operators $F$. With any $f \in F$ is associated a natural number $\alpha(f)$, called the arity of $f$, s.t. $\alpha \in F \rightarrow \mathbb{N}$, and to any operator $f$, corresponds some function $F_A: A^{\alpha(f)} \rightarrow A$. We may then see $F$ as the set of functions associated to each operator $f \in F$.

Our free $O$-algebra $\mathcal{A}$ can now be seen as a free $O$-Word algebra consisting of the set of all words or terms built on $A$ by $F_A$. Obviously, theorem 1.15 holds for $O$-Word algebras: an $O$-algebra $\mathcal{A}$ can be expressed as the homomorphic image of an $O$-Word algebra.

Consider the operator algebra $\mathcal{A} = (A, F)$ as a free word-algebra $\mathcal{T} = (T, F)$ where $T \subseteq A \cup F$ is the set of words built on $A$ by $F$. For all $f \in F$, and for all $t_1, \ldots, t_n \in T$, $f(t_1, \ldots, t_n) \in T$, where $n = \alpha(f)$. The operation of $f$ amounts to parenthesisation on
words of $T$, or to labelling of trees. Indeed, starting with $A = (A, F)$, the elements of its associated free algebra represent the way in which an element of $A$ is obtained by application of the operations $F$, and this can be seen as a tree in $T$.

**Construction of Word-Algebras.**

More precisely, let $\Sigma$ be a finite alphabet. A string $a$ of length $n \in \mathbb{N}$ is a function in $n \rightarrow \Sigma$. The set of all strings over $\Sigma$ is written $\Sigma^*$, and $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$. $\Lambda = \Sigma^0$ is the empty string, $a = 1 \rightarrow a \in \Sigma$ is the unit string.

$\Sigma^*$ is the free monoid generated by concatenation of elements of $\Sigma$, i.e. for $a, b, c \in \Sigma$, $a(bc) = (ab)c$, and $a\Lambda = a = \Lambda a$, and it is ordered by $a < b$ iff $a \leq b$ and $a \neq b$. Let $\mathcal{N}$ be the set of all closed under $\prec$. A tree $M$ such that $M \in D \rightarrow \Sigma$ is called a $\Sigma$-tree and is denoted by $D = D(M)$. Then, $D$ can be seen as the set of words from $\mathbb{N}_+$ (concatenated digits), called the positions, which label the nodes of a tree, i.e. define its structure. And $M$ attaches words built from $E$ to the nodes.

Let a function $\alpha: \Sigma \rightarrow \mathbb{N}$ be the arity function on $\Sigma$. With $\alpha$ defined, $\Sigma$ is a graded alphabet. Let $||M/a||$ be the width of a tree $M$, where for $a \in D(M)$, $M/a$ is the root of some subtree of $M$. Then $M$ is a $\Sigma$-term iff for all $a \in D(M)$, $||M/a|| = \alpha(M(a))$.

That means that a $\Sigma$-term on a graded $\Sigma$ is a $\Sigma$-tree such that the number of subtrees under each node equals the arity of the string at that node. We may best represent this operation as the construction of a tree corresponding to a $\Sigma$-term, for example, $(p \rightarrow .p \rightarrow p)$ where $\Sigma = \{\rightarrow, p\}$, $\alpha(\rightarrow) = 2$, $\alpha(p) = 1$.

If $T(\Sigma)$ is the set of all $\Sigma$-terms and $F \subseteq \Sigma$, if $\alpha(F) = n$ and $M_1, \ldots, M_n \in T(\Sigma)$ then $F(M_1, \ldots, M_n) \in T(\Sigma)$, i.e. every tree of $M$ is of the form $F(M_1, \ldots, M_n)$ where $F$ is the labelling or the parenthetization on terms. Hence, $T(\Sigma)$ is a $\Sigma$-algebra, and since the decomposition is unique, i.e. different terms do not represent the same element of the algebra, it is a completely free algebra.

Since we will need this notion of algebra in Chapter three, we may extend our free word-algebra $A = (A, F)$ to a free word-algebra with variables. Let $V$ be some set of objects (variables) such that $\Sigma \cap V = \emptyset$, and such that for all $v \in V$, $\alpha(v) = 0$. We can talk of the free $A$ - term algebra on a set $V$ such that $V \subseteq A$ and as before, in the definition of a free algebra, for any $B$, if $\phi: V \rightarrow B$, then $\psi: A \rightarrow B$ and $\psi \upharpoonright V = \phi$.

There is a mapping such that $(A, F) = (T, F)$ where $T \subseteq (\Sigma \cup V)$, with $V \in T$ and for all $f \in F$, for all $M_1, \ldots, M_n \in T$, $f(M_1, \ldots, M_n) \in T$.

Then, $T(\Sigma, V)$ is the set of terms with variables, and $T(\Sigma, V) = T(\Sigma \cup V)$. That means that, in a tree, a constant of $\Sigma$ denotes a value, and a variable denotes a term, i.e. its position in the tree.
1.7 Higman's Theorem.

The proof of Higman is rather involved but can be much simplified by resorting to Ramsey's theorem \[Ra30\] and the technique of minimal bad sequences of Section 1.5.2 due to Nash-Williams \[Na63\] which is actually a refinement of the existence proofs techniques which have been used in Algebra since the turn of the century. Ramsey's theorem is not essential to the proof, it just makes it easier. It is used here to stress the relationship between ordering and partition of sets on which we will come back.

Let \( N^2 \) be the set of all pairs of natural numbers, and let \([N]^2 = A_1 \cup A_2 \cup \cdots \cup A_n\) be a partition of \( N^2 \) into \( n \) sets. Let \([X]^2\) be the set of all 2-elements subsets of \( X \). Then,

**Theorem 1.16 (Ramsey)** There exist an infinite subset \( X \subset N \) such that \([X]^2 \subset A_i\) for some \( i, 1 \leq i \leq n \).

We first need some preparatory results, among them the crucial theorem of \[Me73b\].

**Definition 1.17** If \( A, B \) are \( QO \) sets, \( A \times B \) is \( QO \) iff \((a_1, b_1) \leq (a_2, b_2)\). And this is the case iff \( a_1 \leq a_2 \) and \( b_1 \leq b_2 \).

**Lemma 1.18** If \( A, B \) are \( WQO \), then \( A \times B \) is \( WQO \).

**Proof:** We must show that if \((a_1, b_1), (a_2, b_2), \ldots\) is an arbitrary infinite sequence of elements of \( A \times B \), and if there is no infinite antichain, then the sequence is good. Let \( a_n = a_1, \ldots, a_n \) be an infinite sequence of elements of \( A \). Then there exists a subsequence \( a_{n_i} \) of \( a_n \) such that if \( i, j \in X \subset N \) and \( i < j \), then \( a_{n_i} \leq a_{n_j} \) and is strictly increasing.\(^{3}\) By Ramsey's theorem, this sequence exists. For all \( i, j \in X \) we have only four cases to consider: \( i < j, j < i, i = j \) or \( i \not< j \) and \( i \not< j \).

Partition \([N]^2\) into four sets \( Y_i \) such that (i) \( i, j \in Y_1 \) if \( a_{n_i} < a_{n_j} \), (ii) \( i, j \in Y_2 \) if \( a_{n_i} > a_{n_j} \), (iii) \( i, j \in Y_3 \) if \( a_{n_i} = a_{n_j} \) and (iv) \( i, j \in Y_4 \) if \( a_{n_i} \not< a_{n_j} \) and \( a_{n_i} \not< a_{n_j} \).

Now, by definition 1.2, if \( A \) is \( WQO \), then every infinite sequence of elements of \( A \) is good. Suppose that \( A \) is not \( WQO \). Then by definition 1.3 there exists an infinite set \( a_n \) of incomparable elements of \( A \) or there exists an infinite descending chain \( a_0 > a_1 > \cdots \).

In both cases \( a_n \) is bad and \( A \) is not \( WQO \), and this eliminates cases (ii) and (iv). We are left with cases (i) and (iii). So we can select our increasing sequence \( a_n \).

Now the sequence being good, \( A \) must be good. \(^{3}\) By identical application of Ramsey's theorem, \( B \) is \( WQO \), that is, there exists an infinite sequence \( b_{n_j} \in B \) such that \( b_{n_i} \leq b_{n_j} \). Hence \((a_{n_i}, b_{n_j}) \leq (a_{n_j}, b_{n_j})\) and therefore our sequence \((a_1, b_1), (a_2, b_2), \ldots\) is good. \(^{3}\) by definition 1.47.

\(^{3}\) as suggested by \[Sc85\].
Corollary 1.19 \((A)^n\), the Cartesian product of \(A \times \cdots \times A\) is WQO.

The next theorem is often referred to as ‘Higman’s theorem’ in the literature. It should more appropriately be called ‘Higman’s lemma’. This is probably due to the fact that this theorem is sufficient for most purposes in the study of ordered sets. What we call ‘Higman’s theorem’ here was stated at the beginning of Section 1.6 and is actually a generalization of this theorem to operator algebras.

Nash-Williams [Na63] provided the definitive, short, simple and elegant proof of the following theorem using the now standard bad sequence technique. We follow his proof, and we rely also on Ramsey’s theorem and on lemma 1.18.

Theorem 1.20 If \(A\) is WQO, then \(S(A)\) the set of all finite subsets of \(A\) is WQO.

Proof: Assume that \(S(A)\) is not WQO. That is, there exist bad sequences of elements of \(S(A)\) such that for all \(i, j, a_i \leq a_j\).

We construct such a minimal bad sequence, \(a\), inductively: select an \(a_0 \in S(A)\) of minimal cardinality as the first element of a minimal bad sequence of elements of \(S(A)\).

Assuming the axiom of choice, among the sequences of \(S(A)\) starting with this \(a_0\), select a minimal \(a_1\) such that \(a_0, a_1\) are the first two members of the bad sequence, then select a sequence of \(S(A)\) with a minimal \(a_2\), and so on to obtain a sequence \(a = a_0, a_1, a_2, \ldots\)

The argument then consists in recursively extracting the first minimal element of each bad sequence. The sequence \(a\) is bad, hence \(\forall i \lt j \ a_i \leq a_j\). We can select from each of the \(a_i\) the first element. Let \(b_i = a_i - \{a_{i_n}\} \ (*)\) i.e. the \(a_i\) stripped off their first element, \(b_i\) is good. Otherwise, there would be some bad subsequence of \(b_i\), say \(b_{i_m}\), such that for all \(m, n, \ i_m \leq i_n\), and the sequence \(a_0, a_1, \ldots, a_{i(m-1)}, b_{i_m}, b_{i_n}, \ldots\) \((**)) would be bad. But this contradicts the construction of \(a_{i_m}\) which is minimal. Hence the sequence \((**)\) is good.

By lemma 1.18 and using an identical construction, \(b_{i_m} \leq b_{i_{m+1}}\), i.e. the subsequences \(b_{i_j}\) are increasing. Let \(B = \bigcup b_i\), then \(B\) is WQO. Since \(A\) is WQO, \(A \times B\) is WQO, hence there are \(i, j, i < j\ s.t.\ (a_i, b_i) \leq (a_j, b_j)\), and \(a_i \leq a_j\). Consequently, our sequence \(a = a_0, a_1, \ldots\) could not be bad. Hence, \(S(A)\) is WQO.

It is important to note that the theorem can be proved without the Axiom of Choice.

We can now come to Higman’s theorem and prove it in its full form. We have to show the action of the operators on the generating elements of the algebra and prove that the orderings are preserved. This is done by construction of the word algebra.

Theorem 1.21 An algebra \((A, O)\) with a finite set of operations \(O\) and a divisibility ordering has the finite basis property if any of its generating sets has the property.
Proof: By definition of divisibility ordering in 1.4 we have to prove that if $A$ is generated by a WQO set $A$, then $A$ is WQO. Theorem 1.20 already provides half of the proof. It remains to show that the action of the operators $O_i$ on $A$ is "conservative". 

Let $a_i \in A$ and suppose that $A$ is not WQO. Then there are sequences $a_i, b_i \in A$ such that $a_i \not\leq b_i$. As in theorem 1.20, construct an infinite minimal bad sequence $a_1, a_2, \ldots$ and show that $A$ is WQO. 

If the operation is 0-ary, $a_i \in A \cup O_0$. Since $O_0$ is finite and $A$ is WQO, by (vi) in theorem 1.13, the $a_i$ do not contain any infinite descending chain nor any infinite ascending antichain. Then there are only finitely many such $a_i$, and by 1.20, the set of these $a_i$ is WQO. 

If $O$ is $n$-ary, each of the $a_i$ corresponds to the application of some $O_i \in O$ to some subset of elements of $A$: $a_i = O_i(b_i)$. We have to show that, for any $a_i$, the sets $b_i$ are WQO. 

By theorem 1.20, this is the case. Let $n$ be the arity of $O$ in $a_i$ and let $B$ be the set of all sequences $b_i$ as in theorem 1.20 again. Then, $B$ being WQO, by corollary 1.19, $(B)^n$ is WQO.

**Corollary 1.22** Dickson's lemma.

Proof: Lemma 1.1 (or any of its equivalent formulations) is a case of the theorem. Indeed, the divisibility condition on a subset $N \in \mathbb{N}$ amounts to the finite basis property when $a \leq b$ is interpreted as $a \mid b$. Then, by (vi) in theorem 1.13, there is no infinite descending sequence of integers subsets of $N$.

Considering $\mathbb{N}^+$ as an algebra under multiplication we obtain a divisibility order by definition 1.14. Hence, by theorem 1.21, if $N$ has the finite basis property, any subalgebra generated by $N$ has the property, and the corollary follows.

Higman notes that his theorem also applies to words. This is now obvious by the construction of the word algebras: any subset of words over a finite alphabet which are pairwise incomparable in a divisibility ordering is finite. Meyer [Me73c] has also found such an interpretation of $D, K, M$ and gives a very simple explanation of the decision procedure based on this interpretation:

Let $\Sigma$ be the set of all subformulae of some proof candidate in a Gentzen system and let $w_i \in \Sigma^*$ be the sequences of subformulae as they appear on the left or right of $\vdash$. Then, as we have seen, the $w_i$ are finite in length. The proofs of proof candidates can be reduced to a certain normal form in which they are irredundant, and $w_i \in \Sigma^*$ is irredundant iff for some $n, j \in \mathbb{N}$ such that $n < j$, $w_n \not\leq w_j$, where $w_n \leq w_j$ iff for all $x \in \Sigma$, $\text{card}(x \in w_n) \leq \text{card}(x \in w_j)$. By $D, K, M$, the sequence $\Sigma^*$ is finite.

---

1. and by the equivalences of Theorem 1.13.
2. That is, to show that if $A$ is WQO, then applying the operations $Q \in O$ to its elements will not change the WQO of $A$. 
1.8 Kruskal’s Theorem.

Kruskal’s theorems [Kr60],[Kr72] on WQO trees corresponds to Higman’s theorems and are often presented as generalizations of the later. As in the preceding two sections, we consider two theorems and we show that Kruskal’s second theorem is essentially equivalent to Higman’s second theorem. The equivalence follows from their identical algebraic structure, and, actually, at the end of his paper, Kruskall adds a lexicon giving the correspondence between Higman’s vocabulary and his, but it is interesting to prove the equivalence. First we need some graph-theoretic definitions [Kr60],[Na63],[Na65] which can best be read keeping in mind what has been said about word-algebras in section 1.6.2.

Definition 1.23 A graph is an ordered pair of disjoint finite sets \((V, E)\) where \(V\) is the set of vertices \(\{v \mid v \in V\}\) and \(E\) is the set of edges \(\{e \mid e \in E\}\) and such that \(E \subseteq V \times V\).
A graph is connected if any point in it can be reached from any other point along the edges.
A sequence of edges in which each \(e \in E\) appears only once is a path.
A path is closed if the initial and terminal nodes coincide.

Definition 1.24 A tree \(t\) is a connected graph with at least two nodes, in which no path is closed, and with a designated vertex \(v_0(t)\), the root of the tree, such that for any \(v \in V\) in \(t\), i.e. \(v(t)\), there exists a unique \(v_0(t)\) v-path in \(t\).
A finite tree can also be seen as a partially ordered set \(t\) such that \(t\) has a smallest element, its root, and, for each \(t_i \in t\), the set \(\{s \in t_i \mid s \leq t_i\}\) is a totally ordered subset of \(t\).

Definition 1.25 A homeomorphism of \(t\) into \(t'\) is an embedding function \(f: v(t) \rightarrow v(t')\) such that for all \(V(t)\), the images under \(f\) of the successors of \(v\) follow distinct successors of \(f(v)\).

Definition 1.26 \(T\), the set of all trees is QO by the relation \(t \leq t'\) if and only if there is an embedding of \(t\) into \(t'\) (see 1.5.2). Moreover, \(t \times t'\) is QO by the relation \((t_1, t_2) \leq (t'_1, t'_2)\) if and only if \(t_1 \leq t'_1\) and \(t_2 \leq t'_2\).

What Kruskal’s theorem says, basically, is that if \(t, t' \in T\), the set of all trees, then, there is a homeomorphism \(f\) from the nodes of \(t\) into the nodes of \(t'\) such that \(f\) preserves the ordering. That is, for all \(v \in t\) and all \(v' \in t'\), \(v \leq v'\) if and only if \(f(v) \leq f(v')\).

Definition 1.27 A branch of \(t\) at \(v\) is a tree \(r\) such that \(V(r)\) is the set of vertices following \(v\) and \(E(r) = E(t) \cap (V(r) \times V(r))\).

Definition 1.28 A tree \(t\) is structured if every edge is oriented, pointing away from the root, and, at each vertex \(v \in t\), the edges above \(v\) are ordered.
A binary tree is a tree in which no more than two edges are incident at each vertex.
Let \( d(v) \), the \emph{degree} of any vertex of a structured tree \( t \), be the number of edges whose initial vertex is \( v \). A branch of \( t \) at \( v \) consists of \( v \) and all vertices and edges above \( v \). The root of the branch is \( v \), and the branch preserves the linear ordering of \( t \). The branches of \( t \) at \( v_1, \ldots, v_i \) are the branches above \( v \).

**Definition 1.29** A \emph{graded QO space} \( Q \) is a QO set \( Q \) with an infinite sequence \( Q_0, Q_1, \ldots \), of subsets of \( Q \). Let \( \cup Q_i = Q^+ \). Then \( Q \) is WQO if \( Q^+ \) is WQO and if there is some \( n \in \mathbb{N} \), called the \emph{total degree} of \( Q \) s.t. \( Q_n = Q_{n+1} = \ldots \), but \( Q_{n-1} \neq Q_n \).

**Definition 1.30** A tree \( t' \) over \( Q \) is a tree over \( Q^+ \): \( t': t \rightarrow Q \), if for all \( v \in t \), \( t(v) \in Q_{d(v)} \). That is, each vertex of \( t \) is labelled with an element of the appropriate subset of \( Q^+ \).

Let \( T(Q) \) be the family of all trees over \( Q \), since \( T(Q) \subseteq T(Q^+) \), it inherits its quasi-ordering from \( T(Q^+) \).

1.8.1 The Tree Theorem.

Kruskal’s Tree Theorem states that if \( Q \) is WQO, then \( T(Q) \) is WQO.

First, we consider the set of all trees. The proof is entirely similar to that of the corresponding theorem for sequences of integers.

**Theorem 1.31** \( T \), the set of all trees is WQO.

**Proof:** First, from theorem 1.20, considering a tree as an ordered set, it is obvious that the set of all trees is WQO, and the proof is exactly similar to the proof of theorem 1.20. Assume that \( T \) is not WQO, and construct a bad sequence of finite trees \( t_1, t_2, \ldots \) such that the cardinality of each of them \( |V(t_i)| \) is minimal.

Let \( b_i \) be the set of branches of \( t_i \) at the successors of its root \( v_0(t_i) \), that is \( b_i = t_i - \{ v_0(t_i) \} \), i.e. the \( b_i \) only extend up to \( b_i_{d(v_0)-1} \), where \( d(v) \) is the degree of the root \( v_0 \). And let \( B = \bigcup_{i=1}^{d(v_0)-1} b_i \).

Then, \( B \) is WQO. Otherwise there would exist some minimal bad sequence of \( t_i \in B \), say, \( s^n_i = s^i_1, s^i_2, \ldots \) such that \( s^n_i \in b_i \), \( (s^i_i \leq s^i_j \text{ and } i < j) \), and the sequence \( t_1, t_2, \ldots, t_{i-1}, s^i_1, s^i_2, \ldots \) would also be bad. But this would contradict the definition of \( t_i \) which is minimal by construction. Hence, there is no bad sequence \( s^n_i \in b_i \) such that \( s^i_i \leq s^i_j \).

Since any bad sequence of \( B \) would have the same subsequences, it follows that no sequences of \( B \) is bad. Hence \( B \) is WQO, and by theorem 1.20, the finite subsets of \( B \) are WQO, i.e. \( b_i \leq b_j \) for some \( i, j \) such that \( i < j \).


1.8.2 Labelled Trees.

We now consider the additional condition of labelling of the trees.

By definition of a structured tree and by the theorem, we know that all successors \( v_j \) of \( v_i \) are well-ordered. Let \( Q \) be a \( QO \) space, then, by definition 1.30, for each tree, there is a labelling function \( g: t \rightarrow Q \). Defining a structured labelled tree as a triple \((t, <, g)\) where \(<\) is the ordering on vertices and \(g\) is the labelling function, we can define the embedding of trees following definitions 1.26 and 1.28 as \((t, <, g) \leq (t', <', g')\) if there is an homeomorphic embedding \( f: t \rightarrow t'\) such that for all \( v_i, v_j \in t\), if \( v_i < v_j \), then \( f(v_i) <' f(v_j) \) and \( g(v_i) < f'(f(v_i)) \).

**Theorem 1.32** \( TS \), the set of structured labelled trees is WQO.

**Proof**: Continuing the proof from theorem 1.31, since \( b_i \leq b_j \) for some \( i, j \) such that \( i < j \), there is an isomorphic non-descending mapping \( f: b_i \rightarrow b_j \) such that for all \( r_h \in b_i \), \( r_h \leq f(r_h) \). Hence, there exists a \( h_r: r \rightarrow f(r) \).

Accordingly, there exists a \( h: t_i \rightarrow t_j \) such that \( h(v_0(t_i)) = v_0(t_j) \) and \( h = h_r \) on all vertices of all \( r_h \in b_i \) and for all \( v_0 \in r_h \), \( g_{h_r}(v_0) = g'_{h_r}(v_0) \). Therefore \( t_i \leq t_j \) and the sequence \( t_1, t_2, \ldots \) is good. More precisely, \((t_i, <, g) \leq (t_j, <', g')\).

Hence, we have proved that \( TS \), the set of structured labelled trees is WQO.

1.8.3 Higman \( \equiv \) Kruskal.

It remains to show that Higman's theorem is equivalent to Kruskal's theorem.

**Theorem 1.34** Higman's second theorem \( \equiv \) to Kruskal's Tree Theorem.

**Proof**: Half of the work has already been carried out in the presentation of word-algebras. 

\( \Rightarrow \) Consider the algebra \( A = (A, o) \). By Birkhoff's theorem there is an associated free algebra \( F = (B, O) \) such that \( A \) can be seen as the set of all terms built up by the operations \( O \) on \( B \). Indeed, let \( F = (B, O) \) and let \( (B, O) = (T, O) \) where \( T \subseteq (B \cup O)^* \) with \( B \subset T \), and for all \( o \in O \), for all \( M_1, \ldots, M_{o(o)} \in T \), \( o(M_1, \ldots, M_{o(o)}) \in T \).

From section 1.6.2, \( F = (B, O) \) defines the set of all \( A \)-terms on \( B \), and, as such, is equivalent to the set of trees on \( (B \cup O) \). By Birkhoff's theorem, \( F \) is the homomorphic image of \( A \).

By Kruskal's theorem, if \((B \cup O)\) is WQO, then the set of trees on \((B \cup O)\) is WQO. Hence \( F = (B, O) \) is WQO and \( A = (A, O) \) is WQO.
\begin{quote}
CHAPTER 1. PROOF THEORY AND DECISION PROBLEM.
\end{quote}

Starting with $A = (A, 0)$, by Higman's theorem, if $A$ is WQO, so is $A$. By Birkhoff's theorem $A$ is the homomorphic image of a free algebra $F$, and homomorphism preserves WQO. Hence if $A$ is WQO, $F$ is WQO.

It suffices then to show that the tree space is WQO. Again this is done by constructing the free algebra and its corresponding word algebra. As we have seen in section 1.6.2, each word corresponds to a tree and the set of words corresponds to the family of trees. $F$ being WQO, so is the tree space. Hence $T(Q)$ is WQO.

$T(Q)$ being WQO, then for any infinite sequence $T_i$ of finite members of $T(Q)$, there exist $i, j, i < j$ such that $T_i < T_j$, i.e. the sequence is WQO by embeddability. Hence, there is no infinite set of pairwise non-embeddable finite trees. And, of course, the equivalences of theorem 1.13 hold for trees.

1.9 Finitary Unconstrained Algebras.

This theorem of Kruskall has received a lot of attention in Finite Combinatorics and in "Reverse Mathematics" as we will see. Indeed, by König's lemma, there is an easy corollary due to H. Friedman [Si85] who proved the following finitization of the theorem [Dr87]:

**Corollary 1.35 (Friedman)** For every sequence of $n$ finite trees $T_i$, $1 \leq i \leq n$, with $|T_i| \leq c_i$, where $|T_i|$ is the cardinality of the nodes of $T_i$, for any $c \in \mathbb{N}^+$, there exist $i, j$ such that $i < j \leq n$ and $T_i \leq T_j$.

Coming back to the decision procedure of the monoid semantics and the finite model property, in the refutation of a non-theorem, the set of critical elements, i.e. the subset of elements of $\mathbb{N}^k$ at which the formula is false, is finite. These vectors are of uniform finite length and shrunken from $\mathbb{N}^k$ to $i^k$. Using Meyer's interpretation of the decision procedure in terms of words or sequences of subformulae and what has been said before about the construction of a word algebra, the finite irredundant sequences of words corresponding to the vectors in $i^k$ correspond exactly to a finite set of finite bounded trees to which the corollary applies.

To conclude, most of what we have seen in this chapter is summarized in Meyer's construction of finitary unconstrained algebras [Me73c, lemmas 4, 5]. The proof given here shows the equivalence with the preceding corollary.

Consider an operator algebra as before. Then $\mathcal{F} = \langle P, F, O \rangle$, where $F$ is the set of objects constructed from atoms $P$ by operations $O$. The algebra is absolutely finitary if $P$ and $O$ are finite, and it is unconstrained if each element of $\mathcal{F}$ is built in only one way. That is $\mathcal{F}$ is free. $\langle F, \leq \rangle$ is a PO and satisfies the DCC. In order to prove Kripke's lemma or IDP Königly, for all $x \in P$ in $\mathcal{F}$, define the level $k$ of $x$ as follows. $k = 0$ if
$x \in P$, and if $x = o(x_1, \ldots, x_n)$, where $o \in O$, the level of $x = k + 1$, where $k$ is the greatest level of the $(x_1, \ldots, x_n)$. That means that $\mathcal{F}$ is ordered in layers, every member has a level $k \in \mathbb{N}$. Let $F_k$ be the collection of members of $\mathcal{F}$ of level $k$, then $F_k$ is finite, and $\mathcal{F} = \bigcup F_k$. By construction of the word algebras we can consider $\mathcal{F}$ as a rooted tree made up of the $F_i$ spreading from $d(v_0) - 1$. The size of these trees is bounded by $k$: there is some $k$ such that for each $F_i$, $|F_i| < k_i$. In other words, the size of a tree is a function of $k$, and there is a $F_n$ maximum in the sequence for which Kripke’s lemma or IDP holds. Indeed, suppose that there is no such $k$ and no such $n$ s.t. $F_n$ satisfies the IDP. Then, using a compactness argument, there is a $k$ and an $n + 1$ such that IDP holds. If not, there is a $k$ and a $n + 2$, ... In this way we obtain an infinite sequence of trees such that for no $k$ and for no $n + j$ the IDP holds. But our trees are all finite. Hence, by König’s lemma we have an infinite finitely branching tree. That means that there is an infinite branch such that IDP does not hold. But this contradicts the IDP. Consequently, IDP holds in $\mathcal{F}$, ending the proof.
Chapter 2

Constructivity and Complexity.

2.1 Summary.

All proofs of theorems of chapter one are mere existence proofs. All of them use similar techniques. Finiteness is then guaranteed, but practicality or feasibility is not.

In chapter two, the decision procedure and finiteness condition are investigated from a constructive point of view through Hilbert's basis theorem in the theory of polynomial rings. In ring theory, Hilbert's theorem has the same finiteness consequence as Dickson's lemma. A point already noted by Dickson.

G. Hermann [He26] had already attempted to give a constructive proof of Hilbert's theorem. She partly succeeded, and any later developments relied on her original work. Seidenberg finally gave a complete constructive proof of Hilbert's theorem in the 70's. We rely on his various presentations, and from his results, we can convince ourselves that, given the appropriate translation, all the theorems of chapter one have a constructive version. From a constructive proof of Hilbert's theorem, a constructive solution of the word problem for commutative semigroups and monoids is obtained. Hence, an alternative constructive decision procedure of some of the logics introduced in chapter one modulo their Urquhart's translation into Thue-systems and semigroups. This detour through the theory of polynomial rings will allow us to relate our work to other fields of research like Unification theory.

With Dickson-Kripke-Meyer lemma we have a finiteness or termination condition. But if we can count, finite is nothing more than the point we have reached at the time we were tired to add one to $n$. And time permitting this could well be finite but arbitrarily large. Complexity theory tells us what is feasible and what is not. Complexity results for relevant logics are entirely due to A.Urquhart, and, for our purpose, one of his main contributions, on which we will rely, is the appropriate translation of the logic into Thue-systems and semigroups. It then suffices, as it is most often the case in complexity theory, to reduce
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

The problem to some other problem whose complexity is known. Interestingly, some of these results rely on those of Hermann and Seidenberg in commutative ring theory. Since we are mainly concerned with practical computability and automated theorem proving in logics whose complexity had long been suspected on empirical evidence, an interesting question was to investigate the level of complexity in practical cases. Some complexity bounds are obtained in the case of a restricted number of variables in the logic. Moreover, the upper complexity bound can trivially be lowered, but without providing any improvement to his intractability.

To conclude this chapter, the results of this first part are related to some recent results in the foundation of mathematics, particularly in ‘Reverse Mathematics’ and in weak formal theories of mathematics. This allows us to throw some light on the strength of the theorems we have discussed in the preceding chapters and at the same time to settle Kripke’s conjecture about the provability of his decision procedure.

2.2 Hilbert’s Finite Basis Theorem

Except for Meyer’s finite model property for $R_I$, the theorems we have seen in the preceding chapter are non-effective. If we have a proof that some structure is finite, none of these theorems actually guarantees that effective computation is possible on these structures.

In the middle of the nineteenth century, Cayley had studied homogeneous polynomials with arbitrary constant coefficients of degree $n$ in $m$ independent variables which he called quantics. For a long time, the essential question of the existence of a finite fundamental such system remained ignored, and, when it was raised, Cayley claimed that there was no such finite system. Gordan (1868)$^1$ proved him wrong and gave a constructive proof of the existence of finite fundamental invariants and covariants for systems of binary quantics (see [Be45], [Gr03]).

Theorem 2.1 (Gordan) The number of irreducible solutions in positive integers of a system of homogeneous linear equations is finite.

Hilbert [Hi90] gave a more general proof of the theorem which applies to any number of variables. But, what is important to note, is that, contrary to Gordan’s proof, Hilbert’s proof gives no indication as to the actual determination of the finite system. It is a mere existence proof as emphasized by Gordan’s reaction “This is not mathematics, it is theology”. It is only a proof of the existence of an entity without actually exhibiting it nor providing an effective method to do so. Hilbert’s proof marked the opening of a new chapter in algebra with the recognition and almost general acceptance of abstract

---

$^1$I owe these references, communicated by M. Dunn, to M. Wheeler of Indiana University.
non-effective methods. Bell [Be45] notes that up to 1945, a constructive proof of the basis theorem had not been given. Actually, we will see that this had been almost completely done by G. Hermann [He26], a student of E. Noether.

We mentioned earlier that the FMC, and the properties equivalent to it, on a PO set is Noetherian (section 1.4.2). We now turn to this property.

Noetherian Rings.

Let us recall that a ring \( R \) is an additive commutative group together with an associative and distributive multiplication operation. If \( I \) is an additive subgroup of \( R \) such that for all \( a \in I \), for all \( r \in R \), \( ar, ra \in I \), then \( I \) is an ideal.

A commutative ring \( R \) has the basis property if every ideal in \( R \) is generated by a finite number of elements of \( R \). The following finite basis theorem, corresponding to what is called the Hilbert's "finite basis", defines the important notion of "being Noetherian". The essential properties of the finite bases that we have seen in chapter one reappear now in new guises. A classical proof is given in order to make clear some of the results which will be stated later without proofs.

**Theorem 2.2** A ring \( R \) is noetherian if the following conditions are equivalent:

1. every ideal in \( R \) is finitely generated,
2. any ascending chain of ideals is finite,
3. every ideal in \( R \) has a maximal element.

**Proof:**

1 $\rightarrow$ 2. Consider an ascending chain of ideals \( I_0 \subseteq I_1 \subseteq \cdots \subseteq I_i \subseteq I_{i+1} \subseteq \cdots \).

Let \( I \) be the set of all elements in at least one of the \( I_i \). Then, \( I \) is an ideal. Let \( a, b \in I \), \( a \in I_n, b \in I_m \). Then \( a \) is in \( I_n \) as well as in all \( I_{n+i} \). Similarly for \( b \).

If \( I_m \subseteq I_n \), then \( a, b \in I_n \), and by definition of ideals, \( a + b, ab, ac \in I_n \), and so are in \( I \). Hence, \( I \) is an ideal. Let \( (a_1, \ldots, a_n) \in I \) be the finite set of generators of \( I \). Since \( (a_1, \ldots, a_n) \in I_j \), for some \( j > i, I \subseteq I_j \). But \( I_j \subseteq I \). Hence, \( I = I_j = I_{j+1} \) and \( R \) has the ascending chain property.

2 $\rightarrow$ 3. Let \( I_n \subseteq I \). \( I_n \) is finite and maximal. Otherwise, there is an \( I_m \) such that \( I_n \subseteq I_m \) and \( I_m \) is maximal. Otherwise, \( I_m \subseteq I_{m+1} \) etc... Hence, there is a \( I_i \) not maximal such that \( I_i \subseteq I_{i+1} \) and the chain is infinite, contradicting the hypothesis.

3 $\rightarrow$ 1. Let \( I \) be an ideal and \( I \subseteq I \). Let \( a_0 \in I \). If \( a_0 \) is not the maximal element generating \( I \), then, there is an \( a_1 \) such that \( (a_0, a_1) \) generates \( I \) and \( a_1 \) is maximal.

Hence, there is a chain of generators \( (a_0) \subseteq (a_0, a_1) \subseteq \cdots \). This chain has a maximal element, \( (a_0, \ldots, a_n) \). Hence, the finite set \( (a_0, \ldots, a_n) \) generates \( I_i \).

---

4. See Addenda, note 6.
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

Hilbert's Basis theorem.

Theorem 2.3 (Hilbert) If R is noetherian, then the polynomial ring R[X] is noetherian.

Hilbert's Theorem and Dickson's Lemma.

Without entering into any details, let us recall a result of Cohen and Kaplanski [Co46] that establishes clearly the connection between Hilbert's theorem and Dickson's lemma.

Suppose that any element of R, an integral domain (a commutative ring with unity and no divisor of zero, i.e. a ∈ R s.t. for some b ≠ 0, ab = 0 or ba = 0), can be expressed as a product of primes p_i. It is the case only if R is Noetherian. Remembering lemma 1.1, and applying it to an integral domain (the set S is now R, and any element is F), that immediately means that Dickson's lemma defines a Noetherian condition. It is interesting to note here that if factorization is unique, then a ring R in p_1, ..., p_n is a free semigroup with p_1, ..., p_n as generators. That means that the ideals of R are finitely generated, they ACC holds for the ideals of R, that is, there is no infinite AC of ideals (under subset).

Hence, R satisfies Higman's first theorem (theorem 1.13) and D, K, M.

Continuing with [Co46], let M_1, ..., M_h be the maximal ideals of R, and suppose that a_i, b_i are products of primes belonging to M_i, and such that a_1, ..., a_h = b_1, ..., b_h. Then, each of the a_i is pairwise associated with each of the b_i. If M is a maximal ideal with n primes (n > 1) in an integral domain R, every element of M^{n-1} is divisible by every prime in M. Then, for any sequence x_i ∈ R, it is possible to select a subsequence y_i such that y_i | y_{i+1}.

As a corollary, there cannot exist an infinite sequence a_i with the ideal (a_1, ..., a_r) always properly contained in (a_1, ..., a_{r+1}). Hence, by theorem 2.2, R is Noetherian.

To be short, using a well-known corollary of Hilbert's theorem, it is easy to see the relationship of Hilbert's theorem with Higman's second theorem (theorem 1.21): any finitely generated algebra over a field is Noetherian. Hence, what Higman's theorem defines is the condition for an algebra to be Noetherian, and his finite basis property is essentially equivalent to Hilbert's basis. And this equivalence was expected since Dickson's himself remarked that his finiteness condition can be obtained from Hilbert's theorem. It remains to show that R[X] is Noetherian, and this can be done constructively.

2.3 Constructive Approach.

Bell [Be45] who is quoted above apparently forgot the work of G. Hermann who had explicitly tackled the problem of providing Hilbert's theorem with a constructive proof. Interestingly, [He26] german title translates into "the question of finitely many steps in the
Essentially her results amount to showing that if \( I \) and \( J \) are two finitely generated ideals, one can find a finite set of generators for \( I \cap J \) or \( I : J \) (the quotient of \( I \) by \( J \)), or decide if some member of the ring is also a member of a finitely generated ideal. Moreover she provides some bounds on the complexity of solving systems of polynomial equations.

Nevertheless, the classical characterization of "being noetherian", that is, to have the ACC and every ideal finitely generated, may still be too strong from a constructive point of view. Indeed, the ascending chain of ideals \( I_0 \subset I_1 \subset \cdots \) in a polynomial ring \( K[X_1, \ldots, X_n] \) is finite, but we can always select a \( m < n \) and construct the chain \( I_0 \subset I_1 \subset \cdots \subset I_m \subset I_n \) [Se85].

Or, following [Ri74], consider \( I_n \), the set of integers \( \{0, X\} \), where \( X \) represents the multiples of the least positive \( k \leq n \) such that the sequence 0123456789 occurs in the first \( k \) digits of the decimal expansion of \( \pi \). Then, \( I = \bigcup I_n \) is an ideal in the ring of integers. Obviously, the problem is that a finite set of generators for \( I \) has still to be found. And the same example shows that, even though it exists, no asymptotic bound can be given to the AC of ideals.

The solution proposed by Seidenberg in several papers is that if some bounds are put on the degrees of some basis elements of the ideal \( I_n \), then a bound can be placed on the length of the ascending chain. This solution is important because, as we will see later, there is no hope to deal with complexity efficiently if the problems are not bounded. But even so, there is no guarantee of feasibility.

Another solution [Ri74] is to admit a minimal condition for "being Noetherian" which is still equivalent to the ACC, and which is constructive, for example, finding two finitely generated ideals that are equals, or deciding for a given element if it is in a given ideal. Then, if \( P \) is a property of a ring equivalent to being Noetherian, and assuming that it is shown, for example, that if \( R \) has property \( P \) then \( R[X] \) has property \( P \), then a constructive version of Hilbert's theorem has been proved.

### 2.3.1 Seidenberg Construction.

In order to convince ourselves that a constructive proof exists, we will follow the main steps of Seidenberg's construction [Se72],[Se74],[Se74a]. Then, the minimal requirement for constructivity that we have just seen will suffice for our purpose. The point in going through the exposition of these generally ignored results is to show that some claims regarding constructivity made below are supported.

Again, in order to be self-contained, we start by recalling some algebraic notions (See for example [Bh86]).
Polynomials.

Let $S$ be a set, and $\mathbb{N}$ the integers seen as a monoid. Then, $N(S)$, the set of functions $f:S \rightarrow \mathbb{N}$ is a free multiplicative abelian monoid. Every element or term $x_i \in N(S)$ is uniquely represented as a product $\prod x_i^{f(x)}$ with $\prod x_0$ the unit of the monoid.

Let $A$ be a commutative algebra. Then, in the polynomial algebra $A[N(S)]$ of $S$ over $A$, every element is uniquely represented as a sum $\sum a_f \prod x_i^{f(x)}$ where the terms multiplied by non-zero coefficients $a_f \in A$ and called monomials form the basis of $A[S]$ over $A$. Let $S$ be the set $\{x_1, \ldots, x_n\}$. Then, $A[S] = A[x_1, \ldots, x_n]$ is the polynomial ring in $x_1, \ldots, x_n$ indeterminates over $A$, where the polynomials are written as sums $\sum a_f x_1^{f_1} \cdots x_n^{f_n}$ on all n-tuples of integers $f_1, \ldots, f_n \geq 0$, i.e. $a_0x^n + a_1x^1 + \cdots + a_nx^n$. From now on, when no confusion is possible we will write $A[X]$ for $A[X_1, \ldots, X_n]$.

Modules.

**Definition 2.4** Let $R$ be a ring and $M$ an additive abelian group. $M$ is a module if there exists a mapping $\phi: R \times M \rightarrow M$ s.t. if $a \in R$ and $x \in M$, $\phi(a, x) = ax$, and the usual axioms of distributivity and associativity are satisfied.

For example, with $R = \mathbb{Z}$, an abelian group is a $\mathbb{Z}$-module, and, with $K$, a field, a $K$-vector space is a $R$-module.

**Definition 2.5** A submodule of $M$ is a subgroup of $M$ closed under multiplication by the elements of $R$. If $f: M \rightarrow N$ is a module homomorphism, then the kernel of $f$, $\ker(f) = \{x \in M: f(x) = 0\}$ is a submodule.

**Definition 2.6** A module $M$ is finitely generated if it is generated by some finite subset $S$ of elements of $M$. If $x_1, \ldots, x_n$ generate $M$ and $r \in R$, then $\{M = r_1x_1 + r_2x_2 + \cdots + r_nx_n | r_i \in R\}$.

**Definition 2.7** $S$ is a basis of $M$ if $S$ generates $M$ and $S$ is linearly independent. $M$ is a free module if it admits a basis. If $M$ is of finite rank (has a finite number of elements in its basis) and if there is a map $f: M \rightarrow N$ s.t. $\ker(f)$ is finitely generated, then $N$ is finitely presented or related.

That is, $M$ may be specified by a finite number of generators and a finite number of relations.

**Constructive Proof.**

Consider polynomials $f_1, \ldots, f_s$ in $R[X_1, \ldots, X_n]$ and $(g_1, \ldots, g_s) \in R[X]$. Then, the set of polynomials of degree $d \leq n$, $f_1g_1 + \cdots + f_sg_s$ forms a $R[X]$-module whose finite basis can
be constructed in a number of steps depending only on \( n, s \) and \( d \).

**Lemma 2.8 (Seidenberg)** (i) If the finitely generated ideals of \( R \) are finitely related, then every finitely generated submodule \( M \) of a finite rank free \( R \)-module \( N \) is finitely related, and a finite set of generators for the submodules \( M' \) of \( M \) consisting of the polynomials of degree \( n - 1 \) can be found.

(ii) If in addition, \( R \) satisfies the ACC,

(a) then for any chain \( M_1 \subset M_2 \subset \cdots \) of finitely generated submodules of a finite rank free \( R \)-module \( N \), one can find an integer \( i \) s.t. \( M_i = M_{i+1} \),

(b) and \( R[\mathcal{X}] \) satisfies the ACC.

**Lemma 2.9 (Seidenberg)** Let \( I = (f_1, \ldots, f_s) \) be such that a finitely generated ideal in \( R[\mathcal{X}] \) (as in lemma 2.8) and \( n \) a bound on the degree \( f_i \) (\( i = 1, \ldots, s \)). Then, one can construct a finitely generated submodule \( M \) of the \( f_i \) s.t. \( M \) generates \( I \). Consequently, \( I \cap R[\mathcal{X}] = M \) and for any \( m \), one can construct \( I \cap R[X_1, \ldots, X_m] \) and, in particular, \( I \cap R \) in a number of steps depending on \( n, s, \) and \( d \).

We simply note that given the ACC and [He26], an integer \( i \) s.t. \( A_i = A_{i+1} \) can be found. Then, by properties of \( R \)-isomorphisms and quotient free modules [Bh86], one can construct the finitely generated submodules of \( M \) isomorphic to the quotient of a finite rank free module \( N \) by a finitely generated ideal.

Construct the module \( M' \) of polynomials in \( M_i \) and the chain \( M_1 \subset M_2 \subset \cdots \), and consider the \( A_1 \subset A_2 \subset \cdots \) generated by the coefficients of the \( X_n \) in the \( M_i \). Then \( A_i M_i \) is a submodule of \( M \). By induction on the elements of \( M \) and the degree of \( X \), \( M_1 \subset M_2 \subset \cdots \subset M_i = M_{i+1} \) can be constructed.

The set of leading coefficients of \( X_i \) in the polynomials \( f \) form an ideal \( L_i(I) \) which can be constructed in a number of steps depending on \( n, s, d \), and \( i \). These coefficients ideals are obviously such that \( L_0(I) \subset L_1(I) \subset \cdots L_n(I) \subset L_{n+1}(I) \subset \cdots \), and \( L(I) = \bigcup L_i(I) \).

We can now follow the main steps of Seidenberg’s proof [Se74].

**Theorem 2.10 (Hilbert)** If \( R \) satisfies the ACC and its finitely generated ideals are finitely related, then so does \( R[\mathcal{X}] \).

**Proof:** Let \( I \) be a finitely generated ideal in \( R[\mathcal{X}] \), then there exists a finitely generated \( R \)-module \( M \) of \( R[\mathcal{X}] \) and an integer \( n \), the maximal degree of elements of \( M \) s.t. \( M \) generates \( I \) as an ideal in \( R[\mathcal{X}] \).

For \( M_i \) given, construct \( M_{i+1} \) as follows: the elements of \( M_i \) of degree \( n \) form a finitely generated \( R \)-module \( N_i \). Set \( M_{i+1} = M_i + X N_i \), then \( M_{i+1} \) is a finitely generated module generating \( I \) as an ideal in \( R[\mathcal{X}] \), and consists of polynomials of degree \( \leq n \).
Then, \( M_1 \subseteq M_2 \subseteq \cdots \subseteq R \)-module of rank \( n + 1 \). Hence \( M_i = M_{i+1} \).

Let \( I_1 \subseteq I_2 \subseteq \cdots \) be a chain of finitely generated ideals in \( R[X] \). One can compute the leading coefficients ideals \( L(I_1) \subseteq L(I_2) \subseteq \cdots \) and bases for the \( I_i \) yielding \( L(I_i) \). We have to show that \( I_i = I_{i+1} \).

Let \( n_i \) be the maximum degree of such a basis and assume \( n_{i+1} > n_i \). Consider first \( n_1 \) and \( L_{n_1}(I_{i_1}) \subseteq L_{n_1}(I_{i_2}) \subseteq \cdots \). Then, one can find an \( i_1 \) s.t. \( I_{i_1} \cap (R + \cdots + RX^{n_1}) = I_{i_1+1} \cap (R + \cdots + RX^{n_1}) \). Hence, s.t. \( L_{m}(I_{i_1}) = L_{m}(I_{i_1+1}) \) for \( m \leq n_1 \).

Next, consider \( n_{i_1}+1 \) and the chain as before, \( L_{n_{i_1}+1}(I_{i_1+1}) \subseteq \cdots \), and find an \( i_2 > i_1 \) s.t. \( L_{m}(I_{i_2}) = L_{m}(I_{i_2+1}) \) for \( m \leq n_{i_1}+1 \).

Repeating the construction, consider \( L_{n_{i_1}+1}(I_{i_1+1}) \), and find a \( j \) s.t. \( L_{n_{i_1}+1}(I_{i_1+1}) = L_{n_{i_1}+1}(I_{i_1+1}) \) (for \( k = j + 1 \)).

Since \( L_{n_{i_1}+1}(I_{i_1+1}) \subseteq L_{n_{i_2}+1}(I_{i_2}) \subseteq L_{n_{i_2}+1}(I_{i_2+1}) \subseteq L_{n_{i_2}+1}(I_{i_2+1}) \), by (*) we obtain \( L_{n_{i_2}+1}(I_{i_2+1}) = L(I_{i_2+1}) \).

Hence, for any \( m' > m_{i_1}+1 \), \( L_{m'}(I_{i_k}) \subseteq L_{m'}(I_{i_k+1}) \subseteq L_{m'}(I_{i_k+1}) \) and so, for all \( m \), \( L_{m}(I_{i_k}) = L_{m}(I_{i_k+1}) \).

Since \( I_{i_k} \subseteq I_{i_k+1} \) and \( I_{i_k} = I_{i_k+1} \).

It follows that

\[ L_{n_{i_2}+1}(I_{i_2+1}) = L(I_{i_2+1}) \]

2.3.2 Computing with Polynomials.

Not only Hilbert's theorem has a constructive proof, but it can be actually constructed and the finite basis can eventually be computed. Computation here amounts to solving systems of Diophantine equations and finding the finite bases.

Basically, this is what is done in some other fields of computer science logic, for example, in Unification Theory, in Rewriting Systems or in the computation of the Gröbner bases. One result in the field of Automated Theorem Proving based on the computation of finite bases is Wu’s algorithm and the geometry theorem prover based on it. It is claimed that most theorems of Euclidean Geometry are proved mechanically. And this is surely an achievement. But, considering the complexity of solving polynomial equations and computing finite bases, one may wonder if, after all, the theorems proven are really very hard to prove, or if they are only the easiest instances of much harder problems.

In [St78] M. Stickel observed that the pattern matching problem for multisets can be reduced to the problem of finding positive solutions to sets of homogeneous linear Diophantine equations over \( \mathbb{N}^+ \). Siekmann and Liversey [Si89] showed the relationship between the associative-commutative unification problem and the same problem. They proposed a reduction to sets of inhomogeneous linear Diophantine equations.

In AC unification, the set of most general unifiers is finite and corresponds to the set of solutions of Diophantine equations. Finiteness is guaranteed by Dickson's lemma or

---

4. That means that, as noted above, by Lemma 2.8 (ii) (a), and by induction on the rank of \( N_i \), for any chain \( M_i \), one can find an \( i \) s.t. \( M_i = M_{i+1} \).
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

Hilbert’s theorem.

The problem of computing in the monoid $\mathbb{N}^+$ or in a ring thus amounts to solving systems of linear Diophantines equations, i.e. polynomials. And there are efficient algorithms to do so (see [Hu78],[He87],[Gu85],[Gi64]).

From the examples of modules given in definition 2.2 we note that we can work in the module of linear vector spaces over a computable field [Gi66],[Gu85].

Consider the vectors $\mathbf{a} = (a_1, \ldots, a_m) \in \mathbb{N}^m$ and $\mathbf{b} = (b_1, \ldots, b_n) \in \mathbb{N}^n$, $(m, n \geq 0)$, where the $\mathbb{N}^i$ are seen as subsets of $\mathbb{R}^n$, the vector space of all the rational numbers over the rationals. In what follows, we only consider homogeneous systems of equations and $\mathbb{N}^i$ is always $\mathbb{N}^i - \{0\}$.

Then, $\mathbf{a}$ and $\mathbf{b}$ determine a homogeneous linear diophantine equation $\mathcal{H}(\mathbf{a}, \mathbf{b}) = a_1x_1 + \cdots + a_mx_m = b_1y_1 + \cdots + b_ny_n$ written as $\mathcal{H}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{m} a_ix_i = \sum_{j=1}^{n} b_jy_j$ which has a solution iff $n > 0$ and $m > 0$.

For example, if $\mathbf{a} = (1, 2)$ and $\mathbf{b} = (1, 2, 3)$, then $\mathcal{H}(\mathbf{a}, \mathbf{b}) = x_1 + 2x_2 = y_1 + 2y_2 + 3y_3$.

The set of all positive integral solutions $S(\mathbf{a}, \mathbf{b})$ of $\mathcal{H}(\mathbf{a}, \mathbf{b})$ is $\{\mathbf{y} = (y_1, \ldots, y_n) | (a_1y_1 + \cdots + a_my_m = b_1y_1 + \cdots + b_ny_n = 0, y_i \in \mathbb{N}\}$. The basis, $M(\mathbf{a}, \mathbf{b})$, of $S(\mathbf{a}, \mathbf{b})$ is the set of all its minimal elements.

Let $S \subseteq \mathbb{N}^n$ and $x, y \in S$. Define $x = (x_1, \ldots, x_n) \leq y = (y_1, \ldots, y_n)$ iff $x_i \leq y_i$, $(1 \leq i \leq n)$.

The set $M$ of minimal elements of $S$ with respect to the ordering $\leq$ is such that for any $s \in S$ there is an $m \in M$ and $m \leq s$ and for any $m \in M$ if there is an $s \in S$ and $s \leq m$, then $m = s$.

That is, $x$ is minimal if there is no $y$ such that $y \leq x$, otherwise, $x = y$.

Let $B = \{b_1, \ldots, b_k\}$ be a basis of $S(\mathbf{a}, \mathbf{b})$, then $S(\mathbf{a}, \mathbf{b}) = \{x | x = b_1b_1 + \cdots + b_kb_k, b_i \in \mathbb{N}\}$.

And the set of integral solutions is generated by the finite set of minimal elements of $S(\mathbf{a}, \mathbf{b})$. By Dickson’s lemma, this set is finite.

This short ex cursus on polynomials is important because in many areas of symbolic computation, computing is done by finding solutions of systems of polynomial equations. It is the case in Constraint Logic Programming and some of its dialects like “Contrainte avec Logique” [Sa89] where Buchberger’s algorithm is used to compute the so-called Gröbner bases. The algorithm transforms any given basis of a polynomial ideal into an equivalent basis in normal form, its Gröbner basis. Let $I(F)$ be an ideal in $\mathbb{Q}[X]$ generated by a basis $F = \{f_1, \ldots, f_n\}$. Then $F \subseteq \mathbb{Q}[X]$ is a Gröbner basis if, for all $p \in \mathbb{Q}[X]$, $q_1, q_2$ are the normal forms of $p$ mod $F$, then $q_1 = q_2$. For our purpose, it is interesting to note that, by Dickson’s lemma, the algorithm terminates.
2.4 Constructive decision procedure.

The detour through Seidenberg’s proof has been long but worthwhile. Indeed, the constructive proof of Hilbert’s theorem provides a constructive solution to the word problem for commutative rings as well as for commutative semigroups and monoids.

A classical technique due to McKinsey [Mc43] used to solve the decision problem for some classes of sentences can also be used to solve the word problem for commutative rings [Si70]. The technique consists in reducing the problem and in showing that some systems of linear equations over polynomial domains are solvable. Let $L$ be the first-order language associated to a ring, then the set of universal sentences (i.e. sentences in prenex normal form without existential quantifiers) which hold in some decidable subsets of the class of rings is recursive.

Following McKinsey, a conditional sentence is of the form $(\forall x_1, \ldots, x_n)[f_1 = 0 \land \cdots \land f_r = 0 \rightarrow f = 0]$, where the $f_i$ are terms in the variables $x_i$, and each terms of the sentence is seen as a polynomial in variables $x_i$. In order to show that some set of $L$ is recursive, it suffices to first show it equivalent to some conditional sentence, then to translate the statement that some sentence is in that set into a statement concerning the membership of polynomial ideals in the ring of integers, a problem which has a constructive solution [He26], [Se74].

Simmons [Si70] shows that, for the ideal $I = (f_1, \ldots, f_n)$ in $\mathbb{Z}[X]$ generated by polynomials $f_i$, there is an effective procedure to decide the membership of some arbitrary polynomial $f$ in $I$. That is the membership problem (PI) for polynomial ideals is effectively solvable. By Seidenberg’s construction, we now know that it is constructively solvable.

In [Si80] these results are extended to give a solution to the word problem (WP) for Thue systems. And this solution translates almost immediately into a solution of the same problem for commutative semigroups and monoids. As we will see, [Me82] give a complete proof relating the solution of WP in commutative semigroups to the membership problem in polynomial ideals, that is, WP is reducible to PI.

Simmons’s procedure works in both directions, from WP we can pass to PI and inversely, that is, we can now pass from Hilbert’s theorem to Dickson’s lemma.

Let $\Sigma$ be a finite alphabet and $w_1, w_2, \ldots$ words in $\Sigma^*$ as before. A semi-Thue system $S = (\Sigma, R)$ consists in a set of production rules (R), $w_1 \rightarrow w_2$ s.t. if $w_1, w_2 \in \Sigma^*$, then $w_1 \rightarrow w_2(R)$ if there are words $u, v, r, s \in \Sigma^*$ s.t. $u = rw_1s$ and $v = rw_2s$. That is, $v$ is obtained from $u$ by replacement of $w_1$ by $w_2$, called a derivation $w_1 \rightarrow w_2$. means that there is a sequence $w_1 \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow w_n \rightarrow w_2$. For example, if $ab \rightarrow aa, ba \rightarrow bb \in P$, then there is a derivation $aba \rightarrow abb \rightarrow aab \rightarrow aaa$, i.e. $aba \rightarrow aab$.

A Thue system $T$, also called a semigroup presentation, is a symmetric semi-Thue
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

system. That is, if \( w_1 \rightarrow w_2(R) \), then \( w_2 \rightarrow w_1(R) \), that is \( w_1 \rightarrow w_2(R) \). \( T \) generates a congruence, that is, the reflexive transitive closure of \( \rightarrow \) when, for \( xuy \rightarrow xvy, u \equiv v(R) \), (i.e. mod \( R \)) if \( u \xrightarrow{\sim} v \). The congruence class of \( v \) is \([v] = \{ u \in \Sigma^* | u \xrightarrow{\sim} v \} \). And the congruence classes of \( T \) form a monoid under multiplication and identity, i.e. the monoid presented by \( T \). Even though we are not interested in it here, a usual way to show termination of a derivation is by the Church-Rosser property. \( T \) is Church-Rosser if for all \( x, y, z \), \( x \rightarrow y \) implies that there is a \( z \) such that \( x \xrightarrow{\sim} z \) and \( y \rightarrow z \).

Let \( T = (\Sigma, R) \) be a Thue system with relations \( U_r \equiv V_r(R) \), where the \( U_i, V_i \) are words on the alphabet \( \Sigma_1, \ldots, \Sigma_n \in \Sigma \).

The variables \( x_i \) can be seen as the indeterminates in a polynomial ring \( Q[X] \) (or \( Z[X] \)), that is, each word \( w \in \Sigma^* \) is a monomial in \( Q[X] \).

We can then consider polynomials \( f_i = U_i - V_i \) of \( Q[X] \) and the ideal \( I = (f_1, \ldots, f_n) \) in \( Q[X] \), then \( U \equiv V \) in \( T \) is equivalent to \( f \in I \) in \( Q[X] \) (or \( Z[X] \)) [Si80].

The proof is easy: show that there are polynomials \( g_1, \ldots, g_r \in Q[X] \) such that \( f = g_1 f_1 + \cdots + g_r f_r \) holds. By definition of \( T \), there is a derivation \( U \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow V(R) \). Hence, given the previous reducibility of \( WP \) to \( PI \), by Seidenberg construction, there is a constructive procedure to solve the membership problem for polynomial rings. By Simmons lemma, there is a constructive procedure to solve the \( WP \) for commutative semigroups and monoids.

We can now come back to Dickson’s lemma. A consequence of the lemma is that the finitely generated commutative semigroups are finitely presented. [Re65] [Cl67].

Consider a semigroup presentation, that is, a Thue system with \( U_i \equiv V_i \), and the ideal \( I \) generated by \( (U_i \equiv V_i) \), where \( i \in I \). Then, by Simmons lemma, \( U \equiv V \) in \( T \) if \( U \equiv V \). If \( I \) is finitely generated, there exists some \( J \subseteq I \) s.t. \( U_i \equiv V_i \), where \( i \in J \) generates \( I \). Hence the presentation is finite.

Starting with Hilbert’s theorem, consider an algebra \( RM \) constructed from a commutative ring \( R \) together with a multiplicative monoid \( M \) and let \( Q_i \) be a congruence in \( M \), i.e. the equivalence relations seen as submonoids when \( Q_i \subseteq M \times M \). Let \( I \) be an ideal in \( RM \) generated by \( (U - V) \) in \( Q \). Then [Ei69], \( Q_i = \{(U, V)|U, V \in M, U - V \in I(Q_i)\} \) if \( Q[X] \) and \( RM \) are Noetherian. Hence, the ideals as well as the congruences, satisfy the ACC.

Consequently, any congruence in a finitely generated commutative semigroup is finitely presented [Re65], and this is equivalent to every finitely generated commutative semigroup is finitely presented (theorem 72). And by Clifford’s proof [Cl67], who derives the former result from it, we fall back on Dickson’s lemma in its group-theoretic formulation: the set of all minimal elements of a subset \( A \subseteq F \), \( F \) a free semi-group, is finite. It should be noted that Higman’s theorem obviously still holds in the vocabulary of commutative semigroups and Noetherian rings.
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

The argument will be completed in the next section, but for the moment, we stop at this point, relying on Richman criterion [Ri74] and satisfied that our algebraic structures, the commutative monoids and semigroups, can be shown constructively to possess the property of being Noetherian. This may be a weak notion of constructivity, and, moreover we rely on Seidenberg construction for rings and the translation from rings into semigroups without actually exhibiting the same construction for monoids or semigroups. But, from what we have seen, it can be done. And, after all, even though constructivity may have some appeal in the framework of this research, the main question is to know what is feasible, that is what is computable and practically computable, and in general, computation is performed on polynomial equations.

2.5 Complexity.

2.5.1 The complexity of the word and membership problems.

Herman had shown that if \( p \in (p_1, \ldots, p_n) \in \mathbb{Q}[X_1, \ldots, X_n] \), there exists \( g_1, \ldots, g_n \in \mathbb{Q}[X_1, \ldots, X_n] \) s.t. \( p = \sum_{i=1}^m p_i g_i \), and the degree \( \text{deg}(g_i) \leq \text{deg}(p) + (md)^2n \) where \( d = \max(\text{deg}(p_i)) \), \( i \in I = (p_1, \ldots, p_m) \). She obtained some bounds on the complexity of solving system of linear diophantine equations, i.e. polynomials in \( K[X] \) where \( K \) is some computable field. To give a single example, Herman obtained a complexity degree \( D(n, d) = (d)^{2n-1} \) where \( d \) is the maximal degree of the polynomial in \( K[X_1, \ldots, X_n] \).

Algorithmic improvements later gave a value \( (d)^{\sqrt{3n-1}} \) [La76], and more recently, several other finer approximations have been obtained, particularly in the computation of standard bases or Gröbner bases. For example, the latest best upper bound for \( PI \) is \( D(n, d) \geq d^{2m} \), where \( m \sim n/2 \), [Ya91]. And the best upper bound for the Gröbner bases is \( G(n, d) \leq d^{2m} \); the best lower bound in the case of an ordering on \( d \), \( G(n, d) \geq G_<(n, d) \) is to compare with the earlier doubly exponential bounds for the normal forming algorithm which also holds for Church-Rosser commutative Thue-systems [Hy86a].

From Hermann results, and using the equivalence \( WP = PI \), given a semigroup presentation and an instance \((a, b, P)\), where \( P \) is the presentation, of the \( WP \), Meyer and Mayr [Me82] show that \( a \equiv b(P) \) iff there is a derivation \( a = c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_n = b(P) \) of \( b \) from \( a \) s.t. \( \text{length}(c_i) \leq 2^{2^{\text{size}(a,b,P)}} \), \( 0 \leq i \leq n \). Hence, there is a Turing machine \( T \) and some constant \( d > 0 \), s.t. for any instance of \( WP \) (or \( PI \)), \( T \) decides whether \( a \equiv b(P) \) in space at most \( 2^{2^{\text{size}(a,b,P)}} \).

Then, reducing the semigroup presentation to a succinct presentation, they show that there is a set \( ESC = C \) where \( C \) is a terminating 3-counter machine whose computation is bounded by \( 2^{2^{\text{size}(C)}} \), which is EXPSPACE-complete and log-lin reducible to \( WP \).
Hence the WP is EXPSPACE-complete with respect to log-lin reducibility.

2.5.2 Complexity of the Logic.

We have seen in section 1.3 that the first degree fragment of most Relevant Logics has a characteristic four-valued model. Since the decision problem of classical logic reduces to the decision problem of this fragment, the later has the complexity of Classical Logic, i.e. NP-completeness [Ur90].

Nothing was known about the complexity of the relevant logics until Urquhart [Ur90] showed that the decision problem of the conjunction-implication fragment of the logics between $T$ and $R$ is EXPSPACE-hard under log-lin reducibility.

Technically, Urquhart’s argument follows that of [Me82] with appropriate modifications to take account of the logic. Since some of the earlier arguments rely on it, we summarize Urquhart’s translation of the logic into a semi-Thue system.

Starting with $R_{\wedge \wedge}$, the implication-conjunction-fusion fragment of $R$ with propositional constant $t$, a model structure is a triple $M = (0, K, R)$, where $K$ is a set of possible worlds, $0 \in K$ and $R$ is a ternary accessibility relation satisfying the usual postulates of the Routley-Meyer semantics [Ro73]. A valuation in $M$ is a function $V$ assigning a value $V(P) \subseteq K$ to each propositional variables $P$, s.t. if $a \in V(P)$ and $R0ab$, then $b \in V(P)$. A formula is valid if $0 \models A$, i.e. $A$ is true at world 0 if it is valid in all models $M = (M, V)$.

A semi-Thue system (defined as before) is commutative-contractive if for $x, y \in \Sigma$, it contains all productions of form $xy \rightarrow yx$ and $xx \rightarrow x$ respectively. From a commutative-contractive semi-Thue system $S = (\Sigma, R)$, Urquhart constructs a model for the logic $R_{\wedge \wedge}$ as follows: define a relational structure $M(S) = (0, \Sigma^*, R)$ where $0 \in \Sigma^*$ is the empty string, and for $a, b, c \in \Sigma^*$, $Rabc$ iff $ab \preceq c$ for worlds $a, b, c$. Then, $M(S)$ is a model structure. Each variable of the logic is correlated to elements of $\Sigma$: for $a \in \Sigma$, $P(a)$ is the corresponding propositional variable, and for $s \in \Sigma^*$, $P(s)$ is the corresponding propositional expression. Concatenation in $S$ corresponds to “$\circ$”, fusion, and the empty string is the propositional constant $t$.

If $a \in \Sigma, b, c, d \in \Sigma^*$ in $S$, the canonical model $M(S)$ associated with $S$ is the model defined on $M(S)$ by $V(P(a)) = \{s \in \Sigma^*|a \rightarrow s(S)\}$, and $V(P) = \emptyset$ for uncorrelated variables. Then, $c \models P(b)$ in $M(S)$ iff $b \rightarrow c$.

Let $a, b \in \Sigma^*$ in $S$, and define a formula $F(S, a, b) = \left[ P(c_1) \rightarrow \left[ \left( P(d_1) \wedge \cdots \wedge P(c_n) \rightarrow P(d_n) \right) \wedge t \right] \rightarrow P(b) \rightarrow P(a) \right]$, where $c_1 \rightarrow d_1, \ldots, c_n \rightarrow d_n$ are the non-contractive non-commutative productions in $S$. Then, $R_{\wedge \wedge} \models F(S, a, b)$ iff $a \rightarrow b(S)$.

Since $R_{\wedge \wedge}$, as well as $LR$, are translatable into $R_{\wedge}$, the complexity result holds for $LR$. Moreover, the result extends to $R_{\wedge}$, and that is not surprising given its commutative
monoid semantics.

A corollary of the \textit{EXPSPACE}-completeness proof of the word problem \( WP \) is that the \( PI \) membership problem is \textit{EXPSPACE}-hard with an upper bound is probably \textit{doubly} exponential [Me82]. We have seen that this bound has been recently lowered. Nevertheless, at the present stage of complexity theory, and from a theoretical point of view, we may assume that taking our decision problem from the side of Hilbert's theorem and computing the standard bases or Gröbner basis is still harder than from the semigroup side.

We can now come back for a moment to the constructivity of the decision procedure of the logics. As we have seen, the decision procedure basically amounts to reduce a proof to some irredundant normal form which is finite by \( D, K, IDP \). Given the former reduction of the logic to some semi-Thue system, we can now see that the procedure is similar to the solution of the word problem for commutative semigroups, monoids or Thue-systems. Indeed, these systems are expressible into a semi-Thue system which determines a congruence and equivalence classes. And for two classes \([u], [v]\), the system terminates if there is a derivation ending with \([u] \equiv [v]\). By Dickson's or its corresponding equivalent formulations, the system terminates, otherwise there would be an infinite derivation without termination. And this is exactly what the decision procedure amounts to. Hence, by Urquhart's reduction of the logics, and by Simmons's translation, there is a constructive decision procedure for the logics.

2.5.3 Lowering the complexity.

In practical cases, we do not consider infinitely long inputs, and complexity results are worst cases situations. So it is interesting, and obviously essential when working on real applications, to ask at which level the complexity strikes. In our empirical investigations reported in the next chapters, the formulae of the logics were always restricted to formulae in no more than five variables. The main reason is that in the case of a formula with many different variables, if it is not a theorem, its variables multiplicity makes it an easier candidate for refutation by filters and matrices. But iff it is a theorem, it may be extremely difficult to prove it by mechanical means. Some examples are given in Chapter 6.

Some partial results on the complexity of the word problem for commutative semigroups are known, and, relying on Urquhart's procedure, we will draw on them to derive some information on the complexity of the logics.

Using an ingenious construction which holds only in the case of polynomial rings in
four generators, Huynh [Hy86] reduces the satisfiability problem (SAT) to the membership problem, and shows that \( PI \) is \( NP \)-hard, possibly the best available result. (A language \( L \) is \( X \)-hard \( (X = NP, EXP, \ldots) \) iff, for every language \( L' \), \( L' \) is polynomially reducible to \( L \). It is \( X \)-complete iff it is \( X \)-hard and belongs to \( X \)).

By equivalence of \( WP \) and \( PI \) the result holds for \( WP(4) \), the word problem of commutative semigroups in four generators.

In addition, on the basis of the former results on the complexity of \( WP \), [Hy85] shows that \( WP(1) \) is \textit{polynomial}, as one would expect, and more interestingly, that \( WP(X) \) where \( X \geq 6 \) is \textit{SSPACE-complete}, symmetric linear space complete, i.e. between \( DSPACE \) and \( NSPACE \). That means that with a \textit{bounded} number of generators, the \( WP \) for commutative semigroups is in \( PSPACE \).

\( PSPACE \) is the class of languages recognizable in polynomial space. \( L \in PSPACE \) iff there is a Turing Machine \( T \) and a polynomial \( p(n) = n^k, k > 0 \), s.t. for all input \( a \in L \), \( T \) computes \( a \) in space bounded by \( p(length(a)) \). To locate it in the complexity hierarchy, \( P \subset NP \subset PSPACE \).

In order to show that the lower result applies to the cases of two and three generators semigroups, we rely on an embedding theorem of Evans [Ev52] according to which \textit{any countable semigroup can be embedded in a two-generators semigroup}. It is known that the word problem for semigroups is already unsolvable with two generators [Ha49]. But for commutative semigroups, it is generally solvable. The problem is to insure that the embedding of a commutative semigroup into a two generators commutative semigroup preserve solvability.

Let \( S \) be a countable semigroup generated by \( g_i \) generators \( g_1, g_2, \ldots \), and defined by relations \( r'_i(g_1, g_2, \ldots) = r'_i(g_1, g_2, \ldots) \). As before, we can see the semigroup as a semi-Thue system with production rules \( r_i \rightarrow r'_i, r'_i \rightarrow r_i \), and with with derivability defining an equivalence relation on \( S \).

Following [Ev52], let \( F \) be a free semigroup with two generators \( a, b \), and let \( FS \) be the subsemigroup generated by \( \{bab, ba^2b, \ldots \} \) where the \( ba^ib \) are in a one-to-one correspondence with the \( g_i \). Then, for \( w_1, w_2 \in FS \) and \( w_1 = uw_2v \) where \( u, v \) are words in \( F \), \( u, v \in FS \) (*).

Imposing on \( F \) the corresponding relations, \( r_i(bab, ba^2b, \ldots) = r'_i(bab, ba^2b, \ldots) \) define an equivalence relation on words of \( F \), hence a semigroup \( G \) (since the relations define the actual elements generated).

By definition of the relations on \( F \), if \( w \in FS \), then any word equivalent to \( w \) is also in \( FS \). Let \( w' \) be a word in \( FS \) corresponding to a word \( w \) in \( S \) under the mapping \( g_i \rightarrow ba^ib \), then [Ev52] shows that, if \( u, v \) are words in \( S \), by induction on transformations from \( u \) to
v, \ u' = v' \ \text{in} \ G \text{ iff } u = v \ \text{in} \ S. \text{ And the embedding theorem is proved by showing that there is an isomorphism between the equivalence classes in } S \text{ and } G \text{ consisting of words in } FS.

To answer our question, and it is actually the main condition to impose in the proof, it is essential to know which semigroup can play the role of FS. Assume that S is commutative, i.e. in the commutative semi-Thue system, we have, for all \ g \in S, \text{ the relation } g_1 g_2 = g_2 g_1. \text{ Then, the necessary and sufficient condition to impose on } FS \text{ is that it be freely generated and satisfy condition (*)}. \text{ And commutativity does not oppose to these conditions.}

To conclude, we still need a further refinement provided by Neumann [Ne60]: every finitely generated semigroup can be embedded in a two generators semigroups and every countable semigroup can be embedded in a three generators semigroups. For our purpose, the embedding in a three generators semigroup does not add much of course, as long as embeddability in a two generators is granted.

It is interesting to note the complexity of the embedding in the generalization of Neumann’s result: if S has d generators and is defined by relation \ r(x_1, \ldots, x_n) = r'(x_1, \ldots, x_n), \text{ then S can be embedded in a two-generators semigroup that satisfies the relations } r(x_1^m, \ldots, x_n^m) = r'(x_1^m, \ldots, x_n^m), \text{ where for any } d, m \geq d + 3(\sqrt{d}) + 3.

Of course, if WP(4) is embeddable into WP(2), the embedding does certainly not reduce the complexity. So, at the moment, NP-hardness of WP(2) is the best approximation, and, in general, for bounded WP(n), in PSPACE.

We can now think that these results apply to the logics. Indeed, [Hy85] shows that WP(n) is in SSPACE(n) as a consequence of the exponential-space complexity of the WP for commutative semigroups. The procedure applied to obtain the complexity of WP(n) consists in reducing linear-space-bounded symmetric Turing machines to exponential-space-bounded symmetric Turing machines, and then to reduce exponential-space-bounded symmetric counter machines to WP(n). To show SSPACE(n)-hardness, linear-space-bounded symmetric counter machines are shown to be simulated by a specific exponential-space-bounded 3-counter machine, and a computation of the later is described as an instance of WP(6).

On the other side, Urquhart’s procedure consists in first constructing models for the logics from semi-Thue systems. Then \ R_{\wedge, \wedge} \text{ is shown exponential-space-hard with respect to log-lin reducibility by log-lin reduction of an exponential-space hard set to it. That set is } ESC \text{ in section 2.5.1. Finally, a commutative-contractive semi-Thue system is constructed from a 3-counter machine and } ESC \text{ is reduced to the WP for these semi-Thue systems, giving the exponential-space lower bound for the logics. Up to this point, both procedures are exactly similar, the first one using an appropriate automaton for the bounded case.
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

reduction. The essential steps of Urquhart’s procedure are, first, the translation of the logic into a semi-Thue system where each propositional symbol corresponds to an element of the generating alphabet, and, secondly, insuring that contraction does not increases the complexity, i.e. that the productions of the system do not exceed the capability of the 3-counter machine. Obviously, given both procedures, the essential steps can reproduced in the first procedure showing that the bounded cases complexity results apply to the logics with a bounded number of propositional variables. This provides an indication of the complexity level which we can already expect in simple cases. In any cases, WP(2) is in PSPACE, i.e. for our logics, possibly the same complexity already in two variables as all of intuitionistic logic.

2.6 Kripke’s Conjecture.

2.6.1 A New Upper Complexity Bound.

Kripke had conjectured that the decision procedure proof, that is, the proof that any proof of any formula in a proof search tree constructed by application of the rules of the proof theory is finite, is not provable in primitive recursive arithmetic (PRA), [TMM88].

In order to show that it is actually so, we first consider the upper complexity bound of the logic. In [Ur90], Urquhart has further shown that the decision procedure for $R_{\land \lor}$, hence for any other logic translatable into it, is primitive recursive in the Ackermann function. This result is based on the study of decision procedures for Petri Nets (PN) and vector addition systems (VAS) where a decision procedure analogous to the Kripke-Dickson-Meyer procedure. We first neeed a few definitions.

The finitely presented commutative semi-Thue systems that we have seen before are equivalent to reversible VAS and PN.

A $k$-dimensional VAS is an ordered set of $n$-tuples of integers, $v_i$ ($0 \leq i \leq k$). A vector $v'$ in VAS is reachable from a vector $v$ if there is a finite sequence $v = v_1, v_2, \ldots, v_n = v'$ from $v$ to $v'$ s.t. $v_i = v_{i-1} + v_j$.

A PN is a finite directed graph with nodes called places and transitions. A marking in a PN is a mapping $m$ of the set of places into $N^+$. The reachability problem is then to decide for a PN and a marking $m$ if there is a finite sequence of transitions leading from $m$ to some other place. With respect to decidability, the reachability problem is equivalent to the WP for commutative semigroups or semi-Thue systems.

Karp and Miller [Ka69] have shown that the finite containment (FCP) and the finite equality problems are decidable. That is, it is decidable for two PN or VAS whether each is finite reachable, and if so, whether the reachability net or vector of the first contains the
reachability net or vector of the second. A next result by Mayr and Meyer [Ma81] showed that the complexity of the decision procedure for each of the two problems exceeds any primitive function infinitely often.

A closer analysis showed [Mc84] that for each \( k \) in a \( k \)-dimensional VAS, or a \( k \)-places PN, the \( k \)-FCP has a primitive recursive decision procedure. And in the unbounded case, McAloon shows that the procedure is *primitive recursive in the Ackermann function*.

It is interesting to note that McAloon’s proof relies on the notion of “relatively large sets” used by Paris-Harrington in their independence result from Peano Arithmetics based on a finite form of Ramsey’s theorem.

McAloon’s result was sharpened by Clote [Cl86]. Using a technique based on a *finite* version of Ramsey’s theorem, he shows that the FCP is \( DTIME(Ackermann) \)-complete with respect to polynomial-time reducibility.

A set \( A \) is polynomial-time reducible to a set \( B \) if there is a function \( f \) computable in polynomial time s.t. for all \( a, a \in A \) iff \( f(a) \in B \). With respect to space, this means that FCP being decidable in time \( T \), \( DTIME(T) \subseteq DSPACE(T/\log T) \).

Of course, this improved bound adapts straightforwardly to the Kripke decision procedure.

This improves the former bound, but we should not forget that a function \( f \) is primitive recursive in a function \( f' \) iff \( f \) is in the class obtained by primitive recursion and composition from \( f' \). And the Ackermann function \( f(a, b) = f[(a - 1), f(a, b - 1)] \) grows very rapidly: for example, \( f(3, 2) = 16, f(3, 4) = 2^{65536} \).

Finally, to see concretely what sort of complexity we are dealing with, and to consider Dickson’s lemma “in practice”, first consider an example from [Gi66] of the very large possible size of \( m + 1 \) pairwise incomparable vectors in \( N^n \), \( n \geq 2 \), defined as follows for \( m > 0 \), \( \{(i, j) \times 0^n - 2|i + j = m\} \).

Next, remember our use of Ramsey’s theorem. A fundamental result of [Na65a] relates the theory of ordered sets to the partition theory in Combinatorics. Nash-Williams proves, on the basis of the *WQO* theory, that if \( I \) is an infinite subset of natural numbers and \( A(I) \) the set of all ascending sequences of elements of \( I \), then, if \( T \subseteq A(I) \) is such that for no two different sequences \( s, t \) such that \( s \) is a left segment of \( t \), if \( T \) is divided into \( m \) disjoint subsets, then there exists an infinite subset \( K \) of \( I \) such that \( T \cap A(K) \) is in a single partition of \( T \). That is, we obtain Ramsey’s theorem. The resemblance of this theorem with Meyer’s *IDP* is striking and calls for more investigation. It is known that the property of a set to satisfy Ramsey’s theorem is equivalent to the Sperner’s property: the power set of a finite set has the Sperner property if for any two subsets, one is not contained in the other [Ne79]. That is, again, the property of being pairwise incomparable. The number of such Sperner systems \( S(n) \) of sets is equivalent, up to two units, to...
CHAPTER 2. CONSTRUCTIVITY AND COMPLEXITY.

the number of elements in a free distributive lattice with \( n \) generators, or to the number of monotonic increasing Boolean functions in \( n \) variables. For example, respectively, for \( n = 2, 5, \) and \( 7, S(n) = 4,7539, \) and 2 followed by 13 digits [Co74].

2.6.2 Reverse Mathematics.

Since the discovery of results of independence from Peano Arithmetics, many questions related to the adequacy of PA have been raised. In Computer Science, some typed languages are so complex that their termination algorithms were proven independent from second-order PA [Le81], making any verification and correctness proof impossible. This sort of situation created a need to investigate weaker systems of arithmetics like the subsystems of second-order arithmetics, \( \mathbb{Z}^2 \). Results obtained in the context of H. Friedmann's program of "Reverse Mathematics" [Hm85],[Dr87] often summarized in the question "Which set existence axioms are needed to prove the theorems of ordinary mathematics?" provide some indications about the relative strength of various mathematical theorems in a hierarchy starting from primitive recursive arithmetics (PRA) up to ZFC (Zermelo-Fraenkel set theory plus the Axiom of Choice) and beyond. And this throws some light on the relative strength of some of the theorems we have used, i.e. how much "mathematical power", hence, what computational resources, are required to prove them. We just collect various known results that are related to the theorems we have used in the first two chapters. In ascending order of strength, the subsystems of \( \mathbb{Z}^2 \) we will refer to are \( \text{PRA}, \text{RCA}_0, \text{ACA}_0, \text{ATR}_0, \Pi^1_1 \text{CA}_0 \).

These systems can be characterized as follows: \( \text{RCA}_0 \) is \( \text{PRA} \) plus the assertion of existence of all recursive sets of natural numbers, adding to it the assertion that any two well-orderings of natural numbers are comparable gives \( \text{ATR}_0 \). \( \text{RCA}_0 \) plus the comprehension scheme \( \exists X \forall x(x \in X \leftrightarrow \phi(x)) \), where \( \phi(x) \) is any arithmetic formula, gives \( \text{ACA}_0 \). If \( \phi(x) \) is any \( \Pi^1_1 \) formula, then we obtain \( \Pi^1_1 \text{CA}_0 \).

We have seen that the upper complexity bound of the logics relies on a finite version of Ramsey's theorem. This finite version of Ramsey's theorem is not provable by combinatorial means alone. The proof requires to use the infinite. In the hierarchy of subsystems, depending on the form of the finitization, the theorem is provable in \( \text{ACA}_0 \) or \( \text{ATR}_0 \) [Dr87].

Friedmann's has shown that the finite version of Kruskal's Tree theorem (corollary 1.35) is not provable by finite means either. This version is not provable in \( \text{ATR}_0 \). The proof requires, in addition, the stronger comprehension scheme \( \Pi^1_1 \). It is then provable in \( \Pi^1_1 \text{CA}_0 \), and for fixed \( c \), it is provable in \( \text{PRA} \) [Dr87].

Hilbert's basis theorem is provable in \( \text{RCA}_0 \) [Si88], and Higman's theorem in \( \Pi^1_1 \text{CA}_0 \).

---

\[ \text{that is, a statement of the form} \ \forall x \exists A, \text{where } x \text{ is a second-order variable ranging over infinite sequences, and } A \text{ is first-order,} \]
where the $\Pi^1_1$ induction scheme corresponds to the recursive argument of the proof [Dr87].

These results suffice to show that, obviously, as conjectured by S. Kripke, the decidability proof of $LR$ and of any other related logical system relying on the same decision procedure, as well as other theories, including Unification theory, Rewriting systems, Gröbner bases, and, as we will see, Constraint logic Programming, in which termination is insured by one of the equivalent theorems that we have studied in these chapters, is not provable in PRA. This is not too surprising, after all, considering the strength of PRA.
Chapter 3

Prolog and Parallel Processing.

In this chapter which makes the transition between the theoretical investigations into the decision problem and its complexity and our empirical investigations based on the implementation of a Prolog theorem prover in Chapter four, we first review some basic aspects of Logic Programming in the classical Prolog paradigm, underlining some of its shortcomings, and some aspects of Constraint Logic Programming which has been proposed as an answer to some of Prolog's problems. Results of the first two chapters are shown to apply to the termination and complexity problems of Prolog II and III algorithm in the algebraic domain of infinite trees.

A usual critique of Prolog as a programming language is its execution speed. The Aurora Or-parallel compiler, still in development, but probably the state of the art in the field at the moment, allows to run Prolog programs on parallel machines and to obtain significant execution speed-ups of these programs. We briefly explain the main features of this compiler. This will allow us to better understand some of the problems encountered in the implementation and experimentation with the sequential and parallel versions of the theorem prover.

Finally, the relationship between Logic Programming and Automated Theorem Proving is examined. Both fields are closely related, and each one may contribute to the improvement of the other. As we will see, we can consider the execution of the proof theory by our theorem prover as an emulation of the execution of Prolog. Moreover, the efficiency of the theorem prover relies on some form of intelligent backtracking close to a similar technique proposed for Logic Programming. Parts of our experimental work suggests an approach to Automated Theorem Proving relying on very large knowledge bases and massive parallelism. In our Prolog theorem prover, the inferential part is executed by the Prolog engine. For efficiency reasons, as much execution requiring heavy usage of computational and memory resources as possible is done by calls to external routines. Prolog
being a theorem prover itself, the same approach is advocated to improve its inefficient and often blind non deterministic all solutions strategy.

3.1 Logic programming and Prolog.

Compared to other programming languages, Prolog, the best known representative of the Logic Programming paradigm, requires a different style of programmation, declarative programmation, in which some set of relations are expressed and then questioned in order to find a solution to a problem. The language is non deterministic, at each step several choices are possible, and backtracking allows to make new choices if some preceding choice was unsuccessful.

It is interesting to briefly recall the origins of Prolog in order to show its close relationship with Automated Theorem Proving. Two basic features of Prolog originate in Colmerauer’s research in formal grammars and parsing techniques to implement efficient compilers on one side, and in his observation that, in order to treat natural language, it is necessary to make inferences from the information contained in a sentence on the other side. To do so, he noted that the techniques used in Automated Theorem Proving could provide a solution. With Roussel and a few others, he developed the Prolog language, “programmation en logique” based on the SL resolution (linear resolution with selection functions) theorem prover of Kowalski [Ko71]. Since then, Logic Programming has become the label of a larger field of research, of course, but in what follows, we will not make any distinction between Logic Programming and Prolog which is only one aspect of it.

The original implementation by Colmerauer and his group was rapidly improved. To note a few specific features, following [Vc86], Boyer and Moore suggested that structure sharing was appropriate for non determinism. Bruynhooge proposed structure copying and terminal call transformation or tail recursion to save memory space. Kowalski and van Emden proposed the restriction to Horn clauses and the semantics of first-order logic. In 1977, Warren wrote the first compiler, and later proposed the model generally accepted for the implementation of Prolog, the Warren Abstract Machine (WAM).

The initial success of Prolog came from its efficiency in performing symbolic computation, its portability and the ease of writing a complete interpreter in another language, or in more recent applications, the opportunity to extend the Prolog engine for specific applications, for example, hypothetical reasoning in N-PROLOG [Ga84], Conditional Logic Programming in CLOGPROG [Bo90], defeasible reasoning in d-Prolog [Nu90], and many more. We may also mention A. Robinson’s Loglisp system, a Prolog written in Lisp which had the peculiarity of exploring all candidate clauses in a close to parallel way. Robinson
believed that a Lisp thus enhanced might well be an effective device for helping to persuade diehard adherents that after all ... something both useful and beautiful has risen from the old resolution theorem-proving ashes” [Ro82, 299]. Nevertheless, Prolog never enjoyed the popularity of Lisp in AI circles. It is its adoption as official language of the Japanese Fifth Generation Computers Systems which gave it some credit and recognition in the CS community.

Still, Prolog has not reached its maturity. It seems very difficult to establish a standardization, there are various schools of thought, and many computer scientists do not take it too seriously. But efficient commercial compilers are available and very large application programs and expert systems are written in Prolog. In addition, to cite two important examples, some interesting mathematical results in Group Theory were obtained using a Prolog implementation [Lu87], and the techniques of Logic Programming are advocated by R. Overbeek to tackle the very important amount of information produced by the Genome Project.

It is interesting to note that the very first Prolog interpreter written in Fortran achieved 200 lips on an IBM 360/67. The fastest sequential Prolog compiler at the moment, claims 850 Klips on a Sun4.

Speed is an important evaluation factor. In the newsgroup “comp.lang.prolog”, it was a topic often discussed until A. Vellino proposed a Prolog program solving some combinatorial problem so efficiently that, he claimed, no other language could beat it. A few days later B. Demoen of the University of Leuven replied that he had a C program doing the same thing much faster. Obviously, knowing how a Prolog compiler is implemented, generally in C, it is not difficult to mimic the Prolog execution in C, bypassing the intermediary step of interpreting the Prolog source code. This settled the discussion for a while. Nevertheless, as we will see, it is an important issue. It is reported that a Prolog theorem prover written by R. Overbeek and compiled in native code was compared to its exact translation into C. The results showed that the Prolog program reached 25% of the speed of C. And this should be considered as a scale of reference when comparing Prolog to C on a classical sequential architecture.

Speed may be one criterion with respect to which Prolog does not perform very well, but there are many other criteria which make it attractive as a programming language. To conclude, we may summarize some of Prolog often advertised characteristics. It is readable (some would say read only), it is at the same time declarative and procedural, simple and easy to use for prototyping and to implement well defined algorithmic problems, it has a built in strategy and it does not distinguish between program and data... This can be illustrated by a simple example.

Suppose that we want to prove theorems in some logic automatically and that the rules
of the proof theory are the following:

Axiom: \( X \vdash X \).

Operational Rules:

\[
\begin{align*}
X, A \vdash B & \quad \rightarrow \text{Right} \\
X \vdash A \rightarrow B & \\
Y \vdash A & \quad \rightarrow \text{Left} \\
X, A \rightarrow B, Y, Z \vdash C & 
\end{align*}
\]

One implementation of this proof theory may be the following:

% declaration of the derivation operator.
?- op(1000, xfx, \( \vdash \)).
% axiom.
prove(X \( \vdash \) X).
% \( \rightarrow \) Right.
prove(X \( \vdash \) [A \( \rightarrow \) B]) :-
    append(X, [A], Premise),
    prove(Premise \( \vdash \) [B]).
% \( \rightarrow \) Left.
prove(Antecedent \( \vdash \) [C]) :-
    append(X, [A\( \rightarrow \)B|Right], Antecedent),
    append(Y, Z, Right),
    prove(Y \( \vdash \) [A]),
    append(X, [B|Z], Premise),
    prove(Premise \( \vdash \) [C]).
% append List1 to List2 to build List3.
append([], List, List).
append([Head|List1], List2, [Head|List3]) :-
    append(List1, List2, List3).

Obviously, this program is just a transcription of the algorithm of the rules with an external procedure concatenating lists. The program terminates when the decomposition of the tree representing the input formula reaches the tip of a branch, the axiom. Of course, this is a crude and not very efficient implementation, but it illustrates clearly the characteristics that we have just mentioned. And as far as logic and automated theorem proving are concerned, [Fi90] provides very interesting and simple Prolog implementations of tableaux theorem provers which illustrate the point.

3.2 The Prolog Model.

A Prolog program describes a finite universe, the Herbrand universe, in which the problem to solve is described rather than expressed in an algorithm to compute the solution. Solving
the problem amounts to asking the relevant question; and the answer is obtained by
refutation based on SLD (i.e. Selection of a literal using Linear strategy restricted to
Definite clauses, i.e. only one positive literal is admitted, while, in general, Horn clauses
may contain negated clauses), and following a depth-first, left to right exploration of an
AND-OR tree, with backtracking on failure (See for example [Sh86]).

As such, Prolog is a theorem prover based on Robinson's unification algorithm which
determines, if it exists, the common instanciation of two terms, the most general unifier.
Consider a Valgebra A = (A, F), as before, where A is the set of elements, F the operators
(F ≠ ∅), and a countable set of variables V such that V ∩ F = ∅, and define T(F, V), the
set of terms T with variables over F ∪ V.
The substitutions in A are the set of all mappings V → T extended to an endomorphism.
Let S be the set of substitutions, σ ∈ S, t, t' ∈ T, and σ t be the application of σ to a term
t. Then the unification problem consists in finding a most general unifier, i.e. σ ∈ S
such that σ t = σ t'.
For example, the two terms f(x, y) and f(g(y, a), h(a)) are unifiable: there is a substitu-
tion σ(x) = g(h(a), a) and a substitution σ(y) = h(a) which unify the two terms in
f(g(h(a), a), h(a)).
A logic program consists in the declaration of some set of procedures and of a goal to
satisfy. Thus one can say that a programming language such as Prolog is both declarative
and procedural. It is declarative in its representation of a problem, and it is procedural in
its expression of the inference system.
A procedure, also called Horn clause, has the form A ← B₁, ..., Bₙ, where n ≥ 0 and A, Bᵢ
are literals of the form R(t₁, ..., tₘ), where m ≥ 1, R is a m-ary relation symbol, tᵢ are
terms, i.e. constants, variables or expressions of the form f(t₁, ..., tₖ), where k ≥ 1, f is
a k-ary function symbol and the tᵢ are terms.
A ← B₁, ..., Bₙ means that A is true if (B₁ and ... and Bₙ) is true (See Figure 1). Such
a conjunction of positive literals t₁, ..., tₙ is called a query.
A Prolog predicate can be composed of several clauses, each being some possible solu-
tion of the goal, starting with the first clause and backtracking on failure until the goal is
satisfied.
The execution of a Prolog program is performed top-down and left to right, selecting the

n. see p. 19.
clauses \((C_i)\) in the declared order and starting with the leftmost literal \((L_i)\) as in the following search algorithm:

1. select the first goal \(- B_1, \ldots, B_n\).
2. select a call \(B_1 = R(t_1, \ldots, t_m) \leftarrow C_1, \ldots, C_m\).
3. apply a procedure \(R(t'_1, \ldots, t'_m) \leftarrow C_1, \ldots, C_m\) matching \(B_1\) with a substitution, i.e. a most general unifier \(\sigma\).
4. if successful, goto 5, else if failure, goto 2.
5. derive a new goal \(- (C_1, \ldots, C_m, B_2, \ldots, B_n)\sigma\), if all goals are derived, goto 6, else, goto 1.
6. End

The search can be best represented in an OR-tree, the search tree (Figure 1), giving all possible proofs of a goal and whose nodes are the goals and the daughters are alternative goals. Each node of the tree corresponding to a subgoal of a goal which has still to be executed is a choice point or a backtrack point as can be seen from Figure 2, where \(S\) indicates success, and \(F\) failure.

The search concludes when the terminal nodes are empty or, when having tried all alternatives through backtracking, the goal is not savable.

A successful goal is represented as an AND-tree, the proof tree, in which only the proofs, i.e. the successful paths, are recorded as in Figure 2.

In Robinson’s algorithm [Ro65], an occur check tests that a variable is not unified with a term in which it occurs. Computationally, this test is expensive and, in most implementations, it is not performed. Another reason to avoid the test is that it happens rarely given the way Prolog generates terms. Nevertheless, this omission allows the generation
CHAPTER 3. PROLOG AND PARALLEL PROCESSING.

of infinite terms (loops) and puts the termination algorithm into question. Without this occur check, the strategy is not sound nor complete. For example, with the following program,

\[
\begin{align*}
p & : - p. \\
p & . \\
\end{align*}
\]

the query “?- p.” should succeed, but it loops, as it does in the next program where it should fail.

\[
\begin{align*}
p & : - p. \\
\end{align*}
\]

A characteristic of the non deterministic all solutions paradigm of logic programming is that, on failure, the program backtracks to the last choice point. But this is performed blindly without consideration of the former choices nor consideration of the heavy cost of having to recompute several times what had already been computed. Several solutions, under the name “intelligent backtracking”, have been proposed to avoid this inefficiency [Br90]. For example, avoiding to redraw the minimal non-unifiable subtrees which would end up in failure through some dynamic analysis of the causes of failure, or keeping a copy of the proof tree, i.e the maximum unifiable subtrees. Mixing both suggestions amounts to determine a maximum subtree on which unification would be unsuccessful. But, in addition to the cost of the analysis, this could require too much resources to be implemented efficiently. It does not seem that any efficient algorithm has been devised to operate a dynamic analysis of the running program. Some optimization algorithms are implemented in most compilers but are far from solving the problem.

We will come back to this characteristic of Prolog in Chapter four. It has been a major source of problems in the implementation of an efficient theorem prover, and we will see what solution has been adopted. In return this solution suggests some way of improving the efficiency of Prolog as we will see already in section 3.4.1.

3.2.1 Prolog III.

As we just said, the usual Prolog implementations are not sound nor complete. There are various ways of insuring soundness and completeness: for example adding occur check and loops detection. We will stop for a moment on Prolog III because it is an important suggestion to improve the classical model of Prolog, it is a paradigm of Constraint Logic Programming, for example the ICOT implementation Contrainte avec Logique [Sa89], and, finally, its theoretical model is, in some ways, an application of what we have seen in
Chapter two.

Colmerauer proposed a radical solution to some of Prolog's problems with Prolog II, and, later, in its extension to Prolog III [Co82]. These extensions to Prolog represent one form of Constraint Logic Programming. Think of standard Prolog as dealing with finite trees, i.e. the parenthetized formulae or terms of some word algebra. The set of all trees constitutes the Herbrand universe. That means that the usual semantics of Prolog is a model over this algebra. Then, the execution of a program consists in finding a tree assignment satisfying a given constraint.

What Colmerauer remarked is that it is faster to check satisfiability of a constraint in an infinite tree than in a finite tree. Moreover the expressive power of these infinite trees is greater. An occur check is no longer required since the domain is the set of all infinite trees. That means that the unification algorithm is completely modified so that one can deal with circular looping structures or infinite structures which may be important to consider in some contexts. For example, infinite trees can be used to represent complex data containing loops or some kinds of graphs and grammars (see, for example, [Gi84]).

Each Prolog variable now represents a finite tree constructed over a set of function symbols (i.e. it is a true algebraic variable) and the unification of a term $f(a, x)$ with a variable $x$ results in an infinite tree. In the case of infinite unification, there still exists a single most general unifier, and an almost linear algorithm to compute it [Hu76].

Two terms $t, t'$ are said to be infinitely unifiable if there is a substitution, possibly infinite, i.e. a unifier of $t$ and $t'$. For example, $t = x$ and $t' = f(x)$ are not unifiable but are infinitely unifiable under the substitution $\sigma(x) = f(f(...))$ because $\sigma(x) = \sigma(f(x)) = \sigma(f(f(f(...))))$.

In this new approach, the domain of infinite trees or terms constitutes the model for logic programming. And unification on infinite trees amounts to solving systems of equations, called constraints, in the domain of infinite trees. When an equation has no solution, the system backtracks.

It should be noted that Colmerauer's ideas about the basic structure varied. He finally restricted the domain of infinite trees to the rational trees. According to a result of [Mh88] these two structures are equivalent.

In what follows, we consider only the rational trees, i.e. trees such that the set of all their subtrees is finite, as the basic data structure. Obviously, any finite tree is rational. Since the number of subtrees of a rational tree is finite, it can easily be represented by a finite graph in which all nodes having isomorphic subtrees are reduced as in Figure 3.
3.2.2 The Infinite Trees Model of Prolog.

As in the word algebras of Chapter two, trees are represented by formulae or strings, i.e. parenthetized terms in which the parentheses determine the hierarchy of the elements in the tree, i.e a function symbol stands at the root of a subtree whose daughters are represented by the function’s arguments.

To give a unified presentation of the infinite model based on [Vc86],[Co90], consider an algebra with terms $t_i$ constructed over $F \cup V$, where $V$ is an infinite set of variables and $F$ the operations or functions. Each node of a tree is labelled with some $n$-ary function symbol, i.e. each node has $n$ daughters. Terms are sequences of concatenated elements from $F \cup V$ of the form $x$ or $f t_1, \ldots, t_n$, where $x \in V$, $f \in F$ and the $t_i$ are basic terms. Constraints are sequences of elements from $F \cup V \cup R$ of the form $r t_1, \ldots, t_n$, where $r \in R$ is an $n$-ary relation. An assignment $\sigma$ to a subset $V' \subseteq V$ is a mapping $V' \rightarrow D$, where some arbitrary set, $D$, is the domain. The assignment can be extended to a mapping $\sigma^*: T_{\sigma} \rightarrow D$ of the terms onto the domain such that $\sigma^*(x) = \sigma(x)$, $\sigma^*(f t_1, \ldots, t_n) = f \sigma^*(t_1), \ldots, \sigma^*(t_n)$.

In order to evaluate a term under $\sigma$, the variables are replaced by their value and the term is evaluated, i.e. if $t \in T_{\sigma}$, the value of $t$ under $\sigma$ is $\sigma^*(t)$, otherwise, it is undefined. An assignment satisfies a constraint $r t_1, \ldots, t_n$ if $\sigma^*(t_i)$ is defined and if $r \sigma^*(t_1), \ldots, \sigma^*(t_n)$.

More practically, define a tree assignment as a set $X = \{x_1 = t_1, \ldots, x_n = t_n, \ldots\}$ where the $x_i \neq x_j$ are variables and the $t_i$ are trees. An equation is a formula $p = q$ or $p \neq q$, where $p$, $q$ are terms. $t(X)$ is a solution of the equations if $p(X) = q(X)$ or $p(X) \neq q(X)$ respectively. In what follows, we will not consider inequations.

Let $t$ be a term, then $t(X)$ is the tree constructed by replacing each occurrence of $x_i$ in $t$ by its corresponding tree with respect to the tree assignment $X$. $X$ is a solution of the system of equations (i.e. an ordered pair of terms) -possibly infinite- $\{p_1 = q_1, p_2 = q_2, \ldots\}$ if $X \subseteq Y$ s.t. for all $i$, $p_i(Y) = q_i(Y)$.

A system is a finite set of equations: $S = \{p_1 = q_1, \ldots, p_n = q_n\}$.
A reduced or solved system, $S$, is a finite set of equations \{ $x_1 = t_1, \ldots, x_n = t_n$ \} where each $t_i$ contains only one function symbol and no variables different from $x_i$. Any reduced system is solvable or decidable, i.e. it has exactly one tree assignment or tree solution. A system is unsolvable if there is an equation without a tree assignment in the system, s.t. $f s_1, \ldots, s_n = g t_1, \ldots, t_n$, where $f \neq g$.

We now consider how to obtain the reduced form of a system as a rewriting system.

### 3.2.3 Reduction algorithm.

The Prolog algorithm now amounts to defining a subset $A$ of the set of rational trees $R$, where $a_i \in A$ are assertions, (i.e. asserted facts), as a set of rules $t_0 \rightarrow t_1, \ldots, t_n$ with assignments $t_0(X) \Rightarrow t_1(X), \ldots, t_n(X), (n \leq 0, t_i \subseteq F \cup V)$.

Each rule $r_0 \Rightarrow r_1, \ldots, r_n$ is interpreted as the context-free rewriting of $r_0$ into $r_1, \ldots, r_n$.

If $n = 0$, $r_0$ is erased. Equivalently, each rule can be seen as a logical implication: $r_1 \in A, \ldots, r_n \in A \rightarrow r_0 \in A$.

Hence, the assertions are trees which can be erased in one or more steps by application of the rewriting rules or the closure of $A$ under implication. If $u, v$ are two sequences of trees, then, $u \equiv_{i+1} v$ iff there is a rule $r_0 \rightarrow r_1, \ldots, r_m$ and a sequence of trees $s_1 \ldots s_n$ s.t. $u = r_0 s_1, \ldots, s_n \Rightarrow r_1 s_1, \ldots, s_n \Rightarrow r_m s_1, \ldots, s_n \Rightarrow v$. If $u = v$ then $u \equiv_0 v$.

That means that $A$ is the set of $r \in R$ such that for some $k$, $r \equiv_k \emptyset$, i.e. such that all literals can be rewritten in the empty set.

A Prolog program can now be seen as a rewriting rule, each term being rewritten into a sequence of terms, and a set of constraints, some relations on terms.

From Chapter two, we already know that decidability and termination could be proved on this basis, but we will follow Colmerauer’s procedure which is close to the algorithm implemented in the Prolog III compiler.

Colmerauer [Co83] gives a simple algorithm to reduce systems of equations. The algorithm reduces a system to normal form, and provides a decision procedure. Five transformations, $T_i$, which preserve the equivalences of systems, i.e. systems having the same tree-solutions, can be applied to a system $S$: (in the following, “$=$” always means equational identity).

1. **Absorption**: if $x \in V$, erase any $x = x$.

2. **Variable elimination**: if $x = y \in S$, $x \neq y$, replace the other occurrences of $x$ in $S$ by $y$.

3. **Variable prefixing**: if $x \in V$, $t \not\in V$, replace $t = x$ by $x = t$. 


CHAPTER 3. PROLOG AND PARALLEL PROCESSING.

T 4. **Conflict:** replace a subsystem \( \{x = t_1, x = t_2\} \) by \( \{x = t_1, t_1 = t_2\} \) if \( x \in V \) and \( |t_1| < |t_2| \) where \( |t_i| \) is the cardinality of variables, constants and parentheses in \( t_i \in F \cup V \).

T 5. **Explosion:** replace \( \{(s_1 \cdot s_2) = (t_1 \cdot t_2)\} \) by \( \{s_1 = t_1, s_2 = t_2\} \) and \( \{f(s_1, \ldots, f(s_n = t_1, \ldots, t_n)\} \) by \( \{s_1 = t_1, \ldots, s_n = t_n\} \), where "\( \cdot \)" is the concatenation operator.

Repeatedly applying the transformations to a system leads in a finite number of steps to a reduced form on which no transformation applies. A reduced form has a tree solution iff each equation is of the form \( x = t \). If no transformation applies and the subsystem of equations is not in reduced form, it is not solvable.

Given the algorithm, one can now ask what its complexity is.

### 3.2.4 Decidability and Termination.

Since there is no infinite sequence of systems \( S_i \) of equations \( S_0, S_1, \ldots, S_i, S_{i+1}, \ldots \) where \( S_{i+1} \) is obtained from \( S_i \) by application of the transformations, the system is decidable and the algorithm terminates.

Consider \( S_0 \), and let \( N \) be the number of equations \( x = x \) or \( x = y \), and \( M \) the number of equations \( x = t \) or \( t = x \). Then, the number of applications of \( T1 \) and \( T2 \) must be less than or equal to \( N \), and the number of applications of \( T3 \) and \( T4 \) must be less than or equal to \( M \).

Following a procedure identical to the one used previously, see for example Meyer's construction of finitary algebras in section 1.8.3, define the corresponding sequences of positive integers \( S_0', S_1', \ldots, S_i', S_{i+1}', \ldots \), where the equations

\[ \{s_1 = t_1, \ldots, s_n = t_n\} = k^{\max(|s_1|, |t_1|)} + \ldots + k^{\max(|s_n|, |t_n|)} \]

where \( k \) is the maximal arity of the function symbols in \( S_0 \), and \( |s_i| \), the cardinality of \( s_i \). By application of the \( T_i \), \( S_i' \geq S_{i+1}' \) for all \( T \), and \( S_i' > S_{i+1}' \) for \( T5 \). If the sequence \( S = \{S_i\} \) were infinite, then \( T5 \) would have applied infinitely many times.

This is impossible because, by the inequalities, "the integers in the sequence \( S_i' \) would become negative"[Co82,237].

What actually guarantees that the algorithm terminates is exactly the same termination property that we have seen before. [Co82] and [Co83] rather informal proof relies on the fact that there does not exist sequences of positive integers infinite on both sides, that is, the DCC, condition \((vi)\) of Theorem 1.13. This is exactly an informal formulation of a form of Dickson's lemma or Kruskal's theorem. The algorithm reduces the sequence of systems to an irredundant sequence of systems pairwise incomparable, and there is no
infinite sequences of systems $S_i$ of equations $S_0, S_1, \ldots, S_i, S_{i+1}, \ldots$ where $S_{i+1}$ is obtained from $S_i$ by application of one of the $T_i$ to $S_i$.

By now, this is almost routine. But we can add a little more to make routine interesting since the trees algebras we have been dealing with in Chapter two play a central role in logic programming languages (as well as in symbolic computation, equational and functional programming languages, program verification...).

In the Horn clauses model of logic programming, by Herbrand's theorem, the usual semantics of logic programs is a model over the algebra of finite trees. Since it is a topic in which we have been mostly interested in this work, we may recall the main lines of the theorem which is usually interpreted as a move from an infinite domain to a finite domain [Go90]. Indeed, the theorem allows to reduce an arbitrary quantified formula to a set of quantifier free formulae provable in truth values. Quantification is replaced by an infinite sequence of finite disjunctions and conjunctions, and it is sufficient to consider a finite subsequence of the infinite sequence: an infinite set of instanciated clauses is inconsistent iff there is some finite set of instanciations which is inconsistent. Not surprisingly, the theorem can be proved using König's lemma, and, in addition, Herbrand's proof procedure provides an algorithm which is similar to Colmerauer's algorithm.

A similar proof procedure is used in [Mh88] to give a complete axiomatization for algebras over finite, regular and infinite trees. Completeness of the equational theories corresponding to these algebras is shown by the method of elimination of quantifiers and by transformations of any given formula into a Boolean combination of basic formulae, i.e. an existentially quantified conjunction of equations in solved form (see 3.2.2). Translation into disjunctive normal form reduces the basic formulae to atoms evaluated in truth values. Decidability of the theories of these algebras follows from completeness. And, as corollary, the regular and infinite trees are elementarily equivalent. Hence Colmerauer's choice of one structure or the other appears to be not essential.

There is another way to show decidability in these trees algebras which is obviously based on our theorems of Chapter one as we have just seen. The procedure is essentially similar to the decision procedure based on Kripke's lemma -or any of its equivalent formulations. The algorithm reduces the systems to some irredundant normal form, and by Kripke's lemma, the sequence being irredundant, it is finite.

Since the procedures are equivalent, we obtain in addition a result for the complexity of the decision procedure of Prolog III's algorithm which is equivalent to the upper bound found for the decision procedure based on Kripke's lemma in Chapter two, section 2.6. Moreover, we also have an indication of the complexity of solving the systems of equations or of finding the least subset of solvable equations in a system (Chapter two, section 2.5.1).
Depending on the method used, it is exponential or doubly exponential. Finally, the detour through infinite trees and solutions to systems of equations may make Prolog III look quite different from what one may consider as classical Logic Programming, but using the translation procedure of Chapter two, the *equational* theory of Prolog III is translatable into its corresponding *logical* theory.

Of course, in real practice it is a very restricted form of the algorithm which is implemented in the Prolog III compiler. It is not uninteresting to note that Colmerauer admits that there is no termination proof for the algorithm implemented in the production version of Prolog II, but “it has always worked without any problem”. This only means that whatever the underlying theoretical model of Prolog II and III is, there is no guarantee that a real program will terminate. As Bruynooghe put it during a lecture, “if your Prolog theorem prover has not found a proof after five minutes, it will not find one”! That is, it could be in a loop or trying to solve a problem too complex.

We should also add that the complexity referred to above is different from the complexity of a specific Prolog application program. This is a related but somewhat different problem. Several complexity measures can be used, like (time) length or depth complexity of a proof of some goal, or goal-size complexity. See for example [Sh86][Sh84]. Termination proof of such programs is much more intricate [Ba88]. To conclude, in the worst cases, as we just said, we may expect an exponential level of complexity.

Then, one may wonder if the “occur check”, however expensive, should not be maintained. It should also be noted that from a computational point of view, the reduction to Horn clauses, rather than using full Predicate Logic in the original Prolog, improved the complexity level in a significant way. Indeed, the satisfiability problem for Classical Logic formulae in conjunctive normal form is NP-complete. The satisfiability problem for Horn clauses is in P [Jo77]. But, in general, a user expects a fast answer from a machine, so the test is omitted. And, in general, the complexity results show that we may actually be dealing with very simple problems to which Prolog can give an answer in a reasonable amount of time.

### 3.3 Parallel Logic Programming.

Prolog is very elegant as a programming language, but, as we have seen, it may not be very efficient. Using the resources of parallel processing may solve its problem of speed in a significant way. Indeed, Logic Programming languages such as Prolog are inherently parallel.

Three main categories of parallelism can be distinguished: *Or*, *And* and *Stream* parallelism.
One can also distinguish between *implicit* parallelism in which a compiler extracts parallelism from a static analysis of the program, and *explicit* parallelism in which primitives, which explicitly create parallel execution, communication and synchronization, are added to the source code.

*Or*-parallelism consists in trying simultaneously all clauses of some predicate to solve a goal. This means that the *Or*-branches of the search tree are explored in parallel. This approach is multisequential, as many branches as available processors are explored at once, and each branch is processed sequentially. This form of parallelism follows pure Prolog non-determinism, all solutions, paradigm and offers large grain size computation. Depending on the granularity, many processors or processes would not necessarily give constant speedups though. Communication between processes is kept to a minimum and is only necessary on backtracking. There are several *Or*-parallelism models, for example, the early ANL-WAM and the SRI model. Examples of *Or*-parallel systems are the Gigalips Project's *Aurora* and *PEPsys* produced by the European Computer-Industry Research Center, the later mixing *Or* and *And*-parallelism.

*And*-parallelism consists in computing in parallel the literals inside a same clause, in order to find an instantiation of variables satisfying all goals. This means that the computation operates on finer grain parallelism and that a main problem for this approach, more so than in the *Or*-parallel approach, will be the conflicting instantiations of variables which may increase the need for communication.

Practically, *Or* and *And* parallelism can be seen as follows in a program:

\[
\begin{align*}
\text{OR} & \quad \text{prove}(X,Y) :\text{-do}_\text{first}(X,Z), \text{then}_\text{do}(Z,Y). \\
& \text{prove}(X,Y) :\text{-do}_\text{second}(X,Z), \text{then}_\text{do}(Z,Y). \\
& \text{...} \\
& \text{prove}(X,Y) :\text{-else}_\text{try}(X,Z), \text{then}_\text{do}(Z,Y).
\end{align*}
\]

And

\[
\text{prove}(X,Y) :\text{-do}_\text{first}(X,Z), \text{then}_\text{do}(Z,Y).
\]

*Stream* parallelism refers mainly to "guarded", "annotated" or committed-choice languages in which some specification or condition of execution (the guard), for example, which instanciations are required, has to be assigned to each clause of the program. This sort of parallelism requires the intervention of the programmer. It is based on *And*-parallelism and requires explicit communication between processes in order to check the consistency with the specifications. Examples of such languages are *Parlog* from Imperial College, *Concurrent Prolog* from the Weizman Institute, the ICOT *GHC*, and *Strand*.

In *Constraint Logic Programming* and in *Strand*, rules appear respectively like
prove\_first(X, Y) :- prove\_first(X, Z), then\_prove(Z, Y).

\{Constraints\}

prove(X, Y) :- \{Guards\} prove\_first(X, Z), then\_prove(Z, Y).

where \(|\) is the \textit{commit} operator.

The \textit{Or} and \textit{And}-parallelism approaches can be complementary and be combined as in the case of PEPsys. [Ca88] propose an experimental combination of the two forms of parallel processing in the framework of \textit{Aurora}. One example could be the following:

prove(X, Y) :- prove\_first(X, Z), then\_prove(Z, Y).

prove\_first(X, Z) :- \ldots find\_a\_solution\ldots, Z = 1, then\_prove(1, X).

prove\_first(X, Z) :- \ldots find\_a\_solution\ldots, Z = 2, then\_prove(2, X).

prove\_first(X, Z) :- \ldots find\_a\_solution\ldots, Z = N, then\_prove(N, X).

\textit{Aurora} is a prototype parallel Prolog compiler developped by the Gigalips Project whose principal partners are the Swedish Institute of Computer Science, the University of Glasgow (formerly the University of Manchester) and the Argonne National Laboratory [Lu88].

In \textit{Aurora}, parallelism is the parallelism of the program, that means that no specific selection is made at the start of the program since the \textit{Or}-model follows the pure Prolog model. The compiler distributes the tasks in accordance with the available resources and creates new processes when possible. Since each process executes sequentially, efficiency is attained through the use of an efficient sequential Prolog compiler (SICStus Prolog), by running the compiled code on as many processors as possible, and by minimizing the communication overheads.

This prototype implements the SRI model, an extension of the Warren Abstract Machine, for \textit{Or}-parallel Prolog execution on a shared memory machine. In this model, workers explore the search tree in parallel and perform some tasks i.e. the execution of parts of the program which amount to resolution and backtracking. When a worker has completed his task, he moves to another part of the tree to perform another task. This task switching...
CHAPTER 3. PROLOG AND PARALLEL PROCESSING.

3.3.2 The Warren Abstract Machine.

Many Prolog implementations rely on the WAM which is now considered as a standard abstract architecture for Prolog execution [Ne90][Ai90]. It essentially consists in an instruction set and data sets on which the instructions operate. An abstract instruction interprets a program code instruction and translates it into some machine operations.

In order to make clear the short overview of the mechanisms of the parallel implementation, we need only to consider the WAM data areas. The code area contains the program; the control area contains the abstract machine registers, essentially, pointers to other areas, and three stacks. The local stack contains the information necessary to allow backtracking (choice points, B) and recursive execution of the procedures (the environments, i.e. goals (G) to execute, and procedure (P) to reactivate on backtracking). The global stack, also called heap, contains the structures, the terms, created during execution and represented as value cells (addresses). The trail contains the conditional bindings of variables, i.e. those made after the latest choice point, which have to be deallocated on backtracking (See figure 4 from [Ap88]).

In a sequential implementation, on backtracking, it suffices to follow the pointers backwards and to reinitialize the variables cells concerned. In a parallel implementation, a main efficiency problem concerns the multiple bindings and deallocations of the same variable.
CHAPTER 3. PROLOG AND PARALLEL PROCESSING.

3.3.3 The SRI Model.

In this model, the parallel execution of a program is represented as a set of workers (processes or virtual processors) exploring the search tree. The nodes of the search tree correspond to the WAM choice points. In addition to the usual Prolog data (see Figure 4), a worker has a private binding array which records its conditional binding of variables, i.e. the bindings made after a choice point and which may have to be shared). When a conditional binding happens, the address and the binding are recorded on the data array of the process making the binding as well as on a trail recording these shareable bindings in their chronological order. Unconditional bindings, i.e. the variables bound before any choice point which keep their value, rely on the usual WAM by updating the variable value cell.

Since each worker can move up and down the tree, deallocating and reallocating its conditional bindings recorded in its binding tables, on reallocation, a copy of the binding array and of the stack of the branch of the tree on which an alternative is found is made. The main goal of the scheduler is to minimize the bindings to allocate and deallocate, and to create work in the most efficient way. To do so, when creating work, the bindings are installed in the array of the worker creating a task, and this is done as closely as possible of the preceding task.

The gestion of bindings uses the memory management of the shared memory multiprocessors architectures, i.e. the WAM stacks are generalized to “cactus stacks” (Figure 5), mirroring the search space, each branch acting as an accumulation of stacks, all sharing earlier parts of the branch. And each worker operates in some allocated segment of the virtual memory.

The search tree is divided into an upper public part, represented by circles in the picture, accessible to all workers and a lower private part, represented by boxes, only accessible to the worker which creates it. This allows to keep the granularity as large as possible. In this way, the work is controled by the scheduler in the public part where tasks are allocated, and it is controlled by the Prolog engine in the private part. See Figure 5.

Aurora has been implemented on Sequent Balance and Symmetry, machines with up to 40 processors and with bus-oriented architecture, i.e. a system with one or more buses (data, addresses, control, ...) to which all the system components are interconnected. It has also been implemented on a BBN Butterfly, a switch-based machine with potentially up to 256 processors whose main switch structure is the Butterfly Switch Network. See [Ta90] for technical details.

The performances of Aurora using the Argonne scheduler have been tested on a BBN, and the results [Mu89] show that it is feasible to use the Or-parallel approach on such an
architecture, and, with large search trees, in general on multiprocessors with up to 100 processors.

3.3.4 Scheduling.

Scheduling, the creation and organization of work is another fundamental issue. Two schedulers exist for the Aurora system, the Manchester and the Argonne schedulers. Obviously, the aim is to maximize the work performed by each worker, i.e. preferably look for large grain size computation, while minimizing the overheads of bindings and communication.

Essentially, the Manchester scheduler tries to find the best matching of the workers with the available work. The Argonne scheduler allows the workers to make local decisions based on their local data, that is, to wait for some new work that might appear in their vicinity.

The Aurora system used in our investigation came with the Argonne scheduler. So we rely on [Ca89] for a comparison of both schedulers. According to their research, the Manchester scheduler performs well, and significantly better than the Argonne scheduler to extract parallelism from fine grain problems. But considering that the overall runtime differences are not very large in most tests, and that these schedulers still under development are workbenches for experimentation and implementation of various strategies, this could not have a very significant impact on our results relying only on the Argonne scheduler. Moreover, as we will see, most interesting speed ups were obtained from the exploitation of coarse grain size work.

Some other issues concerning the creation and termination of work (like workers refusing "to die") should also be examined because they could explain some parasitic or inconsistent behavior of the theorem prover running with Aurora, but these are mostly engineering problems which would take us to far away from our main subject.

In any case, from the point of view of the user, the next topic is more important, and it is
3.3.5 Parallel Programming Style.

Two last issues concern the Prolog side effects and the control of parallelism.

The philosophy of parallel Prolog processing which is behind the development of Aurora is that the programmer should not worry about parallelism [Wa87]. One aim of the Project is to show that conventional Prolog programs can benefit from parallel processing.

This did not prevent some early experiments with the “delta” version of Aurora. Several versions of the same program, a crude version of the theorem prover, were designed in an attempt to discover what effect the programming style had on the performances of the parallel compiler.

This version of Aurora allowed to declare which predicates to run in parallel. Subsequent versions are all parallel by default, and the predicates to run sequentially have to be declared.

One obvious experiment in the framework of Or-parallelism, was to design the program with many redundancies, that is defining the predicates with as many clauses as possible. Doing so, it was thought that the static analysis by the compiler would be enhanced, and more parallelism would be extracted since the number of alternatives to find a solution was increased. Comparisons with timings obtained while running the same program coded in a more classical way showed that no significant improvements could be obtained in this way.

Replacing the different definitions by alternative choices embedded inside the definition of the same predicate (which is not considered as good programming and is, in general, less efficient) did not significantly reduce the execution speed.

A third experiment in which the definitions of the predicates were compacted, that is, one single predicate definition and calls to many different predicates to prove a goal, produced the same results as in the previous experiments. In the end, it appears that none of these programming style had any significant impact on the results.

It should be said that precise timings were always difficult to obtain; they may depend on the load of the machine or some unknown behavior of the program or of the compiler. But on several runs the results were consistent, and rather disappointing since some significant fluctuations were expected when changing the programming style.

It took some more time and experiments to discover that the best speedups were obtained when processing in parallel only the well-defined combinatorial problems. The main conclusion or lesson of such experiments is that, obviously, the compiler will never detect
more parallelism than there is actually in the program. At best, it will find it and execute efficiently, at worst, cumbersome coding style may impede efficient processing. In what concerns Aurora, to quote R. Lusk, a general rule would be "good Prolog programming is good parallel Prolog programming". Then, some knowledge of the compiler and a good understanding of the execution of the program help in the optimalization of the code for efficient parallel processing.

Nevertheless some care should be taken. Mainly when Prolog side effects plays a role in the execution. One common problem, still unsolved, is related to input-output when processing has to be suspended, the results printed on screen or in a file always appear in some random order. But this is a minor problem. More importantly, side effects are created when "cut", the specific Prolog control, is applied during the parallel execution. When this occurs, the pruning of the branch is not executed until the execution of the left side of the search tree is completed. That is, anything being processed before the current branch in which a "cut" occurs has to be executed before pruning the branch. And this can take some time in addition to being a real problem for an efficient parallel processing of a program. To prevent this, Aurora provides an important built-in extra-logical and specific form of cut, called cavalier commit, that allows to return the first solution found and to cut out all other possible solutions by "killing" all other workers. 1

The second issue related to programming style is to control the grain size. Indeed, in principle, all choice points in a program create parallel processes. To answer this question, [Bu88] propose a parallel programming discipline which circumscribes the parallel computation to the parts of the tree which are safe, i.e. without side effects requiring the suspension of the work. They propose to collect the results of the parallel processing of some goal G using the built-in Prolog predicate bagof/3 which returns the multiset of solutions of the goal G in a list L,

\[
\text{get_all_solutions}(X, G, L) :\text{ bagof}(X, G, L).
\]

\[
\text{get_one_solution}(G) :\text{ call}(G),!.
\]

where G is safe, and all solutions are returned irrespective of the ordering of the clauses. If order matters than the predicate should be run sequentially.

Writing a program in such a way that it is safe and as much computation falls into the scope of "get_all_solutions/3" computation on elements independent of the context seems to be a rather restrictive approach with respect to the naive approach which would be to try to process as much as possible in parallel. But, as we will see from the results in Chapter six, this suggestion is consistent and has provided the best results.

One would think that Or-parallelism works best if there are many alternatives and many solutions, but it also works for programs with few solutions, even if there is only one solution, and it is efficient on a single branch as long as the branch is long, i.e. the

1. See addenda, note 7.
We can then think that Or-parallelism is appropriate for our Automated Theorem Prover where very large trees have to be explored and where a lot of combinatorial processing of multisets and other data structures have to be performed.

Nevertheless, in this framework of Theorem Proving, it often happens that the designer of the program wanting to direct the strategy uses a deterministic programming style which then restricts the possibilities of parallelism. This then requires some evaluation of the benefits to expect from parallel processing of specific predicates. In general, it appears that if one accepts the proposed discipline, this forced determinism in the code has no significant impact on the results since the parts selected for parallel processing are exactly those for which it is difficult to impose deterministic constraints.

### 3.3.6 Experimenting with Other Approaches.

In some implementations relying on explicit parallelism, some built-in predicates come on top of Prolog and allow the programmer to create several processes on a multiprocessors machine, providing the ability of opening and closing channels of communication between processes through which information to be processed or processed is sent or received.

The reason to mention this form of parallelism is that some experiments were carried out with an evaluation package of Quintus Prolog which implements such predicates. If the results of running some demo programs were rather consistent with the vendor's reported timings, any attempt to implement a somewhat simple version of the theorem prover and parallelizing the most obvious parallel candidates failed, i.e. ended up with negative speedups. One reason is that Quintus Prolog is a very efficient and fast implementation of sequential Prolog. The overheads of creating processes were too important to attain any significant speed-ups with respect to the sequential execution on one processor. One could think that very coarse grain parallelism would be appropriate to minimize the amount of communication and maximize the work performed. But then, the problem is an unavoidable bottleneck when collecting the information channelled back from the processes.

It is obvious that this compiler offers more a gadget form of parallelism than real parallel processing. But the idea in experimenting along this line of approach was that, contrary to other parallel compilers which leave the programmer confronted with some "black box" compiler taking care of all parallelization, explicit parallelism gave the impression that the user could design his program and effectively implement a parallel algorithm. The naive approach was that if a solution lay on some branch of the search tree, and as many processes as branches were created, then the solution would be found extremely rapidly. This approach is correct as long as the proof search tree is small, that some prediction of
where a solution can be found in the search tree is possible, and the amount of information to channel is reduced. But as we will see, the problem we were dealing with were by far too complex to benefit from this approach.

3.4 Logic programming and Automated Theorem Proving.

[Br87] asks if “Logic Programming is real programming”. It is a question worth asking because very few computer scientists seem to be ready to answer affirmatively. Even though many seem to agree with Hayes’s slogan that “Computation=Controlled Deduction” or Kowalski’s “Algorithm=Logic+control”. If one is ready to follow Dijkstra’s discipline of programming in both frameworks, i.e. the simultaneous design and correctness proof of a program, then Logic Programming and traditional programming languages have a lot in common. What Bruynooghe shows is that it is possible to devise an (ideal) Logic Programming compiler which would be free of side effects and would efficiently control its available resources [Br87]. But, again, it is the responsibility of the programmer to implement the proper guards which would guarantee the completeness and soundness of the deduction process. Then, an efficiently controlled deductive process would guarantee the efficiency of computation as in any other language.

As it is, Prolog is a very efficient language as long as the problems it is asked to solve have a clear logical and algorithmic structure. Outside this specific class of problems it is rather inefficient and in what concerns Automated Theorem Proving, it may simply be hopeless for serious applications, i.e. hard problems.

The perspective which has been adopted in our research is that Prolog may well be appropriate for Automated Theorem Proving if all its shortcomings are circumvented. As we will see in the next chapter, this may well end up in thousands of lines of code devised as “ways round” problems. Hence, the role of Prolog in a Prolog theorem prover should be restricted to what it is good at, to the algorithmic part, i.e. the execution of the proof theory and the implementation of the strategies and heuristics. The core of the automated theorem prover should be Prolog, and anything requiring intensive computation should be done by appropriate low level language routines and by using the resources of fast parallel machines.

Considering some implementations like [St85] or [Fi90], one could be tempted to say that Prolog is the natural programming language for logic.

To fulfil the condition of completeness and soundness, any Prolog program must strictly control the strategy, building in loop detection and some form of occur check. This can be done by keeping a list of all ancestors of a query or applying a selective occur check based on some syntactic analysis of the problem, and by controlling the depth of the
search space or, to insure completeness, by implementing some form of iterative deepening. What is needed for any logical reasoning implementation is a good inference system and a good proof procedure, i.e. an efficient strategy which avoids redundancy, irrelevant derivations etc. Stickel's Prolog Technology Theorem Prover [St85] is certainly one of the best examples of a theorem prover based on Logic Programming and extending Prolog which respects most of these criteria.

But still, we would be tempted to say that no hard problem can be solved by simply extending the Prolog engine and that much more is needed.

Wos and McCune argue that some form of symbiosis of Logic Programming and Automated Theorem Proving can be achieved [Wo89]. Both fields are different: Logic Programming executes efficiently precisely defined algorithms based on logical reasoning, and Automated Theorem Proving is more oriented toward general purpose deduction, or toward more specific and restricted form of deduction in the case of a prover designed for a particular purpose. Moreover in general, the algorithms used in Automated Theorem Proving are not very efficient because of the nature of the problems treated.

Nevertheless they propose to increase the scope of both fields by borrowing some of their respective techniques. And this can be done as a result of experimentation in both fields because both are experimental sciences. The authors claim that Logic Programming is appropriate for symbolic manipulation, but lacks most of the usual strategies used in a theorem prover and which are essential for a successful and efficient implementation. To be fair, the Logic Programming community seems quite aware of its own problems in that respect, and many suggestions have been made to implement some of the strategies of Automated Theorem Proving at the level of compilation of Prolog. The reason this has not been generally done until now are, to a large extent, related to the limitations of the hardware available.

The main obstacles to overcome, of course, are speed and efficiency. Using the resources of parallel computers can solve the first problem. As far as efficiency of a compiler is concerned, a classical Prolog program actually spends a large part of its execution time in computing the same information again and again, going down the tree and back up the tree on backtracking.

Our research in Automated Theorem Proving illustrates clearly this shortcoming of Prolog as a programming language. Quite often one realizes that some pointers or some C "longjump" function would make programming life easier, for example.

But our original intention was to use Prolog, not in the sense of [St85], but simply because it was easy to use and a parallel compiler to experiment with was available. It soon appeared that the complexity of the logic implemented was such that the usual ways of improving the efficiency of a Prolog program were completely insufficient, and some drastic
decisions had to be made which are completely foreign to the spirit of Logic Programming. Doing so, by necessity, our research took an orientation rather close to the suggestions of [Wo89] and it falls more or less into what can be called the ANL perspective of Automated Theorem Proving [Wo90],[Lu90].

If we want to attack a difficult problem, as [Wo89] suggest, we should adopt the attitude of a mathematician who first collects lemmas and intermediary results which are relevant to a solution of the problem. This implies, for example, keeping the interesting information and avoiding irrelevant information, orienting the reasoning toward a solution rather than relying on the non-determinism of the programming language.

We will see in the next chapter that even though some strategies resembling hyperresolution and paramodulation (viewed as rewriting into some normal form and replacing equals by equals) at the preprocessing stage are used in the theorem prover, but not as essential strategies as in resolution theorem provers, it is really strategies like **weighting** and **subsumption** allowing to restrict and direct the logical reasoning which play an essential role. The use of large databases to store the information on which we will come back in Chapter five, is the main example of strategy on which the theorem prover relies. And actually, it is the only way to partly circumvent the inefficiencies of Prolog as well as the complexity of proving theorems in the logic. In return, this strategy used by the theorem prover suggests a way to improve the efficiency of Prolog.

3.4.1 Intelligent Backtracking.

[Br90] shows that the logical reasoning underlying intelligent backtracking, i.e. an efficient pruning of the search space at the object level corresponds to an application of hyper-resolution (here with its usual interpretation of a strategy combining several resolution steps into a single one) at the metalevel, where a metapredicate describes a **conflict set** whose elements are the specific call which creates a conflict node, i.e. a set containing a node from which an unsolvable subset of the set of equations is obtained, the head of the clause used to close the node and its literals.

On failure of unification, a **conflict** is detected in the search tree, i.e. a subset of the closed nodes sufficient to cause the failure. Since that part of the search space which contains a conflict cannot provide a solution, it should be avoided. This technique would dramatically improve the efficiency of Prolog, but it has never been completely implemented in a compiler because there is some "reluctance to use intelligent backtracking unless there is almost no overhead in speed and memory usage during forward execution" [Br90,7], and this is not the case.

We mentionned earlier two ways of considering the problem of intelligent backtracking:
keeping copies of maximal unifiable subtrees, or copies of minimal failure subtrees. If one keeps copies of maximal unifiable subtrees, parallel processing may be of no help because too much communication between processes would be necessary to avoid redundancies. In addition, in this approach the clauses which can still be applied to solve the original call having created a conflict are parts of the description of the conflict, thus, it may simply be impossible to implement this approach in parallel since, depending on the way in which the search tree is distributed among processes, some information, some clauses which are parts of the description, must remain accessible. If, on the other side, these clauses are not part of the description, then one only has to expect communication overheads, and parallel processing could be appropriate. Moreover the problem of determining what to keep or what to leave accessible makes the second suggestion, the minimal failure subset, more plausible.

A second problem in implementing intelligent backtracking is to detect conflict sets. If each binding is incrementally tagged at each step with a set of nodes which will constitute the conflict set when failure occurs, the overhead on forward execution of the program may be too important and may slow down the execution. Tagging a single node at each step forward is too restricted since it would only provide information for a single backtrack point.

Finally, failure corresponding to unsolvable equations, in any case, the problem of finding the minimal unsolvable subsets (or the maximal solvable set in the first approach) is exponential or doubly exponential as we have seen in section 3.2.4. This shows how complex the problem of implementing the technique can be. Actually, it is an intractable problem [Wl89]. Finding the largest maximal unifiable subset in a proof of a problem is NP-complete, and it is an open problem for the minimal non-unifiable subset. Finding all the maximal unifiable or minimal non-unifiable subsets of a problem is a NP-hard problem in the worst cases. As [WZ89] shows, and this applies to Logic Programming, Prolog, and Unification in general, if all the maximal unifiable subsets of assignments are to be computed, the solution length of these subsets is exponential in the length of the input. Wolfram then suggests that if intelligent backtracking is implemented, it should be restricted to a single minimal non-unifiable subset. The author then concludes that this state of affairs justifies the necessity for approximate or heuristic approaches to prevent a program from trashing, i.e. failing to detect that earlier choices have no solution and compute them again. A restricted form of intelligent backtracking could still be implemented.

In a Prolog program, shallow backtracking occurs when unification of a goal and a clause fails and an alternative clause is tried, and deep backtracking occurs when unification of the last clause of a procedure with a goal fails and control is returned to another goal.
One better solution, then, is to allow “shallow” backtracking (that we may represent as in Figure 6) and delay the copying until deeper backtracking happens, i.e. when all successive possibilities of solving the call have been exhausted and the system backtracks deeper (upward in the tree), and only update one global conflict set which is incremented when all shallow backtracking is completed. Many other possible solutions involving complex data structures are theoretically possible. The only problem is that it is “too expensive to be included in the implementation of a programming language” [Br87,8].

![Deep Backtracking and Shallow Backtracking](image)

Figure 3.6: Deep and Shallow Backtracking.

What we have to propose with respect to intelligent backtracking may appear as a first approximation of a partial solution to this problem, which, after all, given its complexity, cannot have a complete solution anyway. If we consider the execution of the proof theory by the theorem prover that will be presented in the next chapter as an emulation of the execution of Prolog, then, as suggested earlier, some insight gained in designing and experimenting with the theorem prover could throw some light on possible improvements of Prolog. In order to obtain an efficient theorem prover, some form of intelligent backtracking based on heuristics and closely related to the preceding propositions had to be implemented.

When using databases of proved and refuted formulae in the implementation of the theorem prover, what we do is actually pruning the search tree, discarding all search paths which we know would end up in a proof or in a refutation. Searching sequentially a database is time consuming. As we will see, using the resources of massive parallelism of the Connection Machine trivializes the problem (within the limits of available memory space, but this is very large) since access time is reduced to a few milliseconds, and checking one element or all elements of the database costs the same. The availability of this new technology allows to suggest an improvement of the efficiency of Logic Programming in implementing the technique of “intelligent backtracking” based on “delayed conflict sets”.

A first remark is that only the real logical inferences in the execution of a Prolog proof should be counted as logical processing steps to take into account, their conflict set containing all results of non logical computation. As [Wo90] put it, it may be hard for
a logician to accept that appending two lists represents several logical inferences. And a large proportions of those lips do just that, repeating elementary operations which, all together, result in the success or failure of a logical step in the execution of the program. This way of seeing the Prolog execution reduces the amount of real lips in a significant way, and allows to consider rather complex data structures, i.e. complete subparts of the search tree, which could be stored and checked in a reasonable amount of time without impeding significantly the forward execution of the program. It is then a matter of experimentation and of evaluation of the trade-off between computing in memory or consulting an external database.

A second remark is that any implementation of intelligent backtracking, as suggested above, would overload the system. This overloading always happens when experimenting with large compiled databases, and external storage with fast access is necessary. Finally, some evaluation of the procedure described has been carried out to assess the efficiency of interfacing Prolog running sequentially with a Connection Machine and the benefit or overhead of suspending its execution to check large databases. We will report the results and precise figures later on in Chapter five and six and show that, in the long run, it is feasible to consider a Connection Machine as a very fast and efficient matching or unification machine.
Chapter 4
Automated Theorem Proving in LR.

4.1 Introduction.

In general, the complexity of the logic LR makes it intractable, and Prolog is a very inefficient programming language to implement an automated theorem prover for this logic. Nevertheless, we will see in this chapter that, to some extent, it is possible to control the complexity of the logic, and that some of the deficiencies of Prolog can be circumvented.

The axiom or rule of contraction is mainly responsible for the complexity of the logic, but a clever formulation of the proof theory, [TMM88], allows to control the effect of contraction by building it into the rules. In addition, some provable properties of the logic further constrain the complexity. In this respect, it appears that, together with some of these properties used as filters, the admissible $K_a$ rule plays an essential role in the proof of difficult problems.

A Prolog theorem prover will obviously suffer from all the shortcomings of the language, some of which were described in the preceding chapter. On the other hand, the language is quite appropriate to implement the algorithmic part of the theorem prover, the proof theory, in a way following closely its logical formulation, thus making a completeness and correctness proof easy to verify. Moreover, implementing some techniques which prevent useless redundant computation, significantly enhances the efficiency of the Prolog theorem prover.
4.2 Dealing with Complexity in ATP.

Decidability of a logic may be a virtue. But mere decidability does not guarantee that there exists a practical and effective decision procedure. If efficiency is measured with respect to the size of the instance to which a decision procedure is applied, relatively short sentences can define relatively large sets. Suppose that we build a computer with switch gates and, or, not. How many such components are needed to decide some fixed length formula? \[\text{Me74}\] shows that if we consider sentences of length \(n\) of some formal theory, whose symbols are coded as binary sequences, each symbol requiring six binary digits, i.e. sentences of length \(n\) corresponds to binary sequences of length \(6n\), the program to compute the binary valuation, i.e. the evaluation to the value true of a \(616\) symbols sentence requires \(10^{123}\) operations, that means as many components for our computer. Obviously, one may want to consider simpler instances of a logical theory.

Before the advent of automated theorem proving, the complexity problems and the efficiency of methods to prove theorems did not matter too much. \[\text{Wa60}\] gives an example of a first-order formula \(\exists x \forall y \forall z((G_{yy} \& G_{zx}) \supset (G_{zx} \& G_{zz}))\) which, by Herbrand's theorem, is a theorem iff \(\exists N \text{ s.t. } s_1 \vee \ldots \vee s_N = D_N\) is a truth functional tautology. Computing \(D_N\), according to traditional methods, i.e. \(s_1:(x,y,z) = (1,2,3)(G_{22} \& G_{11}) \supset (G_{31} \& G_{33}), s_2: \ldots,\) the value of \(N\) is \(N \geq 2^{48} - 1\). And in the general case, for a formula of the form \(\exists x \forall y_1, \ldots \forall y_n P(x,y_1,\ldots,y_n)\), where \(P\) contains \(N\) predicates, testing that some instance such that \(n = 2\) and \(N = 4\) is a theorem iff \(D_K\) is a theorem, a bound for \(K\) can be found, \(K = 8^{65536} - 1\), or \(K = 65535\) depending on the method used.

Of course, the methods to deal with complexity have evolved. For example, the satisfiability problem (SAT), which is \(NP\)-complete is intractable, but it is efficiently solved in the case of Horn clauses and formulae in conjunctive normal form with at most two positive literals. But this is a restriction of the problem to easily solvable cases, and it also means abandoning completeness of the full satisfiability procedure of ground clauses. This is the solution adopted for Prolog, which, according to \[\text{Bi90}\], is still the greatest contribution of the field of automated theorem proving to logic programming and automated reasoning.

Even though efficient proof procedures are now available, all problems are not solved. One of the earliest method largely used in theorem proving, the Davis-Putnam procedure \[\text{Da60a}\], consisted in the application of Herbrand's procedure, i.e. the enumeration and test of the finite set of instantiations of a set of formulae in normal form. This very inefficient method was replaced by Robinson's resolution method \[\text{Ro65}\] which still suffered from inefficiencies \[\text{Ha85}\]. Many attempts, like the connection graphs approach of Kowalski \[\text{Ko74}\] and Bibel, to restrict resolution in such a way that it is an efficient procedure
which, at the same time, preserves completeness were made. Nevertheless, with these procedures, the search space can still be very large, and, moreover, the reduction to normal form may be expensive.

As [Bl77] observes, trying to solve these problems in considering non clausal forms, i.e. the Horn clauses not necessarily in conjunctive normal form, may help in partly solving the problems, but the main stream of research has focused on completeness of the procedures, while, as exemplified by the Boyer-Moore theorem prover [Bm90], which is not complete, important and interesting results can be obtained without completeness.

The same remarks also apply to procedures based on rewriting systems where the reduction to canonical form of expressions before application of the rewrite rules and test of equational consistency, i.e. the equality of two given terms, may be computationally expensive. Moreover the system of equations may be overgenerating under the given rules, and, finally, these systems may terminate in failure or fail to terminate.

Obviously, the right methods or procedures have not been found yet. Considering the field of automated theorem proving today, one has the impression that there are two rather different and opposed approaches to the problem of automating theorem proving. A case example used by [Bi90], is Lukasiewicz’s unique axiom for the implicational fragment of the propositional calculus [Pf88]. One approach advocates a solution to the problem that is close to Lukasiewicz’s proof, which consists in 29 steps. Essentially, the method would be based on some refinement of existing techniques like the connection graphs, the matrices method, or some other method. This amounts to require the theorem prover to behave in a manner closely similar to the behavior of the logician, i.e. to be intelligent. The other approach, not necessarily representative of the brute force approach, is based on heavy use of powerful computational resources. For example, Otter, which is the target of Bibel’s remark, gives a proof of the problem based on hyperresolution in 150 steps, but generates 6.5 millions mostly redundant clauses.

In this work, we started following roughly the first perspective, that is, trying to find heuristics, strategies and some properties of the logic which would make the theorem prover more efficient and still complete. Having learned the lessons of the logic, its complexity and the difficulty to prove what would appear as rather simple theorems, we became more and more directed, and even forced, toward the second perspective, as will be clear from this chapter and the next one. But, in the end, it will also appear that none of these approaches alone would succeed. Obviously, the implementation of heuristics is essential, and this requires processing power too.
4.3 A Gentzen Style Theorem Prover

Several early proposals in ATP were based on natural deduction and Gentzen systems. These systems received a lot of attention in Computer Science during the last decade due to the "proof as computation" paradigm, or due to their "naturalness" for automated theorem proving. This approach to theorem proving relies on the backward chaining procedure of natural deduction systems in which the rules of inference are used to convert a given goal into a set of subgoals.

One problem with this approach is, again, to control the generation of facts or subgoals. Even though, in general, backward chaining can be controlled by a criterion of simplicity, starting from a complex problem and decomposing it into simpler subproblems, at first thought, this strategy may not be appropriate to deal with a logic like LR where contraction has just the opposite effect. Nevertheless, LR has a very elegant and efficient Gentzen formulation which allows to control contraction and to limit its effect.

The effect of contraction in the application of a rule is to allow the complexity of a multiset in a branch of the search tree to grow without bounds. This means that, even for simple formulae, a proof can be very long. And, considering an arbitrary multiset candidate to a proof, the complexity of its proof is almost impossible to predict at the start.

One may wonder whether this does not make the enterprise of ATP in the context of LR hopeless? In worst cases, no proof will be found in a reasonable amount of time. But there is not much difference between an automated theorem prover for this specific logic and any other system using any other proof procedure. All are facing the same problems of efficiency and termination, as we have just seen.

Even though the enterprise seems to be confronted with an intractable level of complexity, it is nevertheless possible to decrease this complexity to some extent, and, as shown in [TMM88], it is possible to build an efficient automated theorem prover for this logic.

In a LR proof, Curry's property terminates the growth of any redundant branch, and so, controls the size of the search space. But this search space can still be very large. Following the observation that decidability is obtained from constraining the use of contraction, the authors formulate several alternative Gentzen formulations of LR, ending up in a formulation in which contraction is completely embedded into the non-invertible rules for disjunction and fusion.

An invertible rule is such that if $\alpha = [\beta, A, B]$ is a multiset in which $A$ and $B$ are complex, and if $\alpha$ was derived by some rule with $A$ selected as principal constituent, then $\alpha$ could have been derived by another rule with $B$ as principal constituent. In this case, the second rule is invertible with respect to the first rule [TMM88,48].
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

The essential consequence of invertibility is that the application of such rules is irrevocable in the sense that it does not prejudice a solution to the problem. In the case of an invertible rule, if \( \alpha \) contains a formula, \( A \), which could be principal under the application of the fission and conjunction rules, then one need to construct only the part of the search tree that has \( A \) as principal. With the rules of disjunction and fusion which are not invertible, particular applications and choices may very well determine a solution of the problem. We will see in Chapter 6 that the selection of constituents as principal and parametric in the application of a rule may make all the difference between an easy and a complex proof.

The proof theory implemented in the theorem prover that we will examine in section 4.5 corresponds to the formulation \( L_5 \) of LR [TMM88]. This formulation improves considerably over the other possible \( L_i \) formulations by restricting the use of contraction. Indeed, the effect of contraction is entirely built into the fusion and disjunction rules, and it is restricted by their conditions of application.

The authors discuss successive improvements made in various formulations of the proof theory. If we consider the simplest and most basic formulation \((L_1)\), corresponding to the operational rules given in the next section, but unconstrained, with, in addition, the structural rule of contraction explicitly stated, in an elementary implementation of the \( L_1 \) proof theory, where contraction applies freely, nothing can prevent its application at almost every step of a proof. Without the Curry property, nothing could stop the generation of more and more complex multisets, making a proof of any formula practically impossible. And even so, without some other provable properties of the logic, nothing could prevent the generation of very long provable multisets like the following theorem for example, \((\sim b \lor b) + (\sim b \lor b) + ((\sim a \circ \sim a) \lor (\sim b \lor b) \circ a) + (\sim b + (\sim a + (a + (a + (b + (b + (b + (b + (b + (b + \cdots + b + \cdots + b) \cdots))))))))))

The improvement obtained by the \( L_5 \) formulation of the proof theory translates immediately into an improvement of the complexity of a proof as shown in [TMM88], and illustrated in the following sections.

4.4 Where Complexity Strikes.

We have seen in Chapter two that the lower complexity bound of LR is EXPSPACE and its upper bound is DTIME in the Ackermann function. Bounding the number of propositional variables to two reduces the complexity of the logic to NP-hardness, i.e. as hard as any \( NP \)-complete problem, and, in general, bounding the number of variables leaves LR in PSPACE.

We became recently aware of Girard's et alii's work on bounded Linear Logic [Gi90], where bounds are placed on the number of applications of the exponential operators “of course”
and "why not". These operators allow to express contraction in Linear Logic and, thereby, give Linear Logic the expressive power of Intuitionistic Logic. Then, as in Intuitionistic Logic, the proofs are guaranteed to terminate, but there is no control of the size of a proof. The authors show that constraining the application of these operators makes cut elimination time-proportional to an integral power of the size of the argument, and allows to reduce the complexity of the logic to P-time. Imposing such bounds on the application of the contraction rule of LR would probably provide the same result, but the logic would no longer be the same, of course.

The logic LR is thus simply intractable. But high level of complexity should not prevent computation on realistic cases, and by realistic, we do not necessarily mean simple cases, because LR being more complex than other logics, seemingly simple and easy instances of LR may prove to be very hard to prove or to refute. And it is not obvious to determine a priori the complexity or the length of the proof of a given LR formula.

What we are interested in, is to discover why, and where, complexity strikes, and to what extent it can be controlled in practical cases.

A first source of complexity is the use of multisets of subformulae rather than sets as data types [Me82]. In a Gentzen system $A \vdash B$ means that the multiset of formulae $B$ is derivable from the multiset $A$, and repetition of subformulae is allowed.

A second source of complexity stems from the axiom or structural rule of contraction i.e. $A$ is proved if $A, A$ is proved, (Girard's "way to infinity") and the absence of the rule of weakening. In a right sided Gentzen formulation of the logic, where $\alpha, \beta, \gamma$ range over multisets, these rules, or inference schemata, are written

$$
\vdash \alpha, A, A \quad \text{Contraction} \quad \vdash \alpha, A, A \quad \text{Weakening}
$$

We illustrate how complexity builds up in the application of the fusion rule:

$$
\vdash \beta, A \quad \vdash \gamma, B \quad \vdash \alpha, A \circ B \quad \text{Fusion}
$$

As we will see, several conditions related to the cardinality of the multisets $\alpha, \beta, \gamma$ and the possible occurrence of repetitions of $A \circ B$ in each of them constrain the application of the rule.

### 4.4.1 Complexity in Action.

We first need to define, following [Cu63], the principal and parametric constituents of a formula.

**Definition 4.1** A formula in the conclusion of each application of a rule is called the principal constituent of the rule. A formula which passes from a premise to a conclusion
without change is called a **parametric constituent** of the rule.

Following Curry's suggestion [Cu50], and as shown by the authors of [TMM88], one efficient way of constraining contraction is to build its effect into the other connectives rules rather than stating it explicitly in the proof theory.

In the fusion rule, contraction is allowed on parametric and principal constituents. This makes the premises more complex than the conclusion, and hence, the number of premises sets from which these premises may have been derived is more complex, and so on.

This complexity generated by the fusion rule can be exactly determined. Let $a$ be the multiset $[C_1^{n_1}, \ldots, C_k^{n_k}]$, where $n_i$ is the number of instances of subformulae $C_i$ in $a$, i.e. its cardinality, $\text{card}(a)$. McRobbie showed that the upper bound on the number of pairs of premises from which a given multiset $a$ with fusion in its principal constituent could be derived is $4 \times 3^{\text{card}(a)}$.

In general, for any multiset, the function can be generalized giving an upper bound of $k \times 4 \times 3^n + 4l + 2m$ on the number of immediate subgoals where $k$ is the number of fusion, negated fission, $l$, the number of disjunctions, $m$, the remaining complex formulae and $n$ is the number of parameters [TMM88].

Here, we will only be interested in the fusion rule. The conditions imposed on the application of the fusion rule allow to reduce the number of possible premises pairs in most cases.

There are $n_i + 1$ ways of partitioning the parameters into left constituents, that is, the number of possible left parametric constituents is equal to $P(\alpha) = \prod_{i=1}^{k}(n_i + 1)$. Let $\text{card}(X; \alpha)$ be the cardinality of $X$ in the multiset $\alpha$. [TMM88] define a count function which determines the number of possible premises pairs in any application of the fusion rule: $\text{count}(A \circ B; \alpha) = P(\delta) \times 3^k \times \min(4, (4 \times \text{card}(A \circ B; \alpha^\delta)))$, where $\delta = \{C\}$ such that $\text{card}(C; \alpha) > 1$, $k$ is the number of generators $D$ in $\alpha$ such that $\text{card}(D; \alpha) = 1$.

The improvement obtained by constraining contraction in $L_5$ is rather impressive, as shown in [TMM88], table 2.2.5. In the worst case, when $\text{card}(D; \alpha) = 1$, the number of premises pairs is still given by the McRobbie function. For 5 different parametric constituents, the numbers of premises pairs is 972; with 5 identical constituents, it is now reduced to 24 while, it would still be 972 in the original formulation of the proof theory.

Suppose that we want to prove a formula like the following simple formula, call it $\alpha$:

$$[a \circ (c \circ \neg d) \circ ((b \circ (d \circ \neg e)) \circ (\neg a + (\neg b + (\neg c + e))))]$$

For simplicity we reduce it to $\alpha = ((A \circ B) + C + D + E + F + G)$.

The proof search tree of $\alpha$ is constructed by application of the rules of the proof theory in reverse so that the possible premises which could have $\alpha$ as conclusion are generated and assigned to the nodes of the tree immediately following $\alpha$. Then the possible premises of
the first two premises of $\alpha$ and so on, until the tip of each branch, i.e an axiom, $[A, \neg A]$, is reached.

In our example, the principal constituent is $(A \circ B)$, and $[C, D, E, F, G]$ are the parametric constituents. According to the McRobbie function, the maximum number of possible pairs of premises to consider in an application of the fusion rule to $(A \circ B)$ is $4 \times 3^5$.

Applying the fusion rule to the principal constituent, a proof of its left constituent, $A$, is first derived using all possible combinations with the parametric constituents. When a proof is found, all legal combinations with the remaining parameters (depending on some appropriate partition of the constituents between $A$ and $B$ to which we will come next) are used to derive a proof of $B$. Call these combinations of $B$ the parametric complements of $A$. Obviously, in searching for a proof, new combinations of fusion subformulae are created, for example, $[A, C]$, where $C$ is $(b \circ (d \circ (e \circ \ldots )))$, increasing further the complexity of finding a proof.

### 4.4.2 Proof Theory.

Axiom: $p, \neg p$

Operational Rules:

- **Invertible Rules:**
  
  \[
  \begin{align*}
  &\vdash \alpha, A, B \\
  &\vdash \alpha, A + B \\
  &\vdash \alpha, A \& B \\
  &\vdash \alpha, A \lor B
  \end{align*}
  \]

- **Non Invertible Rules:**

  - **Disjunction Rules.**
    
    \[
    \begin{align*}
    &\vdash \beta, A \\
    &\vdash \alpha, A \lor B \\
    &\vdash \beta, B
    \end{align*}
    \]

  - **Conditions of application of disjunction:**
    
    \[
    \begin{array}{|c|c|}
    \hline
    \text{card}(A \lor B; \alpha) = 0 & \beta = \alpha \\
    \hline
    \text{card}(A \lor B; \alpha) > 0 & \beta = [\alpha, A \lor B] \\
    \hline
    \end{array}
    \]

  - **Fusion Rule.**
    
    \[
    \begin{align*}
    &\vdash \beta, A, \gamma, B \\
    &\vdash \alpha, A \circ B
    \end{align*}
    \]

  - **Conditions of application of fusion:**
4.4.3 Controlling the Complexity.

In addition to the applicability conditions of the disjunction and fusion rules, the Curry’s property, as well as other properties of the logic, allow to control the complexity of the proof theory.

**Definition 4.2** A multiset $\beta$ is strongly contained in a multiset $\gamma$ if (i) $\forall A$ in $\beta$ and $\gamma$, $\text{card}(A;\beta) \leq \text{card}(A;\gamma)$, and (ii) if $g(\beta) = g(\gamma)$ where $g(\alpha)$ represents the generators of $\alpha$, i.e. the set of elements of $\alpha$.

**Definition 4.3** [Curry’s Property] A potential proof $\tau$ of $\alpha$ has the Curry Property iff $\forall \beta, \gamma \in \tau$, if $\gamma$ is a successor of $\beta$ in the sense that $\gamma$ lies in some branch from $\beta$ to a tip of $\tau$, then $\beta$ is not strongly contained in $\gamma$. I.e. $\beta$ cannot be derived from $\gamma$ by a series of contractions and permutations.

What makes this property important, apart from being essential in the decidability procedure of the logic (it is our earlier irredundancy condition), is that when a new multiset is generated at a node of the tree and is identical to a multiset already occurring on the path back to the initial node, then the growth of the current branch is stopped. This prevents the generation of identical multisets on the same branch and excludes multisets strongly contained in a multiset occurring earlier on the branch. Hence, before trying to
prove any candidate formula, a test is performed to check if it satisfies the property or not.

In the $L_5$ formulation of the proof theory, the Curry Property is further constrained to apply to multisets which, in the proof of some formula, do not contain invertible formulae. Several provable properties of multisets allow to simplify further the derivation of a proof. Since we will explain these properties through their implementation, we just mention them here and delay the explanation until the next section.

- **Preprocessing.**
  - negation normal form,
  - associativity and commutativity of the operators,

- **Filters.**
  - the positive and negative parts property,
  - the strict parts property,
  - the rule of 2 property,
  - the matrix property,
  - the $K_0$ rule, and
  - the derived axioms property.

### 4.5 A Prolog Theorem Prover.

We can now concentrate on the Prolog implementation of the proof theory in the theorem prover. Doing so, some of the characteristics of the language advertised in Chapter 3 will become apparent. More importantly, the explanations of the execution of the program will allow us to show that it is sound, complete, and correct. To keep the exposition simple, the version which is presented here is the distributed version due to P. Thistlewaite. A few changes are introduced to make it easier to follow, mainly, in using the standard Prolog definitions of accessory predicates. This short code implements exactly the proof theory of LR and its decision procedure and so, even though some other strategy can be used in the implementation of the rules, there is no point in presenting an alternative implementation which would only differ in non essential details.

In order to better understand how the proof theory is implemented and make the reading easier, the reader may want to refer to Figure 1 which represents the self-explanatory flowchart of the execution of a proof by the theorem prover. In the schema, the procedures
of selection of constituents and of application of the rules are expanded on the right of the picture. The dotted lines indicate backtracking, and the “!” is the Prolog “cut” operator which imposes one single choice or solution and prevents backtracking.

The goal of the program consists in answering a query about some input formula by trying to satisfy the goal and print “Provable” if it succeeds or “Not Provable” if it fails. The code is transcribed in its logical order, the few accessory predicates which are not immediately obvious are explained at the end of this section.

4.5.1 The Basic Prolog Program.

is_provable(Formula) :-
    provable(Formula, [], 0),
    write(' Provable ') ; write(' Not Provable ').

A formula to test is fed to the theorem prover which tries to satisfy the goal is_provable/1. The input formula is actually first translated into negation normal form and simplified in
various ways, but this does not need to concern us for the moment. The goal is satisfied if the formula is provable, otherwise, (indicated by the Prolog operator ";"), it fails and the formula is not provable.

\[\text{provable}(X, \text{UpPath}, \text{Depth}) : - \]
\[\text{not( curryContained}(X, \text{UpPath}) ) ,\]
\[D \text{ is Depth+1,}\]
\[\text{select}(X, \text{Principal}, \text{Param}),\]
\[\text{Principal } = . [\text{Op|Args}],\]
\[\text{apply}([\text{Op|Args}], \text{Param}, [X|\text{UpPath}], D),\]
\[\text{write(' Proved '), write([X,D])}.\]

The formula \(X\) is provable on a branch of the proof search tree \(\text{UpPath}\) at some depth \(\text{Depth}\) if it is not already on the path of the branch back to its root, i.e. the check of the Curry’s property, and if, having selected a principal constituent for \(X\), the application of the rule corresponding to the main operator of the principal constituent -obtained by the system “\text{univ” predicate “=}...” which decomposes the formula into its main operator or functor and arguments- succeeds.

\[\text{select}([\text{Principal|Parameters}], \text{Principal}, \text{Parameters}) : - \]
\[\text{Principal } = . [’+’|\_], ! ; \text{Principal } = . [’&’|\_], !.\]

As we will emphasize later on, the selection of a principal constituent is crucial in the execution of the proof theory. The application of the selection rule first tries to select an invertible formula, and as we will see, the multisets being ordered (and this is assumed here), if there is such a formula, it will be selected first. The cut operators, !, are essential here, and guarantee that the selection of such principal is an irrevocable choice since it allows only one solution. If there is no such candidate, then the selection is made on a non-invertible formula:

\[\text{select}(\text{Multiset}, \text{Principal}, \text{Parameters}) : - \]
\[\text{msetGenerators}(\text{Multiset}, \text{Generators}),\]
\[\text{member}(\text{Principal}, \text{Generators}),\]
\[\text{delete}(\text{Principal}, \text{Multiset}, \text{Parameters}).\]

The generators of a multiset are the set of its elements, that is, if \([A \circ B, C, D, D]\) is a multiset, its generators are \(\{A \circ B, C, D\}\). We will call \(A \circ B\) a complex generator and \(C, D\), atomic generators, and their respective positive/negative parts, complex and atomic positive/negative parts.

If there is no invertible formula to select, then each complex generator of the multiset extracted from the multiset by \text{msetGenerators}/2 is a potential candidate principal selected by \text{member}/2 which, on failure of the former choice, will successively return each complex generator as principal until the list is empty, and then fail.Obviously, the parameters correspond to the multiset minus the selected principal constituent.
apply([‘+’,A,B], Parameters, UpPath, Depth) :-
    append(A, Parameters, Left),
    append(B, Left, Premise),
    prove(Premise, UpPath, Depth).

The fission rule is very simple. It creates a single premise by concatenating the arguments of the fission operator with the parameters. The rule succeeds if there is a proof of the premise. If it fails, it does so ineluctably since no other choice is permitted.

We skip the transcription of the conjunction and disjunction rules which rely on the same simple mechanism of concatenation.

The conjunction rule is similar, except that it is a two premises rule, each premise being the concatenation of the parameters with, respectively, the left and right argument of the conjunction operation. The rule succeeds if each premise is provable. And the same remark as for fission applies in case of failure.

The disjunction rule is a little more complicated. First, a disjunction is provable if one of its disjuncts is provable. There are thus two rules, one for each disjunct, and the premises of each rule are generated, as in the former cases, by concatenation of each disjunct with the parameters. As can be seen from the conditions of applications of the disjunction rule, if neither of these disjuncts is provable, and if a copy of the principal constituent is not already present in the parameters, we can try to prove either of the disjuncts with a second pair of rules in which the principal constituent is contracted in both rules, i.e. the premises are now constructed by concatenation of the disjuncts with the parameters and with a copy of principal.

The fusion rule is the most complicated. It is also the most interesting in that it will allow us to build some strategy into the theorem prover:

apply([o,A,B], Parameters, UpPath, Depth) :-
    skip_Repeated(Parameters, Non_Repeated),
    partition(Parameters, Leftparameters, RightparameterBase),
    append(A, Leftparameters, LeftPremise),
    prove(LeftPremise, UpPath, Depth),
    selectRight(RightparameterBase, Non_Repeated, Rightparameter),
    append(B, Rightparameter, RightPremise),
    prove(RightPremise, UpPath, Depth).

In what concerns a Prolog implementation, the preceding rules are almost a programming exercise. This, in no way minimizes P. Thistlewaite's contribution. To his credit, we must add that his implementation of the fusion rules is an example of very elegant programming style. In this respect, his definition of partition/3, the partition of a set, a quite standard predicate, has no equivalent in the Prolog libraries of programs publicly available.
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

Coming back to the rule, any repetitions in the parameters are first tested and the non-repeated elements are kept in **Contracts** for later use. The parameters are then partitioned into left and right parameters, as is more easily seen in the following examples which exemplify the application conditions in the cases of \( \text{card}(C; a) = 1 \) and \( \text{card}(C; a) > 1 \) respectively.

**Parameters** = \([A,B,C]\),

**Left** = \([A,B,C], [B,C], [A,C], [C], [A,B], [B], [A], []\)

**Right** = \([[], [A], [B], [A,B], [C], [A,C], [B,C], [A,B,C]]\)

**Parameters** = \([A,A,A,A]\),

**Left** = \([A,A,A,A], [A,A], [A], [A], []\)

**Right** = \([[], [A], [A], [A,A], [A,A,A,A]]\)

Suppose the principal constituent is \(X \circ Y\), then appending the left constituent \(X\) to each partition gives 8 possible left premises in the first case, and 5 in the second.

**selectRight(RightparameterBase, Non_Repeated, Rightparameter)** :-

\[\text{difference(Non_Repeated, RightparameterBase)},\]

\[\text{append(RightparameterBase, Non_Repeated, Rightparameter)}\].

The right parameters to append to the right constituent of the fusion formula are determined by the **selectRight/3** rule. This is a simple, but very clever way to code the partition of parameters in the application conditions of the rules. The elements which were not duplicated in the parameters, i.e. elements \(C\) such that \(\text{card}(C; a) = 1\) may figure in both the right and left partitions at the same time. If they are not already in the right partition, then they are added to the right selection of parameters to constitute the right parameters. This is shown on the following example, where the parameters are \([A,A,A,B]\), where \(\text{card}(A; \alpha) = 3\), \(\text{card}(B; \alpha) = 1\), and principal \(X \circ Y\).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>([B,Y])</td>
<td>([A,B,Y])</td>
<td>([A,A,B,Y])</td>
<td>([A,A,A,B,Y])</td>
<td>([B,Y])</td>
<td>([A,B,Y])</td>
<td>([A,A,A,B,Y])</td>
</tr>
</tbody>
</table>

In the simple case of \(\text{card}(A; \alpha) > 1\), we obtain the following distribution i.e. with \(X \circ Y\) as principal and \([A,A,A]\) as parameters,

<table>
<thead>
<tr>
<th>Left</th>
<th>([A,A,A,X])</th>
<th>([A,A,X])</th>
<th>([A,X])</th>
<th>([X])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>([Y])</td>
<td>([A,Y])</td>
<td>([A,A,Y])</td>
<td>([A,A,A,Y])</td>
</tr>
</tbody>
</table>

To better see the effect of the partition and of the construction of right parameters, consider again the first example of parameters \([A,B,C]\) above. The right selection now
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

generates the following parameters:

<table>
<thead>
<tr>
<th>Left</th>
<th>[A,B,C]</th>
<th>[B,C]</th>
<th>[A,C]</th>
<th>[C]</th>
<th>[A,B]</th>
<th>[B]</th>
<th>[A]</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>[A,B,C]</td>
<td>[B,C]</td>
<td>[A,C]</td>
<td>[C]</td>
<td>[A,B]</td>
<td>[B]</td>
<td>[A]</td>
<td>[]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C]</td>
<td>[A,C]</td>
<td>[B,C]</td>
<td>[B]</td>
<td>[A,B]</td>
<td>[A]</td>
<td>[B]</td>
<td>[C]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C]</td>
<td>[A,C]</td>
<td>[B,C]</td>
<td>[B]</td>
<td>[A,B]</td>
<td>[A]</td>
<td>[B]</td>
<td>[C]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C]</td>
<td>[A,C]</td>
<td>[B,C]</td>
<td>[B]</td>
<td>[A,B]</td>
<td>[A]</td>
<td>[B]</td>
<td>[C]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C]</td>
<td>[A,C]</td>
<td>[B,C]</td>
<td>[B]</td>
<td>[A,B]</td>
<td>[A]</td>
<td>[B]</td>
<td>[C]</td>
</tr>
</tbody>
</table>

In this example, the result of contracting the parameters is an empty list. There are 27 possible combinations of right premises to consider in the application of the rule to \([X \circ Y, A, B, C]\). Suppose that none of these combinations are provable, irrespective of the fact that some of them can pass the filter tests and start to grow a branch down the search tree, they will eventually fail, and on failure, the program will backtrack and try another combination. Summarizing, the program will try all the following possibilities:

<table>
<thead>
<tr>
<th>Left</th>
<th>[A,B,C,Y]</th>
<th>[B,C,X]</th>
<th>[A,C,X]</th>
<th>[C,X]</th>
<th>[A,B,X]</th>
<th>[B,X]</th>
<th>[A,X]</th>
<th>[X]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>[A,B,C,Y]</td>
<td>[B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
<tr>
<td></td>
<td>[A,B,C,Y]</td>
<td>[A,C,Y]</td>
<td>[B,C,Y]</td>
<td>[B,Y]</td>
<td>[A,B,Y]</td>
<td>[B,Y]</td>
<td>[A,Y]</td>
<td>[Y]</td>
</tr>
</tbody>
</table>

We now turn to the second rule for fusion.

apply\([o(A,B), Parameters, UpPath, Depth]\) :-
not( member(o(A,B),Parameters) ),
UpPath = [Xparameters|_],
skip_Repeated(Xparameters, Non_Repeated),
partition(Xparameters, Leftparameters, RightparameterBase),
append(A, Leftparameters, LeftPremise),
prove(LeftPremise, UpPath, Depth),
selectRight(RightparameterBase, Non_Repeated, Rightparameters),
append(B, Rightparameters, RightPremise),
prove(RightPremise, UpPath, Depth).

If all possibilities of application of the first fusion rule have been exhausted and all failed, a proof with the second rule is tried. In case \((X \circ Y)\) is not in the parameters but is principal, we have the following situation for \(\alpha = [X \circ Y, A, A]\), where the parameters
are \([A, A]\):

<table>
<thead>
<tr>
<th>Left</th>
<th>([A, A, X, XoY])</th>
<th>([A, A, X])</th>
<th>([A, X, XoY])</th>
<th>([A, X])</th>
<th>([X, XoY])</th>
<th>([X])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>([Y, XoY])</td>
<td>([XoY, Y])</td>
<td>([A, Y, XoY])</td>
<td>([A, Y, X])</td>
<td>([A, A, XoY, Y])</td>
<td>([A, A, Y])</td>
</tr>
</tbody>
</table>

Now, in order to fulfil the conditions of application of this second fusion rule, contraction must apply when \(\text{card}(A \circ B; \alpha) = 0\). This is the reason why a check that a copy of the principal is not already contained in the parameters is first made. In this case, the partition operates on \(X\) parameters, that is, on the end of the current branch consisting of the entire formula to prove, the current formula which was introduced into the tree at the time of the selection. For example, suppose the formula to prove was \([A \circ B, C, D, E]\), the principal is \([A \circ B]\) and parameters \([C, D, E]\). In the execution of \(\text{selection/3}\), \([A \circ B, C, D, E]\) became the current tip of the branch. Since there are no fusion formulae in the parameters, in order to apply the rule of fusion in such a case, the copy of \([A \circ B]\) is found in the original multiset.

This concludes the explanation and exemplification of the implementation of the proof theory.

The predicate \(\text{prove/3}\) checks if the formula is already proved. If it is not proved, it tests the formula against the filters. If the formula passes the test, then the execution proceeds, trying to find a proof, and "1" prevents any loop, that is, \(\text{proveable/3}\) must succeed or fail.

Note that the filter test is performed in the application of non invertible rules alone since the possible premises are not entailed by the conclusion. It is not performed in the application of an invertible rule since provability is preserved in both directions of the application of the rule.

\[
\text{prove}(X, \text{UpPath}, \text{Depth}) :-
\]

\[
\text{proved}(X) ; (\text{filter}(X), \text{proved}(X)), !. 
\]

The predicate \(\text{proved/1}\) simply expresses a fact. It succeeds if its arguments unify with an axiom.

\[
\text{proved}([\neg(X), X]). 
\]

We will come back to the multiset containment and satisfiability of the Curry property in the next chapter. For the moment, we note simply that \(\text{curryContained/2}\) tests the strong membership of the current formula in the tree along its ancestor branch.

\[
\text{curryContained}(X, \text{[Up_Path\midRest]}). 
\]
4.5.2 Soundness, Completeness and Correctness.

We now show that the program, call it \( P \), of this basic version of the theorem prover for LR is sound, complete, and correct. To do so, we assume the results about the logic in [TMM88], and we rely on the explanations given in the preceding section as well as on Prolog and its non deterministic inference engine which is assumed to be sound and complete. With respect to the implementation, we keep in mind that, as we said already, “it is the programmer’s responsibility to make his program sound and complete”, that is, the implementation should be fair and faithful, no infinite loop should occur, and the program should terminate with success or failure as expected. Since the filters do not play any essential role in the proof theory itself we do not take them into account in what follows.

Again, we rely on the theoretical model of Prolog as explained in Chapter 3.

The program \( P \) is made of a finite set of clauses or rules. A computation of \( P \) finds an instance of a given query logically deducible from \( P \). A goal \( G \) is deducible from \( P \) if there is some instance \( A \) of \( G \), where \( A = B_1, \ldots, B_n \), \( n \geq 0 \), is a ground instance (i.e. with all variables instantiated) of a clause in \( P \), and the \( B_i \) are deducible from \( P \).

**Soundness.**

Let \( P \) be the program, \( X \) be a formula, and \( provable(X,Y,Z) \) be a query, i.e. a question about \( X \). Then, we have to show that any formula provable by \( P \) is valid in the logic.

**Theorem 4.4** If \( X \) is provable in \( P \), then \( X \) is provable in \( L_s \).

**Proof:** \( provable(X,Tree,Depth) \) asks if the goal succeeds, that is, if it is true. This amounts to ask if the query is a logical consequence of the program, and logical consequence is obtained by application of the rules of the program, i.e. all instances of the goal

\[
\text{stronglycontained}(X, \text{Up} \text{ Path}), !  \\
; \text{curryContained}(X, \text{Rest}).
\]

\[
\text{stronglycontained}([], []).  \\
\text{stronglycontained}([\text{Head} | \text{Tail}], [\text{Head} | \text{Tail1}]) :-  \\
\text{stronglycontained}(\text{Tail}, \text{Tail1})  \\
; \text{stronglycontained}(\text{Tail}, [\text{Head} | \text{Tail1}]).
\]

\[
\text{member}(X, [X \_]).  \\
\text{member}(X, [_ \_T]) :- \text{member}(X, T).
\]
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

**provable**($X$,Tree,Depth). If they succeed, the goal succeeds. Since these instances are the rules of $L_5$, if there is a proof of $X$ in $P$, there must be an $L_5$ proof of $X$.

If one instance succeeds by unifying with **proved**($X$,~($X$)), then $X$ is an instance of an axiom and is proved.

The rest of the proof is by induction on the length of a proof of $X$, that is, by induction on the application of the rules **apply_rule/3** in the proof search tree. We do not consider each particular rule.

Each application of a rule is a step of the proof until **proved**($X$,~($X$)) is reached. Let $X_1, \ldots, X_n$ be the premises of some proof of $X$ obtained by application of the rules. Assume that $X$ is proved at some arbitrary node different from a tip of the tree, then, recursively, there is a proof of the $X_i$ at the nodes of $X$’s daughters in the proof tree, i.e. the appropriate premises, depending on the form of $X$. On completion, **provable/3** returns the list of each proved premises with the depth of their proof. If all possibilities of application of the rules have failed, each predicate fails, and the goal **provable/3** fails, that is, the formula is not provable.

**Completeness and Correctness.**

Proving correctness and completeness amounts to ask whether the program behaves as intended and terminates with the appropriate answer [Sh86].

**Definition 4.5** The meaning, $M$, of a program $P$ is the set of ground unit goals deducible from $P$.

The intended meaning $M$ of $P$ (i.e. what the program is intended to compute) is defined by induction on the logical deduction steps. Ground instances of facts in $P$ are in $M$. A ground goal $G \in M$ if there is some ground instance $A \rightarrow B_1, \ldots, B_n$ of a rule in $P$ such that $B_1, \ldots, B_n \in M$, i.e. a ground goal is true with respect to $M$ if $G \in M$, otherwise, it is false. That is, the meaning of $P$ is composed of ground facts, and actually, the meaning of the program is the program itself, i.e. the program “means what it says” [Sh86].

A program is **correct** if everything deducible from it is intended, i.e. “the program says what it means”. That is, $P$ is correct with respect to $M$ if $M(P) \subseteq M$.

The program is **complete** with respect to $M$ if $M \subseteq M(P)$, i.e. if everything intended is deducible.

A program is **correct** and **complete** iff $M = M(P)$.

**Theorem 4.6 (Completeness)** $P$ is complete.
Proof: We have to show that $X$ is provable in a proof search tree at some depth $n$, showing that $\text{provable}(X, \text{Tree, Depth})$ is deducible from $P$ by giving an explicit derivation based on the explanations given in the preceding section.

Either $X$ is provable because it is an axiom and, in this case, it unifies with the fact $\text{proved}(X, \sim(X))$, or $X$ is provable by application of the rules of the proof theory at some depth $n$. By soundness theorem, if there is an $L_5$ proof or a refutation of $X$, there is a proof or refutation of $X$ at some depth $n$.

By decidability of LR, there is a derivation of $X$ which is irreduntant. By induction, for all $X$, there is an irreduntant derivation by the program, since, at each step, i.e. at each inductive step of rule application, the Curry property is tested.

Theorem 4.7 (Correctness) $P$ is correct.

Proof: Suppose that $\text{provable}(X, \text{Tree, Depth})$ is deducible from $P$ in $n$ steps. We have to prove that it is in the intended meaning $M$ of $P$ by induction on $n$.

If $n = 1$, then $\text{provable}(X, \text{Tree, 1}) \in M$, $X$ unifies with $\text{proved}(X, \sim(X))$ and is an axiom as intended.

If $n > 1$, then $\text{provable}(X, \text{Tree, n})$ is deducible from $P$ in $n$ steps by assumption. $\text{provable}(X, \text{Tree, Depth})$ is in the intended meaning of the program, i.e. $X$ is proved or refuted at some depth following the applications of $\text{apply\_rule}(X, \text{Tree, Depth})$, and for each rule, there is one inductive step.

The selection/3 rules recursively select all possible candidates principal in $X$. If principal is of the form

- $A + B$, or $A& B$, there is a unique premise in the case of fission and two premises in the case of conjunction. In both cases, the rules succeed or fail and there is no backtracking.

- If principal is of the form $A \vee B$, if $A \vee B$ is already in the parameters, there are two premise sets for $X$, otherwise, each premise for $A$ and $B$ can be contracted with a copy of principal giving two additionnal premises. On failure, another principal constituent for $X$ is selected.

- In the case of $A \circ B$, the parametric constituents are partitionned as shown in the preceding section. On failure, some other partition is selected and all possibilities of candidates premises are explored by backtracking insuring that all conditions of the rules are fulfilled. On failure of the rule, another candidate principal is selected.

Recursively applying the rules, each premise is eventually proved at some depth $n - 1$. Then, $\text{Provable}(X_1, \text{Tree, n - 1}) \in M$. Hence, $\text{Provable}(X, \text{Tree, n > 1}) \in M$. 

Theorem 4.8 (Termination) $P$ terminates on each computation.

Proof: If $X$ is an axiom. Then by unification with $\text{proved}(X, \sim(X))$, the program terminates.

If $X$ is not an axiom and is of the form $A \ast B$, where $\ast$ is any operator in the set {$+, \&$, $\lor$, $\cdot$}, then all possibilities of selection of principal and parameters are exhausted appropriately. In the case of an invertible formula, the selection is irrevocable and ends up in success or failure without backtracking. In the case of a non-invertible formula, when all possible selections have been exhausted without success, the predicate fails and returns control to $\text{provable}/3$ which must fail, the "!" in $\text{proved}/1$ preventing further backtracking.

When all partitions of parameters have been exhausted in the fusion rule, the predicate $\text{partition}/3$ fails and the rule fails as in the preceding case. Finally, the test for the Curry property prevents the generation of an infinite branch.

The only places where the program could enter an infinite loop are in the execution of the "external" predicates, $\text{append}/3$, $\text{member}/2$, $\text{strongly.contained}/2$, $\text{difference}/3$ and $\text{delete}/3$. The classical example of an infinite loop in Prolog is with $\text{append}/3$ where the empty list condition comes second in the definition of the rule given in Chapter 3. As we said, "it is the responsibility...", and these predicates are actually safely coded.

It is not too difficult to show on a simple program that these conditions are fulfilled. But termination property does not guarantee that there will be a proof or refutation in a reasonable amount of time. And the proofs will be more delicate as soon as we will try to make the program more efficient by adding further predicates to incorporate heuristic information into the program, or to process the information dynamically generated at runtime. We will briefly come back to this problem in the next section. From now on, we will assume that the program terminates with a proof or refutation, irrespective of the secular "bug" or hardware limitations which will be noted in due course.

4.6 Improving the Theorem Prover.

The program we have studied in the preceding section is just a backbone providing everything required of an automated theorem prover for LR. But of course, as it is, it is very inefficient, and not very powerful as can be seen from the results in Chapter 6.

The efficiency of the theorem prover can be improved at the logical level and at the implementation level.

First, as far as the logic is concerned, we said in section 4.4.3 that several properties of the logic can be implemented in the program to enhance its efficiency. We will briefly review the preprocessing step, the use of heuristics and admissible rules and the use of filters.
4.6.1 Preprocessing.

In the preceding section, we did not consider the preprocessing and the ordering of the multisets, even though the effective application of the selection/3 predicates may depend essentially on the ordering. Non-ordered data structures do not change the properties of the program, they only make it less efficient. Preprocessing a formula essentially involves its normal forming and its pre-analysis which, through a dynamic analysis of the formula, collects or constructs information and data which will be required for an efficient execution of the proof.

Translation into Equivalent Formulae.

An input formula to test is first translated into its unique equivalent simplified negation normal form ([TMM88], lemma 2.37), the implication and bi-implication operators are translated into their equivalent fission and fusion forms, and negation has only atomic elements in its scope. Whenever possible, the formula is simplified, and the operators are transformed into n-ary operations having multisets as arguments, making it easier to prove. This translation procedure takes advantage of the properties of associativity and commutativity of all the operators, as well as of the absorption and idempotence properties of the extensional operators & and V as explained in [TMM88, 2.2.1; 2.3.1]. We just illustrate this procedure with a few examples. Consider the translation of “prefixing”: 

\[ a \rightarrow b \rightarrow .c \rightarrow a \rightarrow .c \rightarrow b, \]

\[ \text{I} \text{- translate}(\Rightarrow(\Rightarrow(a,b),\Rightarrow(\Rightarrow(c,a),\Rightarrow(c,b))), X). \]

The program returns

\[ X = +o(\neg(b),a), o(\neg(a),c), \neg(c), b), \]

that is, recombined with the operators in binary form, \((\neg b \circ a) + (\neg a \circ c) + (\neg c \circ b))\).

The negation of “prefixing” translates into \((\neg c + a) \circ (\neg a + b) \circ (\neg b \circ c)).

Finally, based on the properties of & and V, absorption and idempotence properties allow to reduce appropriate formulae into their equivalent form

Absorption: \((a \lor b) \land (c \land a)) \Leftrightarrow (a \land c) \Leftrightarrow (a \land c) \lor (a \land b \land c).

Idempotence: \((a \land c) \land (a \land c)) \Leftrightarrow (a \land c). \]
Pre-analysis.

After translation and normal forming, an analysis tree of the formula is constructed by make_tree/10, shown below in a rather compact form, which parses the formula and returns an analysis tree in the form of a Prolog term representing the tree or in the form of a list.

The tree is constructed using a parser based on the built-in predicates functor/3 and arg/3, the second returning the $n$th argument of the functor. Since we only deal with binary operators, the parsing is very easy, each argument being recursively analyzed until the variables inside the complex parts of the arguments are reached.

make_list recursively constructs the lists of positive and negative parts, that is, the set of literals, of each left and right subformulae LeftSub and RightSub. The necessary information is obtained when the parsing reaches the variables. These variables are declared as facts, in the form of a structure containing information relevant to the processing: the variable itself, its arbitrary (for ordering reasons) relative cost, its order, its positive/negative part, i.e. the variable itself. For example,

```
variable('a',1.20,5,'a',[[]]).
neg_variable('~a',3.05,11,[],'a').
```

The cost and the order of each subformula depend on their main operator, Rootop. They are computed and assigned in the structure representing the formula.

A predicate, find_axiom/10, checks whether there is any possibility of finding an axiom $[X,\sim X]$ inside a subformula. An axiom is found by first checking that the operator is fission, $+$, then intersecting the positive and negative parts, which returns $[+X,\sim X]$ or $[]$ if no axiom is found. This information will later be used in the application of the $K_\circ$ rule.

all_operators/2 constructs the multiset of all the operators in $X$. Finally, the tree corresponding to the parsed formula is built by collect_all/2. Each subformula of $X$ is now represented as a term, All, consisting, in that order, of its main operator, the operators of its subformulae if it is a complex subformula, and its own operator otherwise, the formula itself, its level in the tree (required for pretty printing of the tree), its cost, order, positive and negative parts, and the axioms contained in it, if any.

make_tree(X,Rootop,Operators,Level,Cost,Order,PosPart,NegParts,All,Axiom) :-
    ( variable(X,Cost,Order,PP,NP),
      All = ([],[],(X),Level,Cost,Order,[PP],[[]])
    ; ( neg_variable(X,Cost,Order,PP,NP),
        All = ([],[],(X),Level,Cost,Order,[],[NP],[[]])
    )
    ; functor(X,Rootop,Args),
      Nextlevel is Level + 1,
      arg(1,X,Leftsub),
      arg(2,X,Rightsub),
      make_tree(Leftsub,Lop,LOp,Nextlevel,Lcost,Lorder,LPP,LNP,All1,_),
    make_tree(Rightsub,Rightop,ROp,Nextlevel,RCost,ROrder,RPP,RNP,All2,_),
    all_operators(Operators),
    all_operators(Adom),
    add_operators(Operators,Adom,Pl),
    add_operators(Adom,Operators,Pl),
    collect_all(Leftsub,Leftop,LOp,Nextlevel,LCost,LOrder,LPPl,LPNP,All1,Pl),
    collect_all(Rightsub,Rightop,ROp,Nextlevel,RCost,ROrder,RPP,RNP,All2,Pl),
    add_operators(Pl,Adom,Am)
).
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

make_tree(Rightsub,Rop,ROp,Nextlevel,Rcost,Rorder,RPP,RNP,All2,_,)
make_List(Lsub,Rsub,LOp,ROp,Lpp,Lnp,Rnp,PP,NP),
( ( Rootop = '#', ( Lcost > Rcost -> Cost is Lcost
        ; Cost is Rcost ), Order = 1 )
    ; ( Rootop = '&', ( Lcost < Rcost -> Cost is Lcost
        ; Cost is Rcost ), Order = 2 )
    ; ( Rootop = 'v' -> Cost is (Lcost + Rcost), Order = 3 )
    ; ( Rootop = 'o' -> Cost is (Lcost * Rcost), Order = 4 ) ),
find_axiorn(Lsub,Rsub,Rootop,Lpp,Lnp,Rpp,Rnp,PP,NP,Axiorn),
all_operators(X,Allop),
All = (Rootop,Allop,X,Level,Cost,Order,PP,NP,[Ax]),
collect_all(All, Tree).

In list form, the analysis tree of the parsed formula, here our earlier "prefixing" example, appears as follows:

[ (+,[+,*],+,+),+(o(−(b),a),+(o(−(a),c),+(−(c),b)))),0,3,1,[a,b,c],[a,b,c],
  [(+,[a,b,c],[a,b,c])],
  (+,[+,*],+(o(−(a),c),+(−(c),b))),1,3,1,[b,c],[a,c],[(+,[c],[c])]),
  (+,[+],[+],[c],[c]),
  (□,□,−(c),3,1.27,17.8,□,[c],[□]),
  (□,□,o(−(a),c),2,3.05,4,[c],[a],[□]),
  (□,□,c,3,1.22,16.5,[c],[□],[□]),
  (□,□,−(a),3,1.25,17.4,□,[a],[□]),
  (□,□,o(−(b),a),1,3.024,4,[a],[b],[□]),
  (□,□,a,2,1.2,16.1,[a],[□],[□]),
  (□,□,−(b),2,1.26,17.6,□,[b],[□]) ]

Ordering and Selection.

Subformulae can be ordered in two different ways. The easiest way consists in relying on the order of the operators, fission and conjunction coming first, insuring that the invertible rules will be applied first. Nevertheless, it may happen that a complex subformula having one of these operators as main operator contains subformulae which are costly to prove, i.e. contains disjunction or fusion subformulae. It is then best to avoid selecting these candidates as principal constituents first. Hence, when selecting a principal constituent, the selection function first chooses the cheapest subformula (which may not be a fission or conjunction) as principal. To achieve this, in the execution of the select/3 predicate, the generators of the multiset are ordered by cost, the cheapest coming first.
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

Actually, there is no general rule to prefer one or the other ordering, except that the invertible rules should be applied first for efficiency reasons. So, in general, as noted in [TMM88], the ordering should be based on the operators order.

But in our investigations, we noted that there are instances where reversed cost ordering should be preferred because selecting a complex candidate principal constituent first may provide the opportunity to decompose it into simpler constituents earlier in the proof, and make the proof easier in the long run. It appears that selecting the difficult cases first, and as soon as possible in the proof, may, in some cases, be a sound, but not necessarily generalizable, principle. An empirical demonstration that this good old principle can find an application here is given in chapter 6.

The best approach, which is not implemented at this stage, would be to assess the cost of any complexity increasing selection, for example, by checking the unsuccessful deepening of the search tree (the length of the branch created or the number of new nodes opened), and put a bound triggering a long jump back to the original bad selection which would be discarded. P. Thistlewaite’s Pascal version of the theorem prover contains such a scanning procedure which singles out recalcitrant nodes, but which does not seem to be sufficiently strict in its application.

On empirical evidence, we would suggest that the original ordering by operators, fission and conjunction coming first, should be maintained, but any selection of principal constituents which contain only literals should be avoided if their appropriate complement does not figure explicitly in the parameters.1

Processing of Multisets.

At runtime, in the application of the rules, new multisets are created. The example given above represents the analysis tree of the input formula. At each selection of constituents or partition of parameters in the execution of the proof, the newly created data has to be processed. For example, in the process of the proof of “prefixing”, the following selection and partition occurs:

Principal: o(“(a),c) Parameters: [o(“(b),a),b,”(c)]
Apply fusion: left constituent: “(a),
parameters partition: Left Parameters: [o(“(b),a),b],
to prove: [o(“(b),a),b,”(a)]

The subtree, i.e. the data-structure corresponding to each elements of this new multiset [o(\(\omega\)(b),a),b,\(\omega\)(a)], is first retrieved from the main analysis tree by a make_subtree/11 predicate which grabs the following information from Tree.

Subtree:
\([o,[o],o(\(\omega\)(b),a),1,3.024,4,[a],[b],[[]]),([[]],[],,b,3,1.21,16.3,[b],[[]]),]

1. See addenda, Note 8.
and forms a new term consisting of the multiset, its subtree, main operator, all positive and negative parts, atomic positive \((b)\) and negative \((\sim(a))\) parts, positive \((a)\) and negative \((\sim(b))\) parts of complex subformulae that is passed to a `prove` procedure. Note that the original analysis tree `Tree`, which actually, in memory is a pointer, must be an argument of all the predicates implementing the proof theory in order to be readily available every time new data has to be processed.

```prolog
make_subtree(X,Tree,Subtree,Op,PP,NP,Pat,Nat,Pcornp,Ncornp,Axiorn)
returns (with `Tree` and `Subtree` as above),
[ [o(-((b),a),b,(-ca)), Tree, Subtree, [o], [a,b], [a,b], [b], [a], [b], [a,b]]
```

The list `Op` allows to check that there are complex subformulae and fill the slots reserved for their positive and negative parts, `Pcomp` and `Ncomp`. Again, this is a rather inexpensive operation.

Moreover, at this stage, we could already stop the process of the proof of this candidate, knowing that this multiset is provable. Indeed, the intersection of its positive and negative parts shows that they are equal as multisets, and since there is a single fusion operator, as we will see, this multiset is actually a derived axiom.

Lists as Data Structure.

One could think that working with lists rather than trees, as we do here, is computationally expensive. It is true in general. Nevertheless, the preanalysis tree, which is readily accessible at any stage of the computation, makes the creation of new multisets relatively cheap. Indeed, if we had to reorder the complete lists every time, this would be expensive. In general, any good algorithm to sort a list has a running time of \(O(n \log n)\), where \(n\) is the length of the list. But we are always dealing with ordered lists, and the constitution of a new multiset by insertion or deletion amounts to inserting some element in these pre-ordered lists, and this does not take much more than \(O(n)\), the time complexity of inserting and deleting with hashing. So, we may think that working with lists is not very much less efficient here than working with balanced trees. Moreover, the lists are often short, and some Prolog programming techniques like difference lists, or tail recursion allow to perform this operation efficiently. Some other techniques like unification with "canned" lists, i.e. if one wants the sub-list \([a, b, c]\) from the list \([a, b, c, ..., n]\), unify the list with \([X, Y, Z|..]\), still improve the efficiency, if not the elegance of the program. Finally, all operations on lists of elements are performed on their corresponding subtree or informational structure at the same time, preventing duplication of the processing. That is, if any
element is an argument of some operation, its corresponding subtree is also an argument undertaking the same process. And moreover, any operations on sets of elements parallelize well and efficiently [He90],[Di90]. Actually, it is on these operations that the best parallel speedup results were obtained.

Filters.

Several provable properties of provable multisets can be used to test the multisets candidates for a proof. These properties are used as filters which allow to efficiently discriminate between provable and non-provable multisets. And their implementation and execution is relatively cheap -except for the model property test- since a pre-processed structure is passed to filtering/10. On failure of this predicate, the system backtracks and tries an alternative application of the rules.

\[
\text{filtering}(X, \text{Subtree}, \text{Op}, \text{PP}, \text{NP}, \text{Pat}, \text{Nat}, \text{Pcomp}, \text{Ncomp}, \text{Axiom}) :- \\
( ( ( \text{member}(v, \text{Op}), \neg, \text{Axiom} \not= []) \\
\text{PP} = \text{NP}), \\
\text{subSet}(\text{Pat}, \text{Ncomp}), \text{subSet}(\text{Nat}, \text{Pcomp}), \\
( ( \text{not}(\text{member}(o, \text{Op})), \neg, \\
\text{length}(X, N), N < 2 ) \\
\text{true}), \\
\text{msetFormula}(X, '+', F), \\
\text{not}(\text{counterModel}(F)), \neg )).
\]

The effect of the various filters is summarized in the results presented in chapter six. Here, we will just define them. As can be seen from the definition of filtering/10, their implementation is very simple.

- A multiset has the strong positive-negative parts property iff each propositional variable \( p \) is a member of the set of negative parts iff it is a member of the set of positive parts.

A multiset has the weak positive-negative parts property iff the intersection of positive and negative parts is not empty.

By [TMM88, theorem 2.43], if a multiset \( \alpha \) contains no disjunction, then it is provable only if it has the strong positive-negative parts property. Otherwise, if it contains a disjunction, it is provable only if it has the weak positive-negative parts property.

The implementation of this filter is easy. First, check whether \( \vee \) is a member of the set of operators. Since the sub-list \text{Axiom} was obtained by intersection of the
positive and negative parts, we have only to check that it is not an empty list, the other checks are straightforward. One or the other property must be fulfilled for \texttt{filtering/10} to succeed up to that point.

- A multiset has the \textit{strict positive-negative parts} property iff the set of positive/negative atomic parts is contained into the set of negative/positive complex parts.

  A multiset $\alpha$ has the strict $+/-$ parts property $\Leftrightarrow$ \cite{TMM88, theorem 2.45].

  Again, the test is easy, the conditions are tested by \texttt{subSet/2}, a generalization of \texttt{member/2}.

- A multiset $\alpha$ has the \textit{rule-of-2} property iff it does not contain a fusion operator. Then, $\alpha$ is provable only if $\text{card}(\alpha) \leq 2$. If $\alpha$ is provable, then it has the property. \cite{TMM88, theorem 2.47].

  This property is easily implemented using the \texttt{length/2} built-in predicate which returns the length of a list. If there is a fusion in the list of operators, the property does not apply, and the execution goes on. This is effected by declaring the alternative \texttt{true}, which always succeeds.

- The last filter, the model or matrix property is applied to the multiset recombined into a fission formula. The explanation of this property and test is delayed until chapter five.

The $K_0$ Rule.

This invertible rule, proved admissible by R. Meyer, resembles the rule of weakening, but it allows to reduce a copy of the constituents of a multiset \textit{only} in the presence of an axiom. If it has the opportunity to apply, it is a very effective and powerful idempotents reductor allowing to weaken a copy of $A$ in the premise.

$$\vdash \alpha, A, B, \sim B \quad K_0$$

$$\vdash \alpha, A, A, B, \sim B$$

The rule is implemented to operate at the time of the proof of the multiset. The procedure \texttt{prove/12} is defined in such a way that it can succeed in three different ways: first, a check is made to insure that the multiset was not already proved earlier. Then, if it is not the case, the $K_0$ rule can be applied if the sub-list \texttt{axiom} is not empty. If the rule applies, the reduced multiset is eventually filtered and, if it passes the filters, or if it does not require filtering, the execution proceeds with the reduced multiset. Finally, if the $K_0$ rule does not apply, the execution proceeds, depending on the rule application, a test against the filters is eventually made, and eventually passing the multiset to \texttt{provable/6}. 

CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

An additional test for \( K_0 \) application should also be performed earlier, in the application of the fusion rules, after the partition of parametric constituents. Suppose that one of the left or right or both constituents of principal is a literal. Then, it may happen that several parameters are duals or complements of this literal, triggering the application of the rule. Consequently, for all possible combinations of the constituents with partitions of parameters containing a duplicate which would be erased by the application of the rule, one single combination should be tested. This situation often occurs in hard cases, where several hundreds, sometimes thousands, of combinations with duplicated parameters have to be tested. At the moment this test is not implemented in the theorem prover. The main reason is that there are already so many other tests performed as heuristics in the body of the fusion rules, that, adding more control, creates real problems for backtracking. But it is not impossible to implement the test externally and pre-process the partition of parameters before starting the execution of the rule.

Derived Axioms.

A multiset of the form \([A, dual(A)]\), where \(dual(A) = normal\ form(\neg A)\) is a derived axiom. When encountering an instance of an axiom, like \([\neg A \circ \neg A, A, A]\), the branch can be pruned. In order to improve the performances of the theorem prover, a small database of derived axioms is used. Actually, in order to simplify the coding, this database contains more than derived axioms. These are included in a database of simple theorems which play an important role in many proofs and are heavily used in the heuristics as the next section explains. Since a part of chapter five is devoted to the use of databases we will not develop this topic here.

4.6.2 Heuristics.

In order to improve the efficiency of the theorem prover, several heuristics have been implemented. Basically, the heuristics rely on the use of rather small databases of elements, or facts, called duals and complements. These elements are derived axioms and theorems which keep reappearing in the process of the proof of many theorems.

An advantage in using Prolog to access the databases of derived axioms or proved multisets is that retrieval is performed by unification and matching. The data, represented using Prolog variables which unify with any appropriate propositional variables, are encoded as schemata rather than instantiated provable formulae, i.e. a theorem \((a + a) + (\neg a \circ \neg a)\) is encoded as \((X + X) + (\neg X \circ X)\). This amounts to a form of equational axiom where all equivalents are recognized. This encoding is close to the taxonomic representation of knowledge in the framework of knowledge representation theories. A theorem is some
parenthesized structure ordered by operators, where anonymous variables are place-holders for appropriate propositional variables. For example,

\[
\text{dualcomp}(o(X,Y), \neg(X), \neg(Y)).
\]
\[
\text{dualcomp}(o(\neg(X),X), \neg(X)).
\]
\[
\text{dualcomp}(o(v(\neg(Y),Y),X), \neg(X)).
\]
\[
\text{dualcomp}(v(o(v(\neg(Y),Z),v(\neg(Z),Y)),o(\neg(X),Y)),X), [Z, \neg(Y)]).
\]

We briefly review the main heuristics used. They only concern the application of the fusion rules.

\[
A \circ B,
\]

i) A first test is performed on the left constituent of a fusion formula to see if it has been proved already. Since, given the distribution of partitions of parameters, this possibility of \(A\) being proved without parameters occurs last, if it is already proved, we have only to check that the right constituent, \(B\), is proved with all the parameters. Actually, this situation does not occur very often, but in some cases, when the constituent is an axiom, a derived axiom or a theorem, this test puts the burden of a proof with all parameters on \(B\). If it happens that \([B+all\ parameters]\) passes the filters, then we may well have created a rather complex multiset possibly more difficult to prove than the original multiset given a more balanced repartition of parameters.

If this first test fails, then a systematic heuristic search can be carried out

ii) Two predicates \textbf{selection\_left} and \textbf{selection\_right} are used to try to find the best selection of parameters, that is the selection for which there is a proof of the candidate, or if there is no such immediate proof, the selection most likely to provide a short proof.

First, all left and right partitions of parameters are generated using the \texttt{findall/3} built-in predicate which returns the list of all solutions of some predicate in its arguments. That is, we create the list of all possible left and right partitions.

The \textbf{selection\_left} predicate calls first a \textbf{best\_selection} predicate which dynamically analyzes the available information and tries to find a best match between the constituents of the fusion formula and their possible legal assignement of parameters.

Several situations can arise:

- there is a winning parameter assignement of \(A\) which matches its complementary assignement for \(B\), i.e. for which \(A\) and \(B\) are proved. Then \(A \circ B\) is proved.
• If such assignment is not found, the analysis goes on to find a *best* selection. That is, the program tries to find in the list of possible parameters the partition with which \( A \) is proved. This is done by consulting the small databases of theorems and derived axioms. Then it tests whether this selection for \( A \) would agree with some legal partition with which \( B \) is proved.

• If none of these situations occur, the last solution is to pick the *most likely* complement which can lead to a proof of either \( A \) or \( B \). This is done by analyzing the formula. (i) If \( A \) and \( B \) are variables, then a complement is selected which contains the dual variables. (ii) If \( A \) and \( B \) are complex, a best match between \( \text{Pcomp} \) and \( \text{Ncomp} \) is selected.

Specifically, if \( A \) and \( B \) are complex, the program will try to find in the data-structure of all elements of the list of all possible partitions the best match between the atomic parts \( \text{Pcomp} \) and \( \text{Ncomp} \) of \( A \) and \( B \) and the complementary parts of any element of the list.

iii) These heuristics were also used with much larger databases of theorems, but, as we will see in the next chapter, the amount of information to process is by far too large. Consequently, no heuristic search of best parameters choices is performed on these databases. Only a less expensive test on the large databases, *subsumption*, is performed, and it concerns all candidates to a proof.

A procedure \( \text{already_proved}(X) \) checks whether \( X \) has already been proved during the execution of the proof. If it has not been proved, then it checks whether \( X \) is in one of the large databases. If it is not, then \( X \) is decomposed into its head and parts, and a \( \text{smart_check}(\text{Head, Parameters, Databases}) \) tries to find the elements of the databases matching \( \text{Head} \), i.e. any constituent of elements of the databases matching the head. When a match is found, a test of strong containment is performed on the remaining constituents of this particular element, i.e. a test whether \( X \) is strongly contained in that element.

iv) If none of these heuristics work, a last test is made on a file of pre-processed possible combinations to find an eventual partition which would pass the filters. Otherwise, the usual execution procedure by exhaustion proceeds, all possible combinations of \( A \) and \( B \) with parameters being tried.

The implementation of these heuristics requires a lot of coding and adds choices and controls which impose severe constraints on the execution of the program. These could put the completeness of the program in jeopardy, but careful checks make sure that it is not the case. Indeed, if none of the heuristics apply, these procedures constitute only one
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

condition of success of the rules. On failure, control is returned to the non-deterministic all solution execution of the program which takes over the rest of the execution of the procedure.

A second problem is that these heuristics affect the execution time of the program. Indeed, they require a very large amount of processing which slows down the execution. But if a winning choice or a best choice is found, then the overhead created by the search for the right parameters is compensated by the early pruning of the current branch which is closed immediately or a few steps later. And actually, it seems that no proof of many hard cases can be found without some sort of heuristic selection of principal and parameters.

Finally, note that, as suggested by P. Thistlewaite, the possible partitions of left and right constituents of fusion subformulae could be precomputed and tested against the filters before starting the execution of a proof. But this is a very expensive process. This has rather suggested to rely on an additional specific database of pre-processed possible combinations of principal and parameters. As we will see in chapter six, that database has been generated by systematic processing of entire sets of test formulae.

4.6.3 Remembering the Information.

We have already said that one cause of Prolog’s inefficiency is that it forgets what it has already computed, i.e. all its previous instantiations. On the other side, one cause of inefficiency of the theorem prover itself is that it has to compute and to test many times the same information. Hence an improvement of the efficiency of both would be to insure that they remember all computations already performed.

This seems to be an essential efficiency condition to impose on any program. Without it, there would not be much point in trying to speed them up using the resources of parallel processing, for example. Actually, there is no maximally parallel evaluation method for logic programming without some device to remember information [Ra89], or some evaluation strategy which avoids redundant computation [Ci83].

As we emphasized in the preceding chapter, the similarity between Prolog and Automated Theorem Proving suggests that the methods used to improve the performances of one could be used to improve the performances of the other.

Lemmas, Tabulation and Memoing.

In Prolog, a built in predicate “assert/1” allows to add facts or information to the internal database at runtime. The corresponding “retract/1” allows to erase some facts from the database. This is an interesting feature of the language as long as it is used safely, that is as long as it does not change the semantics of the program. In this respect, this facility is
not harmless, and this may explain why some Prolog implementations do not provide it.

Since the early days of Prolog, assertion of information obtained during execution of a program has been suggested as a way to improve the efficiency of the language through the use of lemmas,\cite{Lo69, Ko79}, memo-functions in functional languages \cite{Mi68}, extension tables \cite{Di87} or tabulation \cite{Bi80}.

Implementing these techniques requires that all new goals be identified at runtime as having been solved already, and that the information necessary for the remaining execution of the program be available, as well as all generated solutions.

It should be noted that these techniques preserve completeness of the program, and in many instances are a *sine qua non* condition of termination of a program.

Memo-ization or *memoing*, as this technique is sometimes called, is an optimization technique used in dynamic programming which consists in storing all results or intermediate results in a table, or in any other appropriate structure, to avoid redundant computations and the inefficiencies in evaluating recursively defined functions. In Prolog, this amounts to keeping a copy of all goals and facts generated at runtime. Every time a function $f$, or a goal, has to be evaluated for a given argument $x$, a check is made to determine whether $f(x)$, or the corresponding fact, has been previously computed

\begin{verbatim}
function (memo)
    lookup(table)
    return (memo)
else
    compute (memo)
    save_in(table)
\end{verbatim}

The classical example of a function requiring such storage of intermediary results to prevent multiple repetition of the same computation is the function $fib(x)$ which computes the Fibonacci numbers and which has a running time of $O(\phi^x)$, where $\phi = 1.618\ldots$. For example, $fib(x)$ requires two evaluations of $fib(x - 2)$, three evaluations of $fib(x - 3)$ and so on.

This function is typically exponential, but with an efficient search strategy and memo functions, it is linear. Thus, \cite{Bi80} shows that any recursive function which can be evaluated in a time linearly related to the number of subsidiary function calls can always be transformed into an equivalent program with an $O(n)$ running time, where $n$ is the number of distinct arguments for which the function value is required.

In our approach to Automated Theorem Proving in Prolog, we rely heavily on the use of databases or knowledge bases which contain all collected information which could enhance the efficiency of the theorem prover. And, as we already emphasized, this approach imposed itself upon us.

One way of viewing the databases of theorems and of non-theorems is to consider them as
repositories of permissible inference steps in a proof search tree, or inference steps useless to try.

In the current implementation, these large databases are mainly used for look-up, testing that some multiset has been proved or refuted, and subsumption test.

In an earlier implementation, a technique similar to some suggestion found in the field of deductive databases called counting was used. Basically, it consists in computing indices for each fact deduced that indicate the relevance of this information in the deduction, allowing to skip some literals in the rules and arguments in the predicates definitions of the program [Ba86].

Our implementation of this technique was restricted to a labeling of the elements of the databases. That is, the databases also contained information relative to the proof step which had generated the data, at each application of a selection function. The principal and parametric constituents were recorded along with a label indicating the level at which the selection was performed in the tree, and each data generated downward was recorded with the label of its ancestor. This amounted to a kind of implicit forward chaining strategy strongly determined by the data deducible from the initial formula. That is, at each new inference step, i.e. each new application of a rule, we could safely skip all the intermediary steps already computed and proceed from the data recorded in the last place. This also amounted to some sort of inheritance strategy in which the information collected during execution of a goal is inherited by all its descendants.

This technique did not allow to perform any dynamic program transformation of the theorem prover as in [Ba86], i.e. calling alternative predicates or rules for execution, given the available information. First, the technique (asserting and retracting specific rules) seems to work well for small programs and small amount of information. Secondly, proving theorems is different from databases processing. In a theorem prover we have to deal with dynamic information, i.e. arbitrary and variable information generated by the application of the rules. Finally, the inefficiency of the theorem prover at that stage made this approach unworkable. Strictly speaking, all goals are descendants of the original goal at the first node of the tree. The memoing procedure with labelling had thus to be restricted. In any case, it soon appeared that keeping plain databases of proved and refuted multisets was sufficient to perform the same sort of optimization. If we consider the elements of the databases as proofs or refutations schematas, they represent all the proof theoretical steps required to proof or refute them anyway, and thus, there is no point in storing any additional information.
CHAPTER 4. AUTOMATED THEOREM PROVING IN LR.

There is a way to implement the "intelligent backtracking" procedure referred to in the preceding chapter, but it is expensive, requiring large amounts of space and time. Nevertheless, in the limits of practicality, there is a positive trade-off between this expensive house-keeping and memory management operations and the net gain in efficiency attained. As the results of Chapter six show these techniques may very well make all the difference between a proof and no proof at all.

i) During the execution of a proof, any formula or multiset which is proved or refuted is asserted, and before each attempt to prove any other new candidate formula or multiset, a check of the table is made.

ii) A second use of this Prolog facility is to store every selection of principal and parametric constituents. If all possibilities of finding a proof with some selection, for example, [selection 1] in Figure 1, have been exhausted and have failed, this selection is kept in a table, and any other selection recognized as identical which occurs later on in the proof, [selection i], will cause the failure of the goal and trigger an immediate alternative selection.

By step (1), the information (theorems and non-theorems) gathered during the processing of this selection is kept and remains available during the remainder of the execution of the proof.

1. See addenda, note 9.
iii) If, during the process of the execution of a proof, for some selection, for example, [selection 2], there is one or more partitions of parameters which create a multiset which passes the filters, the selection is recorded along with the partition which allows to pass the filters, as well as with the remaining partitions which have not been tried yet.

This is necessary to insure that all partitions except those which fail will be explored later on in another branch of the tree, [selection n], since these could very well be the condition for the execution of the proof to proceed on that branch. Suppose that completeness is abandoned and not all solutions are tried. Skipping [selection n] altogether, i.e. leaving out some solution when the immediate ancestor of [selection n] was an invertible rule baring backtracking would create an infinite loop.

Storing these partial sets of solutions of a goal is more delicate to implement. First, we have to store the original selection and any further selection, and for each of them, the particular partition which succeeds and those which have not been explored. For each partition which creates a multiset which passes through the filters, the same process applies iteratively. This creates a problem of book-keeping and of updating of arrays (lists of indexed lists) difficult to handle if too many new nodes are created below the original selection.

iv) The current implementation goes around this problem by keeping the first option: (i) storing the selections which fail for all partitions and perform an additional storage operation in the application of the fusion rules. At this level of the rules application, all partitions for all constituents of principal are stored. (ii) If any one of them ever passes the filters, it is retracted, i.e. erased from the table. See Figure 2
This will insure that any further combination of parameters which would fail will never be tried more than once. Those which could succeed are still available for processing. In the figure, one can see that one single storage operation could be sufficient, performing it at once when one partition passes the filters, but this would imply to add some more arguments to the procedure `prove/12` which is already rather heavy.

This use of memoing or storage tables improves the execution time of the theorem prover in a very significant way as long as it has the opportunity to apply. The idea of keeping a copy of all computed results allows to implements exactly to the idea of "intelligent backtracking" in Prolog. In the case of the theorem prover, we can compare the storage of all `failing selections` to "deep backtracking" and the storage of `failing partitions` to "shallow backtracking". But, as with Prolog, severe limitations on the implementation of these techniques are imposed by the availability of working memory space and by the overheads created by the storage and retrieval of information. We will see in the next chapter that the use of massive parallelism prevents these limitations, to some extent.
Chapter 5

Databases and Massive Parallelism.

5.1 Introduction.

To summarize the problems with which we are confronted, we have to deal with the high level of complexity of the logic and to find the most efficient way to implement an automated theorem prover based on Logic Programming which, at the same time, is powerful enough to tackle the hard and complex problems of the logic without being impeded by the specific limitations of the programming language. Actually, if the complexity of a problem is too large, the issue of the choice of language becomes irrelevant. But, we are interested in practical and feasible cases, and in our investigations, a previous efficient implementation by P. Thistlewaite of the "Kripke" theorem prover was extremely useful in showing what the hard problems are, and his theorem prover was used to control the correctness of the output of the Prolog implementation.

In this chapter, we review two complementary ways to tackle these problems which have proved to be successful. First, the obvious necessity of retaining the information generated. To that effect, large knowledge bases are used. Next, the importance of controlling the amount of information, to access it and put it to good use. In this case, we are confronted with the usual bottlenecks of sequential processing. Parallel processing in Prolog has proved to be a first step in solving this problem, but the amount of information being very large, massively parallel processing appeared to be required and allowed to solve, at the same time, the problem of the inefficiency of the programming language and the problem of processing large knowledge bases in parallel.

The amount of generated information is now controlled by storing the databases on a Connection Machine where the subsumption test is performed on all the databases at once and
always in constant time. Doing so, all expensive tasks requiring intensive computation are processed on the appropriate hardware, freeing Prolog from that burden, and leaving it in control of the execution of the proof theory.

5.2 Some Issues in Automated Theorem Proving.

In order to automate logical reasoning, large amounts of knowledge have to be implemented in an automated system. In return, an automated system should be able to discover or help in discovering some general principles of the logic and, possibly, some principle of knowledge representation and automatic inference.

In [Ov89], Overbeek and Wos argue that subsumption is indispensable in any automated reasoning program dealing with deep questions because it is the best way to discard irrelevant or redundant information.

As we already emphasized, the problem of finding a strategy which would prevent a system from computing many times the same results has been central in Logic Programming. Nevertheless, the authors characterize two approaches which either retain, or reject all redundant information. According to them, the former is characteristic of Logic Programming which they consider hopeless and unable to "simply bypass, without cost the need to cope with redundant information" [Ov89,5]. In the approach they advocate for automated theorem proving or automated reasoning, only the relevant information is kept, and so prevents a system from wasting time on irrelevant computation. Their critique actually should not concern us too much because, even though we are working in the framework of Logic Programming, as we said above, our aim is to use Prolog mainly as an inference engine rather than as a theorem prover in se, and, in addition, in our work, we have tried to show that, given the available technology, it is possible to overcome some of the deficiencies of Prolog.

What they consider as a representative of a Logic Programming automated reasoning system is Stickel's PTTP [St88] which is an extension of pure Prolog, and thus suffers from most of the language shortcomings. Another critique is directed toward the tendency in AI and Logic Programming to emulate the way a person reasons in the hope that this may provide some clues or heuristics easy to implement in a "reasoner". According to them, it is best to use these clues to restrict and direct the strategies of a computer oriented "reasoner".

There are many reasons why Logic Programming as paradigmatic of the automation of reasoning is the wrong approach. As long as the reasoning is algorithmic and does not require information, it is effective, but without any specific means of dealing with information it is useless.
The best available way to cope with large amounts of information and to retain only the relevant one is subsumption, which the authors consider as the major contribution of A. Robinson [Ro65] to automated reasoning. According to them, subsumption "is to automated reasoning as the wheel is to transportation" and a substitute for it has still to be found.

After having characterized the flawed folk wisdom of the field, e.g. the growth of the search space has always to be exponential, if no proof is found in a few minutes, none will be found, or the insight gained in investigating simple problems will apply automatically to more complex problems... they ask how to integrate the two disparate styles of deduction, one which retains information, and Logic Programming which does not retain any information.

Elsewhere, [Wo87a,b], L. Wos underlines the main obstacles to the effectiveness of automated reasoning programs. These obstacles are related (i) to the strategy used in the reasoning program and (ii) to the management of its databases.

The strategy may not be well directed, it may lack heuristics to guide the choice of relevant clauses from which to draw conclusions, the size of the deduction steps may be inappropriate and metarules may not be adequate guidelines. Moreover too many unnecessary or redundant deduced clauses may be stored, and the management of an unnecessarily large database requires time and space.

As a research problem [W87b], he asks what strategy can be employed to deter a reasoning program from deducing a clause already retained, or from deducing a clause that is a proper instance of a clause already retained.

Our answer to this question, which at the same time allows us to partly control the complexity of automated theorem proving in LR, has been to use as much knowledge as possible in the form of databases that we will call dynamic if they are produced at runtime while executing a proof, or static if they were generated and updated while experimenting. The later contain all information collected while running the theorem prover and can be compiled with the program.

Figure 2, infra, represents these databases schematically. The static databases are on the left of the figure, the dynamic databases on the right. Since we already explained the use of most of these last databases before, we will only be concerned with the former ones in this chapter.

5.3 Database of Efficient Matrices.

Logical matrices or finite algebraic models have always been important tools in the study of logical systems. It is obviously the case for multivalued logics [Re69], but their use
CHAPTER 5. DATABASES AND MASSIVE PARALLELISM.

is mainly important in the study of structural properties of non-classical systems. For example they were largely used in the study of modal logics before Kripke's semantics, and they allowed to prove the independence of the axioms of intuitionistic logic.

Since the method of matrices plays a central role in relevant logic and in automated theorem provers for these logics, we first outline the essential features of these algebraic models and their role in logic. More information can be found in [Ki77], [Sl80], [Me87], [Ma88] and [TMM88].

5.3.1 Finite Models.

The simplest matrices are the truth tables of classical logic which allow to give an interpretation to the formulae of the language by assigning truth values to atomic formulae and computing the resulting truth values of complex formulae in the tables. If the resulting value is true for some assignment of value to the subformulae, the matrix is a model for the complex formula.

All theorems of classical logic are tautologies, that is, for all assignments of value to atomic formulae, the value of the complex formula is true. In this case, the matrix is not only a model, but a characteristic model of this logic. All logical systems do not enjoy this property.

In Relevant Logic, the matrices used can be seen as generalizations of classical matrices with the addition of ordered values between true and false, some of these values being designated (i.e. true, and noted *d), some being non-designated.

The Lindenbaum algebra of a logical system constitutes a matrix defined by the equivalence classes of formulae of the system seen as elements of the matrix. More simply, matrices are lattices with the usual operations ⊃ and ▽, the greatest lower bound and the least upper bound, and the usual lattices conditions. Figure 1 is an example of a matrix with its negation table and its corresponding lattice.

Given the interpretation of A ⊃ B ∈ D in a lattice as A ⊃ B ∈ D iff A ≤ B, where D is the set of designated values, the axioms of the lattice allow to check which formulae and axioms schemata are valid.

These lattices are generally represented as Hasse diagrams where the ▽ of two elements is
computed downward from these elements to the first \( \cap \), and reciprocally for \( \cup \).

Summarizing these notions in a definition, let \( L \) be our preferred logical language. A **matrix** \( M \) is a pair \( M = (N, D) \), where \( N \) is the set of values, and \( D \) the set of designated values of \( M \), such that for \( d \in N \), \( *d \iff d \in D \).

**Definition 5.1** The *valuation* of formulae of \( L \) in \( M \) are homomorphisms \( h \) from formulae into \( N \), and the assignements are functions from variables into \( N \).

A formula is *satisfied* for a valuation iff, for some assignment \( v(A) \) of values to its variables, \( v(A) \in D \).

A set of formulae \( F \) is satisfied by \( v(F) \) in \( M \) iff \( v(F) \) satisfies every \( A \in F \).

A formula \( A \) is valid in \( M \) iff \( A \) is satisfied for each valuation \( v(A) \in M \).

A matrix \( M \) satisfies \( L \) if every theorem of \( L \) is valid in \( M \), and a matrix \( M \) is characteristic for \( L \) iff the theorems of \( L \) are valid in \( M \).

A *finite characteristic matrix* of \( L \) allows to determine in a finite number of steps if a formula \( A \in L \) is a theorem. That is, the finite characteristic matrix property of \( L \) provides a decision procedure for provability in \( L \).

### 5.3.2 Matrix Testing.

In the execution of a proof by the theorem prover, when constructing the proof search tree, we should search for a proof that has the matrix property, i.e. a proof in which the multiset candidate to a proof has the matrix property. And any multiset which fails to have the property should be discarded.

To do so, all multisets generated by application of the rules in the construction of a proof search tree undergo a test to check if they have the matrix property. If it is the case, the execution of the application of the rules proceeds, otherwise some other rule, or some other possible application of the same rule in the case of fusion, is applied.

In order to test the multisets candidates to a proof against the matrices, the commas separating the constituents of a multiset are reinterpreted as fissions.

Let \( T(\alpha) \), the recombination of a multiset \( \alpha \), be defined by \( T(A) = A \), and \( T([A_1, \ldots, A_n]) = T(A_1) + T([A_2, \ldots, A_n]) \), the translation of the multiset into a fission formula. Then, \( \alpha \) has the matrix property with respect to some set of matrices iff \( T(\alpha) \) is valid in all members of the set.

There are two problems related to the matrix test.

The logic \( LR \) has not the finite characteristic matrix property. That is, there are infinitely many possible matrices or finite models against which to perform the matrix property test. A first problem is then to decide how to select which matrices to use, i.e. to assess how faithfully various \( LR \) matrices model the logic. A second problem is the overhead created
by the evaluation of formulae. Indeed, matrix testing, even though it is a very efficient
filter to refute non-theorems, can be a computationally expensive test. The number of
possible valuations of a formula is $m^n$, where $m$ is the number of values of the matrix and
$n$ the number of variables in the formula evaluated.

Since the matrices are used as filters to refute possible candidates to a proof, that is,
matrix testing is used to prune the proof search tree, it is thus important to select the
most efficient one, i.e. some subset of the set of all LR-matrices which are most likely to
refute most refutable candidates.

In order to partially solve our two problems, the selection of efficient matrices and the
reduction of the evaluation time overheads, some empirical research was carried out to
single out the most efficient matrices up to size ten. In addition a database of prepro­
cessed possible partitions of parameters and principal likely to occur in the application
of the fusion rule was created. All combinations of possible principal constituents with
the possible partitions of parametric constituents of fusion formulae were generated and
tested against the filters. Doing so, we can perform some heuristic selection of the “good”
partitions of parameters in the application of the fusion rules, and this allows to skip the
filter test whenever some appropriate partition is found in the database.

Since we already mentioned this second database earlier, and we will come back to it in
the next chapter, we will not consider it here.

5.3.3 Finding the Right Models.

The implementation of the Kripke theorem prover makes use of two matrices (called
CHAIN and CRYSTAL, due to the form of their Hasse diagram) which on empirical
evidence appear to be particularly powerful. The choice of only two matrices was guided
by efficiency considerations. Earlier versions did use a small set of matrices.

The overhead to test a multiset against a matrix imposes practical constraints which left
the authors with a choice of only a few small matrices even though the more matrices used,
the closer one approximates the language, and larger matrices are in general stronger.

[TMM88] note that if some multiset is not rejected by some model, this model could
provide information to be used to constrain various search options, for example, ruling
out some particular members of the multiset from being candidate principal constituent.
The generation of a database of possible combinations of parameters and principal puts,
in some way, this idea into practice. But considering the investigation of the topic by
these authors, this is still far from what they really intended to do.

Indeed, a formal comparison of finite models is undertaken in an attempt to single out
the most efficient models and which would provide the basis for their selection. But this

1. See addenda, note 10.
is an intractable problem given that, for all \( n > 0 \), there are infinitely many logically non-equivalent \( n \)-variables formulae in \( LR \). And this makes impossible any exhaustive analysis of the members of logically non-equivalent \( n \)-variables \( LR \) non-theorems refuted by the models.

For example, the authors discuss the principles of comparison of the refuting strength of various matrices and show that some a priori founded condition may not be respected in practice. They finally express their concern that "the investigation of subalgebra orderings has not provided a way of assessing whether or not one matrix is indeed stronger at refuting \( LR \) non-theorems than another" (p.80).

This also means that our empirical investigation was necessary and justified until deeper theoretical results are available.

For the record, it should be noted that R.K. Meyer suggested an entirely different way of considering the model testing. The idea can be summarized by simply saying that, under his suggestion, a formula should construct its own refuting matrix. The intuition behind this idea is interesting, but we could not find a way to start an implementation of it, and it is then left aside for future research.

J.Slaney’s program \textit{MaGIC}, Matrix Generator for Implication Connectives, [Sl91], was used to generate the 4533 \( LR \)-matrices up to size \( 10 \times 10 \). Empirical investigation allowed to first single out 12 efficient matrices which all together refuted all the non-theorems generated while developing the theorem prover. At that stage, this represented \( \approx 8565 \) formulae. While building the database of possible “good” partitions, 4 new matrices were discovered whose overall efficiency is high, but which, in practice appeared less efficient than \textit{truth value}, for example. Nevertheless, since they were the only matrices refuting some multisets, they were added to the database in the hope that in the long run they would contribute to the efficiency of the theorem prover.

The same consideration lead to more research which came out with 6 additional size 10 matrices. The theorem prover was used to systematically generate all possible combinations of constituents of multisets candidates to a proof. Any combination which was not refuted, i.e. which passed the filters, including the selection of efficient matrices of course, were tested against all 4533 \( LR \) matrices as well as by “Kripke”. This produced thousands of non provable multisets which were refuted by Kripke and by some matrices. The 6 additional matrices are those which, together, refute all these multisets. One multiset which was refuted by Kripke passed throughout all the matrices. Several others are refuted by only one matrix, others by two etc. Some which can be considered as prototypes of "being false" in \( LR \) are refuted by all the 4533 matrices. Not surprisingly, \( a + a \) is such a formula.

The list of the 22 efficient matrices - with respect to the 3 tests files (Standard, Impset
CHAPTER 5. DATABASES AND MASSIVE PARALLELISM.

and Asset of [TMM88]) and the multisets generated from them - is given in next chapter. Note that, depending on the selection of principal constituent, and we essentially used only two ways to do it, other new multisets could be generated which would require some other matrices to be refuted.

In practice, the theorem prover only uses 15 matrices. Given the refuting strength of CHAIN, for example, most non-theorems are refuted by some of the first matrices in the test. Moreover most candidates contain only two or three different propositional variables. Hence the matrix test is relatively cheap considering the improvement in efficiency. Nevertheless, adding 7 size 10 matrices to the original set increases by 15 to 25% the execution time of the theorem prover in many cases. For example, one theorem in 5 variables passing the test requires to perform more than 1.33 million useless valuations.

This inefficiency of matrix testing will change dramatically when the current research of M. McRobbie and R. Whaley to implement this test on the Connection Machine [Mc91] will be operational.

Finally, for efficiency reasons, this test is performed in C, and uses the algorithm of MaGIC. Doing this test in Prolog is possible, of course, but represents a waste of time. For example, testing a 5 variables theorem against a size 10 matrix in C (optimized) takes 7.2secs, but it takes 300.05secs in Prolog. Some optimization of the program could half that value, and some Prolog compiler with double indexing could still further improve this result [De89]. But, in any case, Prolog is not the appropriate language to perform this test. It should be noted though that matrix testing produced the best speedups results in parallel Prolog.

5.4 Databases of Theorems and Non-Theorems.

As we emphasized earlier, in order to avoid redundant computations and to speed up the execution of the program, it is necessary to maintain some of the information which is generated during the process of a proof.

The theorem prover is set to keep a record of what it proves and of what it refutes as well as of some other information. Doing so, the performances are increased in a very significant way.

The easiest way to store information in Prolog is to store it in the internal database using the assert/1 predicate. This technique, even though it consumes a lot of the computational resources, works well as long as the size of the database is kept relatively small.

The following table shows the effect of using this asserted information to process an entire
file of 169 Standard simple formulae with the theorem prover running on a Sun3/50.

<table>
<thead>
<tr>
<th>Information</th>
<th>Generated</th>
<th>Subsumed</th>
<th>Time (secs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theorems</td>
<td>534</td>
<td>288</td>
<td>No info: 236.4</td>
</tr>
<tr>
<td>Non-Theorems</td>
<td>134</td>
<td>69</td>
<td>Thms: 101.3</td>
</tr>
<tr>
<td>Pass filters</td>
<td>216</td>
<td>109</td>
<td>+ Non-Thms: 90.0</td>
</tr>
<tr>
<td>Not Pass filters</td>
<td>513</td>
<td>350</td>
<td>+Pass+Notpass: 84.5</td>
</tr>
</tbody>
</table>

The databases are updated at runtime during the processing of the entire file. That is, the theorem prover starts with no knowledge at all, apart from what an axiom is, and when it proves or refutes a formula it increments its databases. When the program is compiled with the data collected during a preceding run, a proof or a refutation amounts just to a simple table look up, and no computation is involved since the information is complete. With these small databases compiled, the entire file is processed in less than 2 seconds. But this is a toy example. Things are quite different when we want to prove difficult problems, and in this case, this technique has its limits.

A first problem comes from the inefficiency of asserting facts in the internal database. This is an expensive operation which may overflow the system, even though some Prolog compilers, like Sicstus, allow to preallocate storage space for the internal database and have efficient built-in routines to control any space overflow and perform garbage collection. Still there are hardware limitations, and some other technique to deal with large amount of information is required.

In order to keep the size of the database reduced, a subsumption or strong containment test is performed in order to discard any redundant information by retracting the subsumed formulae. This test still adds to the execution cost.

A second problem comes from the fact that the information originally collected was of little help in trying to solve hard problems. In order to further enhance the efficiency of the program, tests were carried out on all the available test files and on many of the theorems produced during the proofs of their elements to generate systematically as much information as possible. We ended up with four irredundant databases: ≈ 8000 theorems, ≈ 4000 non theorems, ≈ 1100 multisets which pass the filters and ≈ 1300 which do not pass the filters.

Keeping the databases redundant puts the total number to more than 25000 elements. The reason to use the full redundant databases is to avoid the subsumption test whose cost increases proportionally to the size of the databases since each candidate may have
CHAPTER 5. DATABASES AND MASSIVE PARALLELISM.

120

to be tested against many elements. It is simpler to retrieve information in the databases by matching. To do so, the databases are indexed and their elements are stored with identifying keys to reduce the retrieval time. Without indexing, finding an element takes from $\approx 40$ to $\approx 600$ msecs. Dividing the databases of theorems into 34 partitions, and the database of non-theorems into 19 partitions with their elements indexed by keys, already improves these figures. Since the formulae are in prefixed operators notation, these are used as keys, and matching being performed on the head first, retrieval of an element is quick.

Considering that the time required to unify two variables in the database is $\approx 16$ msecs, the partitions under the same keys are further subdivided into a total of 214 subparts according to the number of variables contained in each element. This implies some processing of the formula to match, but this operation is performed only once, while simple matching in the database could involve useless unification of several thousand variables, since most partitions contain many formulae differing only by their last subformula or literals, and some large partitions include formulae containing on average 16 variables. To sum up, the access time to the database and retrieval of any element (or no element found) takes on average $\approx 140$ msecs in the small partitions and $\approx 260-280$ msecs in the larger ones, and this is still too long.

Keeping large databases in the working space creates an additional problem: the size of the compiled databases (30 Mb) exceeds the working memory space and increases the overheads created by paging. As we will see, advanced technology can help to solve this problem. But all problems will not be solved by simply using however large databases. A good memory is of no use without some intelligence, and some is needed in the theorem prover in order to make it efficient.

In addition to the heuristics discussed in the preceding chapter, the database of preprocessed fusion subformulae mentioned above is consulted to select the best combinations of parameters. To make its function clearer, we recall the flowchart of the execution of the fusion rule in the figure representing the interfaces with the databases. It appears that the cost of this heuristic operation is, in general, higher than simply generating all possible combinations and testing them against the databases of theorems and non-theorems which contain most of these combinations anyway. Nevertheless, there are cases where the number of possible partitions is so large that a heuristic search, however expensive will still be more economical. Some additional data about this database are given in the next chapter.

Summarizing the use of databases, Figure 2, on next page, schematizes the interface to the internal databases in the flowchart of the execution of the proof procedure.
CHAPTER 5. DATABASES AND MASSIVE PARALLELISM.

Figure 5.2: Internal Databases.
5.5 Massive Parallelism.

The need to retain information may be obvious, but one problem is to store the information in such a way that it is readily available and easy to use, and moreover, in such a way that it does not overflow the system, nor slows down the execution of the program. With respect to the use of information, the advantage of Prolog or of the Logic Programming approach is that it combines in a single programming language relational databases, high level programming and logic, making it easy to manipulate information represented in structured data. (See, for example, [Ka90]).

Nevertheless, when running an application which generates large quantities of data, called a stream or ordered sequence of records [Pa90], it may be necessary to analyze this information on the fly, either because part of it is required to continue the execution, or to minimize the consumption of resources. Storage of information is a weakness of Prolog. The stacks described in Chapter three, as well as the internal database which contains the code and the asserted clauses, and the atom table in which all atoms are stored, can grow very rapidly to a very large size. A small program can already have a memory image of several Mbytes, and using a parallel compiler may require several Mbytes of memory per processor. For example, loading Aurora already reserves $\approx 4$ Mbytes per processor, and running a very small compiled program like $\texttt{fib}(N, M)$ or $\texttt{nqueens}(N, M)$ with it, requires $\approx 12$ Mbytes per processors! That means that very large amounts of memory are necessary to prevent the overheads of swapping to disk.

It appeared then that using some available technology could solve our problems to some extent.

5.5.1 Interfacing Prolog with a Connection Machine.

As we said, using large compiled databases which inflate the size of the running program creates a memory problem. Moreover, the database access time for each check, even with partition and indexing of the 25k elements databases, is still too long. Finally, some formulae are so hard to prove that it appeared necessary to use fast and powerful machines. This has led to the interfacing of Prolog with a Connection Machine to store the databases, to perform the multiset containment test and the selection of parametric constituents.

The CM2.

The Connection Machine works on the data parallel computing model using thousands of processors (8k, 16k,..., 64k) having each their own memory and being all driven by a
sequential machine, called the front-end [Hi85]. The Connection Machine Model 2, (CM2) at ANU is a 16k machine operated from a Sun 4/390.

Each processor of the CM2 is allocated an element of the data. In our implementation, the data are multisets or formulae and each subformula or constituent of a multiset is assigned to a processor and stored in its own memory. In the SIMD model, (single instruction, multiple data), all processors perform the same operation or instruction on all the data elements at the same time. The processors can communicate, and the program allows to control the work of the processors, for example, in assigning more tasks or in keeping idle some or all processors. Moreover, virtual processors are supported by dividing the memory associated with each physical processor to create some $2^n$ multiples of the set of processors. But in this case, increasing the number of virtual processors increases the time required to perform the operations, of course.

In order to make clear our use of the CM2, we briefly review some elementary notions of programming the CM in $C^*$, a parallel extension to $C$ [CM90].

Parallel data are declared according to their shape, i.e. according to the way in which the data must be organized. Depending on the nature of the data, its shape may be its number of dimensions represented in a grid whose coordinates indicate the rank, position,... of any element of data, and which allow to retrieve them. Then, parallel variables of this shape, i.e. variables on which the computation is performed, are declared. Parallel variables contain several elements like char, int,... And the role of shape is to define how many elements of a parallel variable there are.

There are several ways to communicate with the computation. Essentially, we use only two kinds of computation: scan and reduce.

Two positions are said to belong to the same scan class if their coordinates differ only along a specified axis of the grid. A scan subclass is constituted by the only active positions in the computations within a scan class. A function may operate independently on various parts of a scan subclass called a scan set, i.e. parts of the scan subclass created by a scan function which divides the scan subclass into scan sets. A reduce function puts the result of an operation into a single parallel variable element in each scan subclass.

### 5.5.2 Implementation on the CM2.

The first experiments with the Connection Machine amounted to "using a supercomputer as attached processors" [Mc85]. It also required some additions to the Prolog sources. This work was done with the help of R. Whaley of Thinking Machine Corporation who wrote the $C^*$ program for the CM2 and added the C routines to the Sicstus compiler, and later continued with J. Barlow of the ANU Parallel Computing Facility [Ri91]. In general,
Prolog implementations do not provide a “string” datatype. Of course, it is possible to go around this limitation with the name/2 predicate which translates an atom into a list of ASCII characters (integers) and vice versa. Still, a string has to be translated into an atom, and to do so requires some cumbersome techniques like writing to, and reading from a file, or using Unix virtual files.

For our purpose, in order to interface Prolog with C for efficient matrix testing and to C* to implement the databases and the containment test on the Connection Machine, it was necessary to hack the Sicstus sources code and to implement new built-in predicates, essentially some way to pass the address of a string to C or C* in an appropriate format.

Databases.

The databases of proved and refuted multisets (theorems and non-theorems) are stored in the Datavault parallel disk array at ANU’s Supercomputer Facility. The sequential Prolog (Sicstus 0.6) running on the front-end of the Connection Machine calls a C* routine which loads the databases into the memory of the Connection Machine at the beginning of the program. The databases remain in memory throughout the run. In their current form they contain 60k elements and consume 33Mbytes of memory. The CM2 at ANU has 512Mbytes of memory, so using significantly larger databases is feasible on this machine. With up to 8Gbytes of memory available in the largest CM2, extremely large databases are practical.

The data are loaded into the Connection Machine in such a way that each virtual processor contains a single subformula or constituent of a multiset (in the following we do not make any distinction between a subformula and a constituent). The virtual processors are arranged with adjacent virtual processors containing the particular constituents for each multiset. The first and last constituent of each multiset are marked to indicate the beginning and end of the multiset. And a counter marks the repetitions of identical constituents in a multiset. Additional information is attached to each constituent, i.e. declared as an element of the parallel variable, for example, if it is part of a theorem or not, or the type of the multiset. Additional types can be used, for example, some multisets corresponding to a particular selection or partition. See Figure 4.

The Prolog program represents multisets as lists of structures. The new built-in predicate allows to pass lists of structures to C* subroutines as a character string. Then the Prolog predicates called for testing a proof candidate formula or multiset against the database are replaced by a call to the C* routines which test the formula against the database on the Connection Machine.

The character string is parsed by some serial C routines which convert it to the format
Figure 5.3: Interface with the Databases on the Connection Machine.

Figure 5.4: Mapping of the Data Structures on the CM2.
of subformulae or constituents used in the database. Each subformula or constituent is then broadcast to the virtual processors on the Connection Machine where it is compared, in parallel, with the subformulae in the database. The matches are recorded. Once all the subformulae of the formula have been broadcast and compared, it remains to see which multisets in the database have all of their subformulae matched.

Using scan operators [Be87], the match information for all subformulae are combined within the formula or multiset. Where all constituents match, the formula matches. A global reduction is performed to see if any formula in the database matches and the result (0: no match, 1: non-theorem match, 2: theorem match) is returned by C* and bound in the appropriate Prolog rule.

What is important to note here, is that this test operation is computationally cheap, and the elements in the databases are checked all at once. When this experiment started, on a 8k processors Connection Machine this took 10msecs (wall time). Using the 16k available reduced this time to 5msecs. On a full 64k machine, and with a more efficient algorithm, an access time of 1msec could have been obtained.

The strong containment test performed by the theorem prover could also be implemented very efficiently with hash tables on both serial and parallel hardware. However, the algorithm described above is easily modified for more computationally expensive operations than strong containment and would provide similar performance in these cases.

**Heuristics.**

Speed-ups of several orders of magnitude are obtained in many cases. Nevertheless, not all the information we would need is already contained in the databases, even though they are updated every time some new multiset is proved or refuted.

A second addition to the theorem prover which further increases its efficiency is the implementation of the selection of parametric constituents on the Connection Machine, using the same algorithm as for the containment test. In the application of the fusion rule, we first compute all the combinations of $A$ with the parameters. These combinations are tested for matching on the Connection Machine which returns all matches found. The complement of $A$ (i.e. $B$ combined with the remaining parameters, see section 4.6.2) is then computed according to the conditions of the rule and tested in the same way as $A$. The matchings returned are then compared. And actually, it suffices that one matching of the complement exists in the database to have a proof of $A \circ B$. Otherwise the theorem prover running on the front end tries to find the missing information by applying the rules of the proof theory. This has proved to be the best improvement of the theorem prover efficiency.

There is still a more efficient way to implement this test, but it is less easy to program
though, and left for future research. Rather than generating all possible partitions for the left constituent of fusion and then, send them to the CM which returns the matching found etc…, A, B and the parameters should be sent at once and the test performed entirely in the CM2.

5.5.3 Memory Based Reasoning

This use of the Connection Machine suggests some new approach to automated theorem proving close to the memory based reasoning model. Or at least this model is worth investigating and could possibly suggest ideas which would reach the stage of an implementation. Memory based reasoning consists in solving a problem by intensive use of memory. Everything the system experiences is stored in memory, the only limitation being the physical size of the memory. When a situation similar to a previous one occurs, the memory of the past situation is used to derive an answer.

In the memory based reasoning model proposed by D. Waltz and C. Stanfill [Sw86] [Wa90], Best-Match Associative Memory Retrieval is the inference mechanism which consists in finding which of all the information contained in the system memory most closely matches the current situation. The CM2 is the ideal hardware to implement this idea because, regardless of the size of the memory, the retrieval time is constant.

Indeed, as we have seen in this chapter, the usual way of retrieving information by lookup of the database is computationally high: we may need to perform an exhaustive search, and in order to minimize this cost the information contained in the database has to be indexed etc… As we have also seen, on the CM2, we can store each element of the database in one processor. When we need some information, each element evaluates its matching with the current situation. And this evaluation is performed by all elements in the database in constant time.

[Sw86] argue that the basic associative memory operation of selecting relevant precedents in any situation is the essence of intelligent behavior. In presence of many precedents or incompatible precedents or “task space” unfamiliar, reasoning is required. We already said that a good memory without intelligence is of no use. Under the memory based model, we find the complementary slogan, no thought without memory. Nevertheless, the sort of symbolic processing we are interested in seems to be a little more specific than what the authors may have in mind. We need more than approximate matching, even though this could be usefull in the heuristic search. But under the scheme they propose, combinatorially explosive search can be avoided since, in any given situation, only a small number of “operations” are plausible, thus constraining the branching factors.

One could think that the proposal is more appropriate for some other fields of AI, like

1. See addenda, note 11.
knowledge representation or case-based reasoning, where deterministic heuristic search, inference and deduction may not be the appropriate paradigm. Nevertheless, and in opposition to the critique of Overbeek and Wos reported at the beginning of this chapter, the model proposed deserves some consideration, even in the domain of automated theorem proving. One reason is that when we want to prove a theorem, for example, we do not try blindly and systematically all possible legal applications of the rules, but we try to recognize some patterns in the unfolding of the proof and try to match them with some other available information. To take a trivial example, if we have $A \circ B$, we will try to find $\neg A + \neg B$. But this approach may not prevent the necessity of intensive computation in many cases. More strikingly, the proposed model seems to correspond to what sometimes happens in chess when some situation or pattern is recognized, i.e. has been seen and stored in memory in some earlier game, and the decision to make a move is based on that pattern and the previously acquired knowledge.
Chapter 6

Empirical Results.

6.1 Introduction.

In this last chapter we present and explain some results which illustrate and support the various points discussed in the preceding chapters.

First, the output of the statistics concerning a proof, as they are displayed by the theorem prover on completion of a proof or a refutation, are explained. This will allow us to show the performances of the theorem prover when various strategies are used, the amount of information produced and consumed, the effectiveness of the filters, mainly the matrix test, the relative efficiency of intelligent backtracking and the overall resources requirements of Prolog.

We already mentionned the Standard set of formulae which contains axioms of LR as well as other classical formulae of logic. The set of formulae Impset contains the 256 implicational formulae representing all the possible implications between the 16 formulae defining an associative connective that we referred to in the introduction. The 32 formulae of Asset correspond to these 16 equivalences divided into their pairs of implicational formulae. Some members of this set can be considered as hard problems to prove.

The results obtained in testing the 32 hard formulae of Asset under the original selection strategy and under the reverse strategy are compared in two tables.

Running the theorem prover in parallel has produced some interesting results which are reported as well as those obtained in interfacing the theorem prover with a Connection Machine.

Finally, we draw some conclusions from these experiments and results as well as some general considerations on parallel processing.
CHAPTER 6. EMPIRICAL RESULTS.

6.2 The Theorem Prover.

The following tables show the statistics as they are displayed by the theorem prover after each run. These two tables contain the cumulated statistics returned by the theorem prover after completing the execution of the Impset file. The results of the first table are obtained using the first strategy in the execution of a proof. That is, the selection of principal constituents of a multiset is based on a least cost first ordering of the constituents of a multiset.

The second table shows the results obtained using the second strategy to select the principal constituent, that is the cheapest constituents come last in the order of selection.

The results concern 238 formulae contained in the file Impset. This file actually contains 254 formulae, but even though all of them are proved or refuted by the theorem prover, some formulae whose proof takes much longer than any other under both orderings are not considered. The reason is that these few formulae would inflate considerably the figures reported, making some assessment of the average time required to prove or refute these formulae difficult.

No databases are used, and no heuristic search is performed. During the proof of each individual formula the information generated (corresponding to the theorems, non theorems, and formulae which pass the filters or not as was explained in the preceding chapter) is used, but the proof of each formula starts with a knowledge null.

Since the input formula is first tested against the filters, no non-theorem is actually proved such, since the matrices used refute all non-theorems contained in this file.

While explaining to what these statistics correspond, we can make several observations concerning these results.

i) Depth is the maximum depth reached in a proof. In some cases, the depth reached during the execution of a proof can be important. The reason is that contrary to "Kripke", there no suspension of the execution to scan the proof search tree and check whether some branch is hard to prove and, in this case, trigger an alternative choice.

This test could be implemented, but it actually presented several difficulties: first, the information concerning each parent node of a difficult branch should be stored, thus increasing the storage space requirements. Secondly, even though Prolog has all that information available internally on the stacks, it has to be made explicit in the coding of the test, and in addition, given the nature of Prolog execution, long jumps back in the tree are difficult to implement.

1. on pp. 132-133.
2. See addenda, note 12.
ii) **IntDBse**, indicates the number of elements used in a proof which were found in the internal database. In these tests, the database only consists of an axiom, \([X, \neg(X)]\). The external database counter **ExtDBse** returns the number of elements found in the database when running the theorem prover with the compiled databases.

iii) **Idem.Reduct** corresponds to the number of application of the \(K_o\) rule.

iv) **SkipSelect** is the number of complete failure nodes which have been stored. This gives some indication on the efficiency of intelligent backtracking since these nodes are only computed once.

v) **AxiomClosed** indicates the number of branches closed by reaching an axiom. In these tests it is equivalent to **IntDBse** since the database only contains the axiom schema.

vi) **Node Opened** is the number of nodes opened in the proof search tree while executing a proof. **Closed** is the number of nodes actually closed. Here it corresponds to the number of theorems proved during the execution.

vii) **To.filter** is the number of multiset candidates proposed as candidates to a proof by the rules of the proof theory. **Filtered** is the number of candidates actually filtered. This shows the efficiency of keeping a database of candidates which do pass or do not pass the filters test.

viii) The next subtable indicates the application and results of each of the five Filters. The results show that, on average, in this test, matrix testing is very efficient, and it is generally so.

ix) **Curry** corresponds to the number of multiset candidates which pass and fail the Curry property. This test is not included in the filters since the multiset candidates generated by the invertible rules do not undergo the filters test.

x) Finally, the subtable **MATRICES** shows the number of successful refutations by each matrix of the database of matrices. It is interesting to note that truth values together with \(RM3\) are very efficient, and together with \(CHAIN\) they often prevent the, in principle, powerful \(CRYSTAL\) to show its strength. Matrix 10.5 is the so-called \(KILLER\) matrix in [TMM88].

The last six size ten matrices which were added later in our investigations are most of the time useless. Since they were selected because they can refute some occasional multiset candidates which are not refuted by any other matrices, we may assume that they are only marginally stronger and, in general, would be more than a nuisance.
CHAPTER 6. EMPIRICAL RESULTS.

Considering their size and the number of multisets which have to be tested against them, their use is in general a waste of resources. Nevertheless, keeping databases of Pass/Notpass multisets reduces this waste. Apart from their effect of discarding some refutable multisets earlier, we were not able to discover whether, in some circumstances, they could significantly change the fate of a hard proof. That is, whether the non-theorems which pass the filters in the absence of these matrices significantly increase the complexity of a proof.

xi) Finally, since we insisted on the large use of resources by Prolog, the statistics concerning the memory usage and the garbage collection are given. It is interesting, and sometimes important when using Prolog, to pay attention to these figures which may provide some indication on the efficiency of the program or on the complexity of a proof. It is also interesting to compare these figures when processing a difficult problem. In some cases, the program can run for hours without inflating too much its memory usage, in other cases, it may fill up the entire memory in a few minutes. This is the case with some hard problems when the proof search tree which is unfolded in memory is so large that Prolog can no longer cope with it and crashes the program, and occasionally, crashes the entire system!

STATISTICS for IMPSET (SELECTION 1) after 5509.811 secs.

Node Opened: 3733  Closed : 3387  Max. Depth : 166
AxiomClosed: 844  IntDBse: 844  NewTheorems: 3387
Idem_Reduct: 6424  ExtDBse: 0  SkipSelect : 782
To_Filter: 57597  Filtered: 10924
 Filters  Pass  Fail
-------  -----  ----- 
PosNeg1: 9161  1763
PosNeg2: 507  1058
Strict : 7596  2072
Rule_2 : 7234  362
Matrix : 4264  2970

---
Pass : 4264  6660
---
Curry : 3388  23063

MATRICES

TV  R3  4V  5N  CH  6V  8.1  8.2  8.3  9.1  9.2

<table>
<thead>
<tr>
<th></th>
<th>798</th>
<th>948</th>
<th>1</th>
<th>0</th>
<th>1182</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>5</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

---

8.1 8.2 8.3 9.1 9.2
CHAPTER 6. EMPIRICAL RESULTS.

The second table now shows the results of the same test, but this time with the selection of the most expensive constituent as principal constituent, i.e. reversing completely the former selection order. Obviously, this strategy is more efficient. We already explained the principle of choice in Chapter 4, but we will come back to it later on when presenting explicit examples of the reasons why this strategy may be more efficient. It should be noted, though, that the application of this principle very often pays off, but it does not generalize. For example, the first formula of Impset is proved in a few seconds when the first strategy is applied, and in a very long time with the second strategy.

STATISTICS for IMPSET (SELECTION 2) after 1522.298 secs.

<table>
<thead>
<tr>
<th>Node Opened: 1568</th>
<th>Closed: 1553</th>
<th>Max. Depth: 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>AxiomClosed: 538</td>
<td>IntDBse: 538</td>
<td>NewTheorems: 1553</td>
</tr>
<tr>
<td>Idem_REduct: 171</td>
<td>ExtDBse: 0</td>
<td>SkipSelect: 129</td>
</tr>
<tr>
<td>To_Filter: 21015</td>
<td>Filtered: 5650</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>PosNeg1: 4230</td>
<td>1420</td>
<td></td>
</tr>
<tr>
<td>PosNeg2: 408</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>Strict: 4012</td>
<td>626</td>
<td></td>
</tr>
<tr>
<td>Rule_2: 3914</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Matrix: 1134</td>
<td>2780</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pass</th>
<th>1134</th>
<th>4516</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curry</td>
<td>1557</td>
<td>750</td>
</tr>
</tbody>
</table>

MATRICES

<table>
<thead>
<tr>
<th>TV</th>
<th>R3</th>
<th>4v</th>
<th>5N</th>
<th>CH</th>
<th>6V</th>
<th>8.1</th>
<th>8.2</th>
<th>8.3</th>
<th>9.1</th>
<th>9.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>293</td>
<td>541</td>
<td>5</td>
<td>0</td>
<td>1921</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10.1</th>
<th>10.2</th>
<th>10.3</th>
<th>10.4</th>
<th>10.5</th>
<th>10.6</th>
<th>10.7</th>
<th>10.8</th>
<th>10.9</th>
<th>10.10</th>
<th>10.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

memory (total) 6995992 bytes
program space 1158812 bytes
garbage collections 25, collected 21500488 bytes
CHAPTER 6. EMPIRICAL RESULTS.

memory (total) 3801112 bytes
program space 1116190 bytes
garbage collections 25, collected 19737952 bytes

The same tests were also performed without intelligent backtracking and, contrary to what we expected, on this file they showed a $\approx 20\%$ decrease of the performances of the theorem prover for both strategies. The reason is simply that the technique finds only a few opportunities to apply, and when it does, it does it in only few cases. This makes the cost of maintaining a memo table which is checked at each selection of principal constituent prohibitive with respect to the expected gains, and in the case of relatively simple problems, it is then simply cheaper to perform the same operation several times. We will see later on the example of a formula of Asset that the same technique increases the efficiency by $\approx 25\%$.

6.3 Data Bases of Matrices.

As we said, matrices play an important role as filters to prune the search tree. It is then important to select the most efficient ones. The following table identifies the most efficient matrices which were empirically found. J. Slaney’s program, MaGIC, was used to generate the 4533 LR-matrices and each element of the database of 8565 non-theorems was tested against each of them to assess their relative refuting strength. The first column indicates the number of the matrix in the database of the theorem prover, that is the order in which they are used in the filter test. The second column indicates its size, i.e. the number of elements of the matrix. Then successively, its negation table, partial order, the designated value, the number of the matrix in the “ugly”, or machine readable, output file of MaGIC, and finally, its relative efficiency in refuting the non-theorems of the more than 8k database, that is, how much of the 8k non-theorems it refutes.

It is interesting to note that, even though the sizes 2 and 3 matrices have a relatively low efficiency, when testing the entire database against them, nevertheless, as the results indicate, in actual practice, they perform very well and, in general, matrices larger than $CHAIN$ have few opportunities to apply. Nevertheless, in some cases, larger matrices could play an essential role in pruning the search tree efficiently. For example, the following formula is refuted in 3 minutes by “Kripke”, in much longer by the Prolog theorem prover, while having the appropriate matrix in the database would refute it at once. But there are only two size 10 LR matrices which refute this formula. The problem, then, is to assess the benefits of incorporating one of these in the database of matrices, and this would require much more empirical investigations. Still, having the most efficient matrices at hand will not solve all problems. For example, the following formula is refuted by “Kripke”, but it is not refuted by any of the LR matrices up to size ten.
CHAPTER 6. EMPIRICAL RESULTS.

Even though such formulae may be rare, they may show up from time to time, and thus require stronger models, larger than size 10.

<table>
<thead>
<tr>
<th>Matrix ID</th>
<th>Size</th>
<th>Negation</th>
<th>Order</th>
<th>Design</th>
<th>Number</th>
<th>Matriz Nber</th>
<th>Rel. Eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11.81</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>32.10</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>32.10</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>32.15</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>96.74</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>95.84</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1</td>
<td>25</td>
<td>2</td>
<td>5</td>
<td>266</td>
<td>92.82</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>326</td>
<td>99.91</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>25</td>
<td>2</td>
<td>4</td>
<td>265</td>
<td>96.74</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>841</td>
<td>99.94</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>842</td>
<td>99.94</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>11</td>
<td>3289</td>
<td>99.94</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>2</td>
<td>37</td>
<td>3</td>
<td>2</td>
<td>3813</td>
<td>99.94</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>2</td>
<td>39</td>
<td>3</td>
<td>5</td>
<td>3863</td>
<td>99.94</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>2</td>
<td>49</td>
<td>1</td>
<td>34</td>
<td>4235</td>
<td>96.82</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>2</td>
<td>50</td>
<td>1</td>
<td>29</td>
<td>4321</td>
<td>96.80</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>1</td>
<td>133</td>
<td>2</td>
<td>4</td>
<td>2941</td>
<td>96.76</td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>1</td>
<td>133</td>
<td>2</td>
<td>6</td>
<td>2943</td>
<td>94.52</td>
</tr>
<tr>
<td>19</td>
<td>10</td>
<td>1</td>
<td>139</td>
<td>2</td>
<td>7</td>
<td>3112</td>
<td>96.76</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>1</td>
<td>140</td>
<td>2</td>
<td>25</td>
<td>3210</td>
<td>95.27</td>
</tr>
<tr>
<td>21</td>
<td>10</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>14</td>
<td>4306</td>
<td>99.91</td>
</tr>
<tr>
<td>22</td>
<td>10</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>4500</td>
<td>96.82</td>
</tr>
</tbody>
</table>

One may wonder what is the effect of using a large number of matrices on the efficiency of the theorem prover. In the case of Impset, the following statistics show the results of running the same tests as in the previous section, but this time with only the matrix CHAIN.

STATISTICS for IMPSET (SELECTION 1) after 2351.5 secs.

Node Opened: 3075  Closed : 2722  Max. Depth : 123
AxiomClosed: 695  IntDBse: 2741  NewTheorems: 2722
Idem_Reduct: 2475  ExtDBse: 0  SkipSelect : 1093
To_Filter: 46403  Filtered: 8760
CHAPTER 6. EMPIRICAL RESULTS.

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>PosNeg1:</td>
<td>7276</td>
<td>1484</td>
</tr>
<tr>
<td>PosNeg2:</td>
<td>438</td>
<td>607</td>
</tr>
<tr>
<td>Strict:</td>
<td>6231</td>
<td>1483</td>
</tr>
<tr>
<td>Rule_2:</td>
<td>5951</td>
<td>280</td>
</tr>
<tr>
<td>Matrix:</td>
<td>2836</td>
<td>3115</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>Pass:</td>
<td>2836</td>
<td>5924</td>
</tr>
<tr>
<td>Curry:</td>
<td>2863</td>
<td>6266</td>
</tr>
<tr>
<td>MATRICES. CHAIN: 3115</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And the same test, using the second strategy shows that, even though many multisets pass the matrix test, the entire file of formulae is processed in a relatively short time.

STATISTICS for IMPSET (SELECTION 2) after 915.4 secs.

- Node Opened: 1494 Closed: 1469 Max. Depth: 71
- AxiomClosed: 518 IntDBse: 1004 NewTheorems: 1469
- Idem_Reduct: 158 ExtDBse: 0 SkipSelect: 712
- To_Filter: 19909 Filtered: 5377

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>PosNeg1:</td>
<td>4016</td>
<td>1361</td>
</tr>
<tr>
<td>PosNeg2:</td>
<td>391</td>
<td>178</td>
</tr>
<tr>
<td>Strict:</td>
<td>3803</td>
<td>604</td>
</tr>
<tr>
<td>Rule_2:</td>
<td>3710</td>
<td>93</td>
</tr>
<tr>
<td>Matrix:</td>
<td>1051</td>
<td>2659</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>Pass:</td>
<td>1051</td>
<td>4326</td>
</tr>
<tr>
<td>Curry:</td>
<td>1714</td>
<td>687</td>
</tr>
<tr>
<td>MATRICES CHAIN 2659</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Obviously, in this case, using \texttt{CHAIN} only is sufficient. More and larger matrices will only impose a large amount of useless computation. But, with hard problems, when there is no way to know how long it will take to prove a formula, the overhead of matrix testing may be less significant and the reduced performances are compensated by the ability to prune the tree efficiently. And this is important considering the examples of formulae, as
above, which have no refutation in a short time, i.e. which require large unfolding of the
tree before a refutation is found.

6.4 Data Base of Partitions.

One way to implement the heuristics and, at the same time, to avoid useless computation
would be to precompile the possible combinations of principal and parametric constituents
in the application of the fusion rules.

This precomputation being expensive, two databases of preprocessed partitions which pass
the filters test, and thus constitute possible good choice of parameters, have been generated
from all the fusion subformulae of the entire Impset file. The next table summarizes these
results and shows that consulting such a file during the execution of a proof may spare the
resources to a large extent. In practice, it appeared that the time overheads created by
the consultation of the file were larger than anticipated, and the expected speedups were
often negative, mainly in the case of simple formulae. Since the possible bad combinations
of constituents of principal with parameters are probably already recorded in the database
of multisets which do not pass the filters, in many cases it will be cheaper to generate all
the possible combinations and test them against the database of formulae which pass or
not the filters. Nevertheless, there are cases where a right selection is required because
we can be confronted with the following situation which occurs with the second formula
of Asset under the least cost first ordering. The number of parametric constituents is so
large that a, the right constituent, can legally be combined in 8192 ways with them, and
obviously, we do not want to try them all.

****FUSION 1  Depth 44
Princ: o(v(#(c),c),#(o(¬(b),¬(b))),(¬(b))),o(#(b,b),o(v(¬(c),c),b))),a)
Param: [o(¬(a),¬(a)),¬(b),¬(b),¬(b),¬(b),¬(b),¬(b),¬(b),¬(c),¬(c),¬(c),¬(c),¬(c),¬(c)]

The table of partitions first gives the number of fusion subformulae generated and
the number of unique such subformulae. This shows that, in general, we can expect the
recurrence of the same combinations many times. This is more true in the case of hard
problems. For example, the same test performed on the Asset file produced 917505 fu­
sion subformulae. In addition, the table shows the relative efficiency of the various filters.
Allmat corresponds to the first 12 matrices in the previous list of matrices.
We ended up with 9188 different possible combinations of fusion subformulae with pa­
rameters generated by Impset. This means that these are the only “good” combinations to
select in the application of the fusion rules. Nevertheless, as we already said, the database
look up is too expensive to perform when processing this file and it has not proved to
be of any really efficient use when dealing with harder problems. The heuristic search on this database was commented out as soon as a better approach to the problem, the smart selection of constituents on the Connection Machine, was found.

<table>
<thead>
<tr>
<th>Asset Partitions</th>
<th>Generated</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filters</td>
<td>199754</td>
<td>36583</td>
</tr>
<tr>
<td>Pos/Neg</td>
<td>189852</td>
<td>36215</td>
</tr>
<tr>
<td>Parts</td>
<td>161079</td>
<td>34345</td>
</tr>
<tr>
<td>Rule2</td>
<td>188873</td>
<td>34113</td>
</tr>
<tr>
<td>CHAIN</td>
<td>62105</td>
<td>9214</td>
</tr>
<tr>
<td>AllMat</td>
<td>61610</td>
<td>9188</td>
</tr>
</tbody>
</table>

6.5 Hard Problems.

The file Asset contains the 32 hard formulae used in an attempt to solve the decision problem for the logic $R$, as we explained in the introduction (See also [MMT83]). Some of these formulae are easily proved, other may be considered as exemplifying the complexity of the logic and have not been proved automatically. They were all proved by hand, in one direction, by M. McRobbie using his tableau method, and with the help of "Kripke". But even "Kripke" running on a Cray2 has been unable to prove them all. Results communicated by M. McRobbie (1988) when we started this research indicate a decrease of the execution time, but using the power of a supercomputer did not prove any formula that had not been proved before. And this showed that whatever the computational resources used, deeper insight into the logic and good heuristics were needed.

We have previously given the reason why we think some of these formulae are difficult to prove. A comparison of the following two tables shows to what extent the order of selection of a principal constituent of a multiset can make a big difference. We will come back to these reasons below.

These two tables reproduce most of the interesting statistical results returned by the theorem prover when trying to prove the 32 hard formulae on a Sun4. The prover was set as in the previous test on Impset, to apply intelligent backtracking and no external databases were used. The abbreviations should be obvious and what they stand for is explained in section 6.2. $CRY$ indicates the number of successful applications of the Curry property. $TIME$ is indicated in seconds, otherwise, a '-' means that no proof has been found, and in most cases, the reason is that the program crashed due to the generation of a data structure or of a string of characters too long for Prolog, or the test was voluntarily stopped. A "-" in the column $MEM$ indicates that the usage of memory was small.
## Table 1: Results for Asset Using the First Strategy.

<table>
<thead>
<tr>
<th>F</th>
<th>NOD</th>
<th>RED</th>
<th>DPT</th>
<th>AX</th>
<th>THM</th>
<th>SKP</th>
<th>FLT</th>
<th>FLD</th>
<th>PASS</th>
<th>CRY</th>
<th>MEM</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>137</td>
<td>2</td>
<td>31</td>
<td>42</td>
<td>119</td>
<td>74</td>
<td>1178</td>
<td>513</td>
<td>133</td>
<td>91</td>
<td>2.9</td>
<td>140.6</td>
</tr>
<tr>
<td>2</td>
<td>402</td>
<td>62</td>
<td>89</td>
<td>85</td>
<td>183</td>
<td>1138</td>
<td>18105</td>
<td>5592</td>
<td>321</td>
<td>141</td>
<td>6.6</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>340</td>
<td>213</td>
<td>195</td>
<td>24</td>
<td>99</td>
<td>194</td>
<td>10884</td>
<td>1796</td>
<td>795</td>
<td>2948</td>
<td>14</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>159</td>
<td>17</td>
<td>76</td>
<td>5</td>
<td>92</td>
<td>55</td>
<td>20179</td>
<td>509</td>
<td>227</td>
<td>835</td>
<td>126</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>251</td>
<td>112</td>
<td>51</td>
<td>66</td>
<td>236</td>
<td>43</td>
<td>7302</td>
<td>1309</td>
<td>219</td>
<td>348</td>
<td>3.3</td>
<td>1369</td>
</tr>
<tr>
<td>6</td>
<td>7655</td>
<td>2097</td>
<td>167</td>
<td>31</td>
<td>829</td>
<td>555</td>
<td>24830</td>
<td>3593</td>
<td>1025</td>
<td>3579</td>
<td>6.7</td>
<td>7188</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>2.1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>1.4</td>
</tr>
<tr>
<td>9</td>
<td>224</td>
<td>131</td>
<td>72</td>
<td>61</td>
<td>187</td>
<td>49</td>
<td>4126</td>
<td>925</td>
<td>243</td>
<td>343</td>
<td>2.9</td>
<td>523</td>
</tr>
<tr>
<td>10</td>
<td>1589</td>
<td>744</td>
<td>96</td>
<td>73</td>
<td>693</td>
<td>686</td>
<td>13550</td>
<td>2717</td>
<td>915</td>
<td>1668</td>
<td>4.2</td>
<td>3898</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>11</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>0</td>
<td>-</td>
<td>4.2</td>
</tr>
<tr>
<td>12</td>
<td>382</td>
<td>585</td>
<td>209</td>
<td>53</td>
<td>131</td>
<td>114</td>
<td>32403</td>
<td>3948</td>
<td>659</td>
<td>3759</td>
<td>12.9</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>783</td>
<td>3797</td>
<td>230</td>
<td>24</td>
<td>322</td>
<td>735</td>
<td>86310</td>
<td>4640</td>
<td>1242</td>
<td>10835</td>
<td>24.6</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>392</td>
<td>4479</td>
<td>143</td>
<td>31</td>
<td>255</td>
<td>330</td>
<td>31250</td>
<td>1614</td>
<td>466</td>
<td>6739</td>
<td>9.9</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>511</td>
<td>62</td>
<td>120</td>
<td>101</td>
<td>210</td>
<td>2921</td>
<td>36028</td>
<td>8408</td>
<td>441</td>
<td>425</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>791</td>
<td>64</td>
<td>120</td>
<td>125</td>
<td>324</td>
<td>3862</td>
<td>48214</td>
<td>10591</td>
<td>652</td>
<td>457</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>247</td>
<td>776</td>
<td>121</td>
<td>31</td>
<td>59</td>
<td>163</td>
<td>2076</td>
<td>855</td>
<td>197</td>
<td>979</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>325</td>
<td>66</td>
<td>119</td>
<td>16</td>
<td>61</td>
<td>345</td>
<td>2621</td>
<td>696</td>
<td>182</td>
<td>514</td>
<td>2.9</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td>-</td>
<td>3.8</td>
</tr>
<tr>
<td>20</td>
<td>963</td>
<td>743</td>
<td>313</td>
<td>80</td>
<td>432</td>
<td>226</td>
<td>38982</td>
<td>2524</td>
<td>1755</td>
<td>32122</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>1.6</td>
</tr>
<tr>
<td>22</td>
<td>3511</td>
<td>1706</td>
<td>1293</td>
<td>32</td>
<td>59</td>
<td>164</td>
<td>15837</td>
<td>4281</td>
<td>1834</td>
<td>5021</td>
<td>11.9</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1631</td>
<td>1006</td>
<td>618</td>
<td>133</td>
<td>625</td>
<td>675</td>
<td>54454</td>
<td>6209</td>
<td>1486</td>
<td>2116</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>1125</td>
<td>1015</td>
<td>303</td>
<td>90</td>
<td>606</td>
<td>986</td>
<td>17751</td>
<td>3943</td>
<td>1375</td>
<td>3859</td>
<td>6.7</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>1208</td>
<td>3686</td>
<td>600</td>
<td>60</td>
<td>423</td>
<td>364</td>
<td>34380</td>
<td>6814</td>
<td>1898</td>
<td>7516</td>
<td>26.5</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>1503</td>
<td>8</td>
<td>428</td>
<td>7</td>
<td>39</td>
<td>53</td>
<td>20404</td>
<td>1849</td>
<td>560</td>
<td>2533</td>
<td>5.5</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>483</td>
<td>0</td>
<td>255</td>
<td>7</td>
<td>35</td>
<td>9</td>
<td>6215</td>
<td>1162</td>
<td>326</td>
<td>734</td>
<td>5.5</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>1340</td>
<td>957</td>
<td>1071</td>
<td>16</td>
<td>18</td>
<td>164</td>
<td>16857</td>
<td>3615</td>
<td>1968</td>
<td>7155</td>
<td>11.7</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>280</td>
<td>63</td>
<td>145</td>
<td>15</td>
<td>70</td>
<td>752</td>
<td>11571</td>
<td>915</td>
<td>326</td>
<td>1609</td>
<td>14.6</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>394</td>
<td>44</td>
<td>183</td>
<td>13</td>
<td>106</td>
<td>380</td>
<td>7080</td>
<td>1583</td>
<td>548</td>
<td>1233</td>
<td>6.18</td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>21</td>
<td>0</td>
<td>11</td>
<td>7</td>
<td>21</td>
<td>0</td>
<td>44</td>
<td>37</td>
<td>17</td>
<td>0</td>
<td>1.9</td>
<td>9.9</td>
</tr>
<tr>
<td>32</td>
<td>320</td>
<td>1267</td>
<td>116</td>
<td>34</td>
<td>259</td>
<td>126</td>
<td>4910</td>
<td>884</td>
<td>336</td>
<td>2411</td>
<td>2.96</td>
<td>678.7</td>
</tr>
</tbody>
</table>
The following tables of statistics show the effect of the filters in trying to prove formulae 15 and 16 of Asset using the first strategy. The rest of the statistics figures in the first of the previous two tables. It is interesting to note the pruning work performed by the database of matrices, particularly the matrices of size larger than 5. This shows that more
and stronger matrices are required in hard cases, but since, in this case, no proof is found under this strategy, more matrices will not compensate the lack of right heuristics.

### STATISTICS FOR FML 15 (SELECTION 1) incomplete run

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosNeg1:</td>
<td>7780</td>
<td>628</td>
</tr>
<tr>
<td>PosNeg2:</td>
<td>82</td>
<td>321</td>
</tr>
<tr>
<td>Strict:</td>
<td>7377</td>
<td>485</td>
</tr>
<tr>
<td>Rule_2:</td>
<td>7375</td>
<td>2</td>
</tr>
<tr>
<td>Matrix:</td>
<td>441</td>
<td>6934</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Pass</th>
<th>441</th>
<th>7967</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curry</td>
<td>745</td>
<td>425</td>
</tr>
</tbody>
</table>

### MATRICES

<table>
<thead>
<tr>
<th>TV</th>
<th>R3</th>
<th>4V</th>
<th>5N</th>
<th>CH</th>
<th>6V</th>
<th>8.1</th>
<th>8.2</th>
<th>8.3</th>
<th>9.1</th>
<th>9.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>309</td>
<td>861</td>
<td>0</td>
<td>0</td>
<td>3192</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1496</td>
<td>748</td>
<td>0</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>10.1</td>
<td>10.2</td>
<td>10.3</td>
<td>10.4</td>
<td>10.5</td>
<td>10.6</td>
<td>10.7</td>
<td>10.8</td>
<td>10.9</td>
<td>10.10</td>
<td>10.11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>326</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### STATISTICS FOR FML 16 (SELECTION 1) incomplete run

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosNeg1:</td>
<td>9842</td>
<td>749</td>
</tr>
<tr>
<td>PosNeg2:</td>
<td>87</td>
<td>488</td>
</tr>
<tr>
<td>Strict:</td>
<td>9267</td>
<td>662</td>
</tr>
<tr>
<td>Rule_2:</td>
<td>9265</td>
<td>2</td>
</tr>
<tr>
<td>Matrix:</td>
<td>652</td>
<td>8613</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Pass</th>
<th>652</th>
<th>9939</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curry</td>
<td>1075</td>
<td>457</td>
</tr>
</tbody>
</table>

### MATRICES

<table>
<thead>
<tr>
<th>TV</th>
<th>R3</th>
<th>4V</th>
<th>5N</th>
<th>CH</th>
<th>6V</th>
<th>8.1</th>
<th>8.2</th>
<th>8.3</th>
<th>9.1</th>
<th>9.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>452</td>
<td>967</td>
<td>0</td>
<td>0</td>
<td>4022</td>
<td>116</td>
<td>0</td>
<td>0</td>
<td>1834</td>
<td>862</td>
<td>0</td>
</tr>
</tbody>
</table>

---
CHAPTER 6. EMPIRICAL RESULTS.

10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9 10.10 10.11

0 0 0 0 359 0 0 0 0 0 0

Comments.

The effect of changing the order of selection of principal constituent of a multiset seems to be effective in several cases. But, obviously, comparing the two tables, this strategy cannot be generalized. It seems that some in depth preanalysis of the formulae is required to be able to select the best constituent. In many cases, the trouble maker is easily identified. For example, trying to prove formula 22 under the first selection of principal, i.e. cheapest first, the same constituent of the multiset keeps on coming in front of the list of candidates principal. The problem is that one of its literals combined with some parameters passes throughout the filters becoming a new candidate multiset for selection... and the same scenario repeats, a new identical principal comes first in the list, is selected and creates the same problem. This explain why, without heuristics and databases, the proof search tree of formula 22 grows very rapidly, passing from depth 8 to 193 and further down without being stopped by the Curry property, and only a few multisets are proved on the way. Changing the selection makes all the difference. It is interesting to note that the same observation has been made in the context of Constraint Logic Programming. In the CAL (Contraintes avec Logique) language developed at ICOT, and based on the computation of the Gröbner bases, some problems, i.e. polynomial equations, are very hard to prove. According to one of the developers, A. Aiba, changing the ordering of the polynomials may also make some difference.

We may then suggest that this sort of phenomenon may, in some cases, be responsible for the surprising inefficiency of "Kripke", which is otherwise extremely powerful. For example, the following two formulae differ only by one negative literal.

\[
(((a)a)(a)(b)((b)b)+((c)c))V(((b)b)(b)(c)c))+((a+((a+(a+a)+a)))
\]

\[
(((a)a)(a)(b)((b)b)+((c)c))V(((b)b)(b)(c)c))+((a+((a+(a+a)+a)))
\]

"Kripke" proves the first in 3 seconds but is unable to prove the second, and there are plenty of similar examples. In any case, this example shows that an apparently innocent negative literal may well increase the complexity of a proof in a very significant way.

Given the effect of changing the order of selection of candidates principal, it is obvious and rather trivial to assume that there is an optimal selection of principal and parametric constituents every time the selection is applied to a multiset.
CHAPTER 6. EMPIRICAL RESULTS.

For example, formula 9 is proved in 7.6 seconds under some arbitrary selection of principal where the members of the list of candidates are permuted at random. Depending on the number of candidates in the list, the order is the following: [2,1]; [2,3,1]; [3,2,1,4]; [4,3,2,1,5]; [3,2,1,4,5,6]. And the same selection is not as efficient in other cases. Under the same ordering of selection, formula 29 is proved in 1001.7 seconds. But all permutations were not tried, of course. These examples only show that the appropriate selection has to be performed by an intelligent heuristic search.

To conclude this section, it should be noted that the problem of the complexity of the logic and the difficulty of “Kripke” to prove some formulae was once discussed during a seminar. The case example was the following formula

\[(A \rightarrow B) \& (B \rightarrow C) \& (C \rightarrow D) \& (D \rightarrow E) \rightarrow A \rightarrow E\]

which is easily proved by A. Bollen’s CLOGPROG but takes a rather long time to be proved by “Kripke”. As R. Meyer, noted, the problem with this formula is that, in order to prove it, the entire search tree has to be unfolded and the opportunity to prove it comes last. Hence, it is obvious that whenever the execution of the proof theory allows to make choices, whenever possible, heuristics should guide the choices. 1

6.6 Parallel Processing.

Parallel processing is required to deal with hard and computationally expensive problems. We first present some results obtained with the parallel Prolog compiler Aurora, then the results obtained in using the Connection Machine.

6.6.1 Parallel Prolog Processing.

While experimenting with parallel Prolog, the best results were obtained on matrix testing and show a close to linear speedup on the 26 processors Sequent Symmetry of the Argonne National Laboratory. (Figure 1).

With only 7 processors on the ANU Symmetry, Figure 2 shows that the amount of work is sufficient and equally distributed to keep all workers busy and show a linear speedup. At some point, though the graph starts diverging, as in the preceding case, due to the progressive consumption of available work.

One explanation for this sort of linear speedup result may be found in [Sp87] who show that superlinear speedups can be obtained on some classes of problems, their results being obtained on a case similar to matrix testing. What explains the superlinear speedups is actually the random distribution of solutions in the search space.

When running the entire theorem prover in parallel, the results are no longer as good, but, nevertheless, they show some interesting speedup when compared to results obtained

1. See addenda, note 12.
CHAPTER 6. EMPIRICAL RESULTS.

Time

5 -4-- __ _...._ ___ -+----
1 e+02 -1r--------+---"---
I ',
309 x 732',
291 x 765')
52
144

5 variables
Linear speedup
4 variables
Linear speedup

24 Processors

Figure 6.1: Parallel Matrix Testing on 24 processors.

Time

8

5

4

3

2.5

2

1.5

1e+00 2 5

4 variables
Linear speedup

7 Processors

Figure 6.2: Parallel Matrix Testing on 7 processors.
with other systems as we will see infra. Figure 3 shows the results of running one parallel version of the theorem prover with 21 processors on the Symmetry at Argonne. The strange behavior of the program has induced some changes in the code, restricting the number of parallel predicates as well as wiser use of cavalier commit. Figure 5 shows the improvement for the same test on a Sequent Symmetry with 8 processors at ANU. The differences in timing are explained by the different processors on the two machines.

![Figure 6.3: Theorem Prover 21 processors. Test1](image)

When some changes are made in the code, making some predicates parallel etc, the performances are obviously improved, Figure 4.

An average 8.8 times speedup is obtained on 21 processors. And this agrees with the results of [Sz89]. Better results could be expected, but it is relatively difficult to obtain full parallel processing on an entire theorem prover program. According to R. Lusk, Stickel's PTTP when tested with Aurora did not perform up to expectations. Figure 6 finally shows the best result obtained on 15 processors.

### 6.6.2 Discussion

One reason why a close to linear speedup is difficult to obtain when running the theorem prover in parallel is that all the parts of the theorem prover are not parallelizable. In addition, and this seems to be the most significant feature of the figures, it has proved impossible to avoid the formation of some "peaks", that is some slow down of the execution when the number of processors increases. This was originally interpreted as some problem in the code, or some defect of the compiler, or else, some worker which refused to die when
CHAPTER 6. EMPIRICAL RESULTS.

Figure 6.4: Theorem Prover 21 processors. Test2

Figure 6.5: Theorem Prover 8 processors.
the processing had finished. After discussion of this problem with P. Thistlewaite, it seems reasonable to assume that when the number of created processes or workers increases, we may be in a situation where one of them will start working on a branch which increases the complexity of the proof while a solution is available on another branch which would have been visited earlier if the number of workers had been lower. If it is the case, then it seems that some changes in the code could prevent this situation from occurring. One such effect is represented in Figure 6.6.

6.6.3 Conclusion on Parallel Prolog.

In their performance analysis of a parallel execution model for Prolog which supports AND and OR parallelism as well as intelligent backtracking, [Fa90] review all usual benchmarks used in assessing the amount of parallelism of Prolog programs in what they claim is the most comprehensive study available. It is fair to say that the paper was submitted two years before its publication, so that the authors had probably no access to the few results obtained with the best parallel Prolog or Logic Programming compilers available today (Aurora, Pepsys, FGHC and KL1) which were mainly published after 1988. A better analysis which partly supports and corrects their results can be found in [Ti91]. Nevertheless the results of the first authors are interesting in that they show which benchmarks should not be used to assess the parallelism of a logic program, and, in addition, they explain why it is so. Most of their findings are corroborated by our experiments with Aurora except for the speedups they claim on some benchmarks which are actually more
important with Aurora than theirs.

A surprising result is that they obtain the best speedups with a Prolog version of the Boyer-Moore theorem prover (337 lines of code), 8.28 on 32 processors in AND-parallel mode. With our Prolog theorem prover, a 8 times speedup is obtained on 24 processors in OR-parallel mode, but some linear speedups are obtained in parts of the theorem prover, for example, as we have seen, the matrix testing and some combinatorial manipulation of sets and multisets.

A second surprising result of [Fa90] is what they call the supermultiplicative behavior of some programs whose performance improvement is obtained when several techniques are combined in running the same program: AND and OR parallelism plus intelligent backtracking. For example, AND-parallelism speeds up a benchmark program, ckt4, by 1.13, OR-parallelism by 1.79, and intelligent backtracking by 8.48. Multiplying these effects gives 17.15, but the actual speedup on 32 processors is larger, 28.72. This result could not be reproduced using OR-parallelism and intelligent backtracking alone. It is quite likely that this sort of speedup will be attained with Andorra [Ti91], which is basically Aurora with the addition of AND-parallelism.

They finally remark that intelligent backtracking has, in general, few opportunities to apply in Prolog programs. But this depends very much on the specificity of the program as the Fibonacci example and our results of chapter 6 show. For the interested reader, we add in Figure 7 the graph of the execution of \( \text{fib}(N, M) \) showing the advantages of keeping the memory of what has been computed throughout the execution. In logarithmic representation, the figure shows the overhead created by the storage and lookup of the table which is soon compensated by the reduction of complexity.

It is interesting to consider a logarithmic representation of the graph which shows the effect of asserting and checking the table. The overhead is soon eliminated when the gain obtained in avoiding repeating the same operation is larger than the overhead of checking the table in Figure 8.

6.7 Massive Parallelism.

6.7.1 Databases on the CM2.

At the moment, the databases installed on the CM2 contain 60k multisets representing 150k constituents and use 33Mbytes of memory. Since 512Mbytes of memory are available, with an effective practical amount of \( \approx 400 \text{Mbytes} \), there is no space problem in updating the databases. In addition, on the largest CM2, 8Gbytes of memory are available.

The following table summarizes the access and retrieval time on an 8k processors CM2
CHAPTER 6. EMPIRICAL RESULTS.

with respect to the number of virtual processors. On 16k processors, these figures hold true, and similarly up to 64k, if the traditional database are used, these figures hold true.

Figure 6.7: Fibonacci.

Figure 6.8: Fibonacci.
CHAPTER 6. EMPIRICAL RESULTS.

with respect to the number of virtual processors. On 16k processors, these figures have to be halved, and similarly up to 64k. If the irredundant databases are used, these figures can still be halved.

<table>
<thead>
<tr>
<th>Virtual Processors</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8k</td>
<td>2-5 msec.</td>
</tr>
<tr>
<td>16k</td>
<td>5 msec.</td>
</tr>
<tr>
<td>On 8k processors:</td>
<td></td>
</tr>
<tr>
<td>32k</td>
<td>10 msec.</td>
</tr>
<tr>
<td>64k</td>
<td>20 msec.</td>
</tr>
<tr>
<td>132k</td>
<td>40 msec.</td>
</tr>
<tr>
<td>264k</td>
<td>80 msec.</td>
</tr>
</tbody>
</table>

6.7.2 Results.

We already emphasized the advantages of using a massively parallel machine. The following tables of statistics show the result of proving formula 12 of Asset on the Connection Machine, using the internal and external databases and intelligent backtracking and under the original selection of candidate principal.

In these tables, IntDBse indicates the number of theorems found during the execution of the proof which were used. CM_DB reports the number of theorems and non-theorems found in the databases and which were effectively used in the proof, as well as the number of multisets which were not found. This represents 31542 calls to the Connection Machine.

The CM is here used in time-sharing, making the timing figures larger than they actually are, but this has no effect on the comparison.

STATISTICS for FML 12 (SELECTION 1) after 3106.391 secs.

Provable 3106.361
Node Opened: 653 Closed: 644 Max. Depth: 119
Idem_Reduct: 18 ExtDBse: 7317 SkipSelect: 155
To_Filter: 31481 Filtered: 2966
CM_DB: Thms: 41; Non_Thms: 7276; Not_Found: 24225

<table>
<thead>
<tr>
<th>Filters</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosNeg1:</td>
<td>2763</td>
<td>203</td>
</tr>
<tr>
<td>PosNeg2:</td>
<td>22</td>
<td>1192</td>
</tr>
<tr>
<td>Strict:</td>
<td>1549</td>
<td>1236</td>
</tr>
<tr>
<td>Rule_2:</td>
<td>1526</td>
<td>23</td>
</tr>
<tr>
<td>Matrix:</td>
<td>770</td>
<td>756</td>
</tr>
</tbody>
</table>
CHAPTER 6. EMPIRICAL RESULTS.

Pass : 770  2196
---------
Curry : 644  799

The second table concerns the same test, but this time without intelligent backtracking. In this case, we can see that, without this technique, the performance of the theorem prover is significantly reduced.

STATISTICS for FML 12 (SELECTION 1) after 3801.83 secs.
Provable 3801.810
Node Opened: 700  Closed : 648  Max. Depth : 102
AxiomClosed: 48  IntDBse: 14  NewTheorems: 648
Idem_Reduct: 25  ExtDBse: 9228  SkipSelect: 0
To_Filter: 4272  Filtered: 2931
CM_DB:  Filters: 46; Non_Thms: 9182; Not_Found: 33559
Pass  : 785  2146
---------
Curry : 648  1502

This shows that obviously, in this case, skipping redundant computations increases the efficiency by ≈ 25%. Nevertheless, as we already said, this will only happen if this technique finds an opportunity to apply, and this is not always the case. Moreover, one may wonder whether the use of large databases always improves the performances. There are cases where it is rather the contrary which happens. For example, the following table shows the effect of using a small database of ≈ 3k theorems in proving formula 5 of Asset on a Sun4 only.

STATISTICS for FML 5 (SELECTION 1, db3) after 3172.8 secs.
Provable 3172.810
Node Opened: 230  Closed : 218  Max. Depth : 51
AxiomClosed: 50  IntDBse: 239  NewTheorems: 218
Idem_Reduct: 99  ExtDBse: 210  SkipSelect : 40

STATISTICS for FML 5 (SELECTION 1) after 1234.6 secs.
Provable 1234.600
Node Opened: 251  Closed : 236  Max. Depth : 51
Clearly, the effect of using an external database will very much depend on the amount of information it contains that is of any use in the proof. Otherwise, the overhead of accessing the database may be too large and slow down the execution. If, on the other side, the information contained in the database plays a role in the proof, then the efficiency is of course increased. For example, comparing the statistics of the proof of formula 16 without databases on a Sun4 (see Table 1), with those obtained when using the 25k elements databases on the CM2, shows a significant improvement. (We only show the matrices that were used).

**STATISTICS for FML 16 (SELECTION 1, db25) after 414.2 secs.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provable</td>
<td>414.240</td>
</tr>
<tr>
<td>Node Opened</td>
<td>182</td>
</tr>
<tr>
<td>AxiomClosed</td>
<td>39</td>
</tr>
<tr>
<td>Idem_Reduct</td>
<td>124</td>
</tr>
<tr>
<td>To_Filter</td>
<td>2774</td>
</tr>
<tr>
<td>Filters</td>
<td>752</td>
</tr>
<tr>
<td>Filters Pass</td>
<td>203, 549</td>
</tr>
<tr>
<td>Filters Fail</td>
<td>392, 203</td>
</tr>
<tr>
<td>PosNeg1</td>
<td>732, 20</td>
</tr>
<tr>
<td>PosNeg2</td>
<td>7, 247</td>
</tr>
<tr>
<td>Strict</td>
<td>478, 261</td>
</tr>
<tr>
<td>Rule_2</td>
<td>392, 86</td>
</tr>
<tr>
<td>Matrix</td>
<td>203, 189</td>
</tr>
<tr>
<td>Max. Depth</td>
<td>77</td>
</tr>
<tr>
<td>NewTheorems</td>
<td>149</td>
</tr>
<tr>
<td>Closed</td>
<td>149</td>
</tr>
<tr>
<td>IntDBse</td>
<td>161</td>
</tr>
<tr>
<td>ExtDBse</td>
<td>0</td>
</tr>
<tr>
<td>SkipSelect</td>
<td>55</td>
</tr>
<tr>
<td>CM_DB Thms:</td>
<td>16; Non Thms: 4470; Not Found: 1093</td>
</tr>
</tbody>
</table>

**MATRICES**

<table>
<thead>
<tr>
<th>TV</th>
<th>R3</th>
<th>4V</th>
<th>SN</th>
<th>CH</th>
<th>6V</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>123</td>
<td>1</td>
</tr>
</tbody>
</table>

6.7.3 Conclusions on Massive Parallelism.

Massive parallelism presents numerous advantages. From a technical point of view, the CM2 has a very large memory and a large IO band-width. But the essential feature is that any test can be performed on all the elements of the database at once, and in constant time.
From the point of view of symbolic processing, the later characteristic is essential because it allows to perform the subsumption test, and any other test on all multisets in the database at once. As we have seen, this test may be expensive to perform sequentially on an entire database, even though some storage technique as the one used in Otter can be used, or if some smart method is used to avoid redundant computations can be devised as in MGTP [Ha91]. The same effect of efficient storage and retrieval, and avoidance of redundant computation is achieved with the CM2 without a too difficult programming effort.

Finally, intelligent backtracking can be implemented in such a way that it does not impede the forward execution of Prolog, and all information is always available and almost instantaneously. Developing further the use of the machine with update of the databases at runtime and increasing the in-memory processing, for example, the heuristic selections and partitions, on hard problems, speed-ups of several orders of magnitude can be expected.

---

1. See [addenda, note 13](#)
Chapter 7

Conclusion.

We have seen that given the complexity of the logic, it is nevertheless possible to obtain interesting results in hard cases. This requires not only the resources of parallel processing, but also and mainly, the discovery of good heuristics. We only found a few of these, but current research using massive parallelism is promising. No important progress will be made without a better understanding of the logics and of its properties. And parallel processing is required to process at once and in constant time large knowledge bases on which to apply the heuristics.

Parallel processing has proved to be one solution to our problem. This has not solved all problems, and there are still hard cases of course, but obviously, massively parallel processing is quite suitable to deal with symbolic manipulation in which arbitrary and complex data structures are constructed and the computation requirements are important.

More importantly, this suggests a completely different approach to the design of the theorem prover which comes close to the memory based reasoning model proposed by D. Waltz and C. Stanfil [Sw86][Wa90], in which a problem is solved by intensive use of memory. These authors note that on some applications tested on the CM2, like vision, natural language processing etc, speed-ups from 100 to 1000 times were obtained. Of course we are still very far from this sort of speed up unless we consider the ratio “no proof on a sequential machine/proof on the CM” as significant.

Since our investigations have only been carried out recently, we did not investigate further their suggestion. Nevertheless, adding to the operations already performed on the CM2 the matrix testing [Mc91] will be an additional step in the direction of the implementation of a memory-based model, and in the achievement of one of our goal, all intensive computation is left to the appropriate parallel hardware, the execution of the proof procedure, the inference engine, is left to Logic Programming.
Parallel logic programming is still in its infancy, and only a few fully developed compilers exist. Much research has still to be carried out to improve the efficiency of Prolog and of the Logic Programming languages in general. After many empirical tests and after the disappointments caused by a first naive approach, the overall impression is that the future of parallel processing is bright because there is no other way to speed up the execution of programs designed to tackle non-trivial problems or to deal with vast amount of information.

Nevertheless, even under the assumption that the problems one would like to investigate can be processed in parallel, efficient and appropriate algorithms have to be devised. There is no point in using large computing facilities to speed up a badly written program or an inappropriate algorithm.

It is well known that increasing the computational resources can only have a marginal effect on complex problems. An algorithm is often considered acceptable if its complexity grows polynomially with respect to the size of the input. This is clearly seen when comparing the growth of a polynomial function with that of an exponential function [Pa82], for example, for \( n^3 = 1000 \), and \( n^3 = 10^6 \), the corresponding growth of an exponential is \( 2^n = 1024 \), and \( 2^n = 1.27 \times 10^{30} \). In addition, the efficiency of a polynomial time algorithm increases further with the technological developments than that of an exponential algorithm. For example, if a function \( n^3 \) solves an instance of size \( 10^4 \) in one unit of time, on a computer ten times faster it will solve an instance of size \( 2.15 \times 10^4 \), while a function \( 2^n \) solving an instance of size 40 in the same unit of time would only solve an instance of size 43. [Pa82].

But, this is not the end of the story since there are counterexamples to this practical approach. The simplest one is, again, Fibonacci where restriction of redundant computation allows to run the same exponential algorithm in polynomial time.

Another example comes from our investigations with R. Whaley on how to use the Connection Machine for symbolic computation. A. Urquhart is rumored to have said that deciding the validity of a formula of classical logic containing a number of variables in the order of 40 (i.e. \( 2^{40} \) valuations) was, at the time, not so long ago, computationally impossible. Working out how to implement this problem on the CM2, it first appeared that Boolean evaluation of a formula in 40 variables would take approximately 4k hours, all in memory, using the largest CM. A few weeks later, R. Whaley who had improved his algorithm and implementation announced that this problem was solved in 2 hours. This result is only meant to give some idea of the computing resources which are now available. But apart from that, one may also wonder what's in an algorithm, and to what extent many best current algorithms cannot still be further improved. As a last example, our research in Automated Theorem Proving shows that considering the proof theory as
CHAPTER 7. CONCLUSION.

the algorithm to solve a problem, i.e. to prove or to refute formulae of a logic, improvements of the proof theory like the one proposed by P.Thistlewaite amounts to a significant lowering of the complexity of the algorithm in most cases. And limiting the amount of computation required in the execution of the algorithm by using the databases, heuristics etc... increases further its efficiency.

Finally, we can summarize some of the basics of parallel processing in Prolog. As we said already, the Aurora system is designed to operate in an almost transparent way with almost no intervention of the programmer. The earlier quotation of R. Lusk, “good Prolog programming is good parallel Prolog programming” reflects this philosophy, but as we said, some care has to be taken in declaring what to run in parallel and what to run sequentially, and in trying to single out the non-deterministic procedures most likely to create large amount of work. We already mention ed the approach advocated in [Bu88] which relies on the use of findall/3.

In addition, it may be possible to design a program such that any large sets of data which would be created incrementally and sparsely at runtime be generated as soon as possible and at once in order to increase the granularity and enhance the opportunities of parallel processing. This is sometimes called “granularity collection” [Ti91]. For example, rather than writing a procedure as: build(data), process-in-parallel(data), add-to-(data1), process-in-parallel(data1),..., one should try, wherever possible, to write something like build(data + data1), process-in-parallel(data + data1),... . Nevertheless, care should also be taken when the amount of data is limited to prevent the mobilisation of many workers when no parallel processing is required.

Finally, some programming tools like sophisticated debuggers, profilers, and some tracing facility are essential to develop efficient parallel programs. A tracing facility for Aurora, Wamtrace [Dil87] running under Suntools, was available at some earlier stage of the development of Aurora and was a very useful tool allowing to visualize the parallel execution of a program. We had no access to the Xwindow version which does not seem to be available yet.

With respect to the development of a parallel program, we may add some elementary remarks on what should be considered as good recipes for parallel processing in Prolog: the parallel execution of goals should be possible, the search tree should be balanced, otherwise it should have long branches, a solution should not be available in the leftmost part of the tree, some way to return the first solution must be available to prevent waiting for all the execution to the left of the successful worker to be completed. To that effect, Aurora support a sort of “cut”, “cavalier-commit” which allows to cut the search space as soon as one solution is found. This mechanism should prevent the formation of the “peaks” we referred to above.
CHAPTER 7. CONCLUSION.

There may be many other ingredients to add, but these are the most obvious one that we have encountered during our investigations.

Considering the use of Prolog as the main programming language in this research in Automated Theorem Proving in Relevant Logic, we have most often kept an optimistic attitude when facing problems specific to Prolog itself. As we have seen, some of these problems can be circumvented provided we rely on the appropriate technology, but this may just underline some of the shortcomings of Prolog-as-it-is when we come to tackle rather complex problems.

The successes obtained in the framework of this programming paradigm are numerous, but in the particular case of our application to theorem proving we have encountered many problems which would not even had appeared had we used a programming language like C. For example, lacking arrays or pointers, or handy and efficient library functions, these (or what they are intended to do) have to be explicitly coded in Prolog, and the way to do it is often far from being obvious.

For example, the suggestion made at the end of Note 12 is very easy to implement if we are dealing with graphs and if we try to find a shortest path to travel from one vertex to another one. Then the suggestion is very similar to standard textbook programming examples. But in theorem proving we are dealing with trees, not graphs, as [TMM88] emphasise. And this makes a big difference. If we want to implement this suggestion we must find ways to reorder the constituents of multisets and to move up and down the tree whenever we wish; we must be able to suspend the execution of the proof at some point, jump back up the tree, try some other candidates and open new branches, and finally we must still be able to resume the proof at the point we had left it if nothing else works. Surely, this is a challenge for Prolog and it would be easier to do in C. But this critique is just in accordance with the remarks of Wos and McCune [Wo89] who pointed out that Prolog and Automated Theorem Proving have both something to gain from their confrontation and that some form of symbiosis of both can be achieved.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


[Ko79] KOWALSKI, R., Logic for Problem Solving, New York, North Holland, 1979


BIBLIOGRAPHY


Note 1.
Let \( \mathbb{N}_n \subset \mathbb{N}_+ \) be the free commutative monoid with \( n \) generators and \( \mathbb{N}^m \subset \mathbb{N}_+ \) be the set of \( m \)-tuples of integers, that is, the free commutative monoid with \( m \) free generators written additively. Then, \( \langle \mathbb{N}^m, +, 0 \rangle \) is trivially isomorphic to \( \langle \mathbb{N}_n, \cdot, 1 \rangle \).
Characterizing relevant divisibility in the vector monoid \( \mathbb{N}^m \), where the \( a_i \) are sequences of elements of \( \mathbb{N}^m \), \( (i < m, 0 < m \leq \omega) \), i.e. \( m \)-tuples of integers, \( a_i \parallel b_i \) iff for all \( i \leq m \), \( a_i \leq b_i \) and \( a_i = 0 \) iff \( b_i = 0 \).

Note 2.
A possible interpretation is an interpretation provided that for all \( A, B \in F \), and for all \( a \in M \), the following truth condition is satisfied: \((A \rightarrow B)a \iff \forall x \ (Ax \Rightarrow Bax)\), where \( x \) ranges over \( M \) and \( \Rightarrow \) is a metalinguistic operator.
Given a possible interpretation \( I \), \( A \) is true at \( a \) on \( I \) iff \( Aa \); \( A \) is verified on \( I \) iff \( Al \); and \( A \) \( I \)-entails \( B \) iff \( \forall x \ (Ax \Rightarrow Bx) \).

Defining validity in the usual way, i.e. \( A \) is valid in \( M \) iff for all \( I, Al \); and \( A \) is valid iff for all \( M, A \) is valid in \( M \), will not suffice for \( R_I \). Indeed, not all theorems of \( R_I \) -and, in particular, the Contraction axiom- are verified on arbitrary interpretations. The set of admissible interpretations has thus to be restricted and this is done by imposing a hereditary condition: an interpretation is hereditary if for all \( M, \) for all \( A \in F \) and for all \( a, b \in M, a \parallel b & Bb \Rightarrow Ba \). The right notion of valuation in \( R_I \) is then a hereditary valuation, that is, a valuation for which the hereditary condition holds. And every interpretation inherits heredity from its determining valuation. Hence, in \( R_I \), for all \( A \in F, A \) is valid in \( M \) iff for all \( I \) s.t. \( I \) is hereditary in \( M, A \) is verified.

Note 3.
Let \( A \subseteq X \). Then, \( A_{\min} \), the set of \( A \)-minimal elements is the set of elements \( a \in X \) s.t. if \( a \in A, \) for all \( y \in A, \) \( y \not\leq a \). \( X \) satisfies the finite minimum condition (FMC) iff \( A \subseteq X, A_{\min} \) is finite. Then, \( D \) says that, for \( n \in \mathbb{N},, < \mathbb{N}_n,|_o > \) and \( < \mathbb{N}_n,|_r > \) satisfy the FMC.

Let \( A = \{a_i\}_{i \in \mathbb{N}} \). Then, \( X \) satisfies the Ascending Chain Condition (ACC) if \( A \subseteq X \) is an infinitely ascending chain, i.e. such that for all \( m, i \in \mathbb{N} \) and \( m < i, a_m < a_i \).
If \( a_m > a_i \), then \( A \subseteq X \) is an infinitely descending chain and \( X \) satisfies the descending chain condition (DCC).
As we will see, the partial order \( \leq \) on \( X \) is said well-founded, and \( > \) Noetherian iff each strictly descending sequence is finite.

In order to show that the \( IDP \equiv D \equiv K \) hold in the free commutative monoid \( \mathbb{N}^k \) partially ordered by divisibility, and so act as termination principle of the decision procedure for \( R_I \) in \( M \), Meyer first notes that for any partially-ordered set \( A \) (i.e. <
Next, if \( A, B \) are any partially ordered sets satisfying the IDP, then their product \( AB \) satisfies the DCC and the FMC. And taking \( X^k \) as the power of \( X \) for finite \( k \), defined by \( XX = X^2, X^{k+1} = X^kX \), \( X^k \) satisfies the IDP. It is then easy to show that the IDP, DCC and FMC hold in the partially-ordered set < \( N^n, \leq \) > and in its isomorphic copy < \( N_n, | >. \) Let \( k \in \mathbb{N}^+ \), the \( k \)th power of \( N \) in the vector monoid < \( N^k, +, 0 \) >, and let \( \leq \) be the natural order on \( N \), and \( \leq_r C \leq \) be such that \( 0 \leq_r a \) iff \( a = 0. \) Then, Meyer shows that < \( N^k, \leq \) > and < \( N^k, \leq_r \) > have the IDP. Hence each of their isomorphic copies < \( N^k, \cdot, 1 \) > have the property. Consequently, \( K, D \) and IDP are true for this commutative monoid.

Note 4.
In order to show that \( R_I \) has the FMP, for \( A \) a non-theorem of \( R_I \) of index \( k \), if \( A \) is refutable in < \( N^k, +, 0 \) > on \( I \), where \( I \) is hereditary, it suffices to find an interpretation \( I' \) in \( i^k \) refuting \( A \) and such that \( I' \) is hereditary. \( i^k \) being finite, \( R_I \) has the FMP.

Note that we only have a semi-decision procedure: every non-theorem of \( R_I \) is refutable in a finite model. We get a decision procedure by associating the other semi-decision procedure: finding a proof. The difference with other classical finite model theorems is that, here, there is no way to put an a priori bound on the value of \( k \).

Note 5.
We are in a situation similar to the second theorem of Higman: we have to show that the embedding function does not change the effect of the labelling. It is a simple adaptation of the preceding proof and definition.

Note 6.
The proof of theorem 2.2 is not quite complete. Either skip condition 3 and use 3 \( \rightarrow \) 1 with 2 \( \rightarrow \) 3 to show that 2 \( \rightarrow \) 1, or, more simply, keep condition 3 and skip the proof of the theorem which is a classical result anyway.

Note 7.
Actually the system provides an operator cavalier Predicate which executes Predicate unconditionally, and a commit operator "I" which prunes the branches to its left and right sides.

Note 8.
This heuristics cannot be considered as a general rule valid in all cases though. We arrived at it by examining many long and difficult proofs in an attempt to single out the reasons why some formulae are hard to prove.
The first explanation we found appeared to be the following. In some of these hard cases the selection of principal and parametric constituents based on a 'less costly first' ordering
of the candidates repeatedly selected the same principal, for example \((\sim a \circ \sim a)\) with some parametric constituents. During the search for a proof of \(\text{[Principal + Parameters]}\), Principal would eventually, in some sense, reproduce itself, i.e. application of the rules of the proof theory keeps on adding \(\sim a\) to the list of parameters to consider in the next proof attempt when the candidate passes through all filters and the \(K_0\)-rule finds no opportunity to apply. This explains why the original database of collected theorems and non-theorems contained many instances of "basic" multisets \(X, Y, Z, \ldots\), 'fission' whatever number of literals: \(X \vdash \sim a \vdash \sim a \cdots\). Subsequent improvements of the selection strategy prevented this situation from happening too often but we conjecture that this may be a major reason why some formulae are hard to prove. We gave an example of a such a formula on page 134. And, as we have seen, an important improvement resulting from this observation was to change the ordering of the candidates for a selection, the 'most costly' coming first. In this way, as the tables of results also show, some formulae difficult to prove under the former ordering are easy to prove under the second ordering.

But this change will not solve all problems of course. And we did not systematically investigated the conditions under which this "empirical heuristic principle" is correct. The reasons are first that in many cases it just happens that decomposing the complex constituents earlier generates the required information to find a short proof, but in many other cases, it is not so. Indeed, an optimal selection of principal will not depend on the complexity (determined by cost) of the constituents. Secondly, further research relying on the use of large databases led us to think (and this was one of the points to use databases) that this problem would show up rarely since most often the appropriate parameters would have been found in the databases. And if these were not found, then we could still consider to change the strategy. We will come back to this problem in Note 13.

A second explanation of why some formulae are hard and which justifies the heuristic is mentioned in the text. Given the order of application of the heuristic principles, considering again the former example of \((\sim a \circ \sim a)\) principal, the first heuristic rule starts by looking for the complement of \(\sim a\) in the parameters and if it is found we have then to prove the right constituent of principal with the rest of parameters. And this may require a lot of useless computation (see the table of possible partitions on pages 88-89) and still more if no proof is found and a new selection of parameters for the left constituent of principal has to be tried. Again, this second explanation is not always verified. The following is an example from a proof of formula 1 (below in this Note) which is hard to prove if the heuristics (in this example, search for the best left and the best right parameters only) are not applied.

```
|   | prove(+(+¬(a),&(b,+(¬(a),&(a,(b)))))),o(a,v(b,v(¬(b),o(¬(a),(a))))))|   |
|   | Princ : +(+(&(¬(b),a),¬a),b),+(o(v(o(¬(a),a),v(¬(b),b)))o(a),¬a)) | Para : [] |
|   | Princ : +o(v(o(¬(a),a),v(¬(b),b)),a),¬a) | Para : [(+(&(b,a),a),b)] |
| SELECT | Princ : +(+(¬(b),a),¬a),b | Para : [o(v(o(¬(a),a),v(¬(b),b))),a],¬a |
| Princ : +(¬(b),a),¬a | Para : [o(v(o(¬(a),a),v(¬(b),b))),a],¬a |
| SELECT | Princ : (¬(b),a) | Para : [o(v(¬(a),a),v(¬(b),b))),a],¬a |
```
Then, the execution proceeds deeper and deeper in the proof search tree as a few selections represented below show:

```
... Depth: 15 FUSION 1 L: v(o(a,a),v(b,b)) + []
Para: [a]
Depth: 15 FUSION 1 R: a + [a]
PROVED depth: 14 [o(v(o(a,a),v(b,b)),a),a]
SELECT Princ: v(o(a,a),v(b,b)) Para: [o(v(o(a,a),v(b,b)),a),a]
SELECT Princ: o(a,a) Para: [o(v(o(a,a),v(b,b)),a),a]
Depth: 17 FUSION 2 L: a + [o(v(o(a,a),v(b,b)),a)]
SELECT Princ: o(v(o(a,a),v(b,b)),a) Para: [o(v(o(a,a),v(b,b)),a),a]
SELECT Princ: o(a,a) Para: [o(v(o(a,a),v(b,b)),a),a]
SELECT Princ: o(a,a) Para: [o(v(o(a,a),v(b,b)),a),a]
SELECT Princ: o(a,a) Para: [o(v(o(a,a),v(b,b)),a),a]
Depth: 22 FUSION 2 R: a + [o(v(o(a,a),v(b,b)),a)]
Princ: v(o(a,a),v(b,b)) Para: [o(v(o(a,a),v(b,b)),a),o(a,a)]
SELECT Princ: o(a,a) Para: [o(v(o(a,a),v(b,b)),a),o(a,a)]
... The simple explanation in this case is that without heuristics the theorem prover has missed an easy and early proof of
Princ: o(v(o(a,a),v(b,b)),a) Para: [a,a,b]
while the use of heuristics provides such a proof (and a proof of the input formula) easily:
... as above ...
Bestparameter: [b]
SELECT Princ: v(o(a,a),v(b,b)) Para: [b]
SELECT Princ: v(b,b) Para: [b]
```
Axiom. \([b, -b]\)

\[
\text{Depth: 8 DISJ.2: } b + [-b]
\]

PROVED depth: 7

\[
[v([-b, b]), -b]
\]

Depth: 7 DISJ.2:

\[
v([-b, b]) + [-b]
\]

PROVED depth: 6

\[
[v(o([-a, -a]), v([-b, b])), -b]
\]

... 

BEST: \([-a]\)

Axiom. \([a, -a]\)

PROVED depth:

\[
[O(v(o([-a, -a]), v([-b, b])), a), -a, -a, -b]
\]

To conclude, to illustrate the above discussion and since we did not report them explicitly in the text, we give some results showing the effect of the heuristic selection of parameters in the application of the fusion rule.

The following table gives the timings obtained without heuristics (except for the ordering by cost); with the best selection of parameters for the right constituent of principal; with the later and the selection of best parametric constituent for the left element of principal. These results do not include the effect of using a database of “dualcomp” (p. 102) and concern the following three formulae of \textit{Impset} selected purely at random.\(^1\)

\begin{align*}
\text{Fml } 1 & : (\neg a + (b \& (\neg a + (a \& \neg b)))) + (a \circ (\neg b \lor (\neg a \circ \neg a))) \\
\text{Fml } 2 & : ((\neg a \circ \neg a) + (b \& (\neg a + \neg b))) + ((a + a) \circ (\neg b \lor (a \circ b))) \\
\text{Fml } 3 & : (\neg a \circ (\neg a + (a + (b \& \neg b)))) + (a \circ (a \lor (\neg a \circ (\neg b \lor \neg b))))
\end{align*}

\begin{tabular}{|c|c|c|c|}
\hline
Heuristics & Fml 1 & Fml 2 & Fml 3 \\
\hline
No Best & SEL 1 & >300 & 148 & 1.82 \\
 & SEL 2 & 37.0 & 161 & >300 \\
Best Right & SEL 1 & 1.79 & 247 & 1.79 \\
 & SEL 2 & 0.6 & 289 & 290 \\
Best Right + Left & SEL 1 & 1.84 & 23 & 1.12 \\
 & SEL 2 & 0.4 & 25.02 & >300 \\
\hline
\end{tabular}

\(^1\)Note that after translation these formulae are processed respectively as

\begin{align*}
&\neg a \lor (\neg b \& a) + \neg a \lor (\neg a \lor \neg b) \land (\neg b \lor (\neg b \land a)) \\
&((\neg b + a) \land (\neg a)) + (a \circ (\neg b \lor (\neg b \lor a) + (\neg a \lor \neg a)) \\
&{(\neg a \lor (\neg a \land (\neg a \lor a))) + (((\neg a \lor a) + (a \lor b) \land b) + (\neg a \land a))} \\
\end{align*}
Note 9.
As one of the examiners remarked, "refuted" here means any multiset that has failed the filters (excluding the Curry check). Indeed, multisets other than the origin multiset that fail the Curry check may be provable. The Curry property says that there is a shorter proof if there is any proof at all. Because of this, if the Curry check has eliminated any choice in any subtree of the proof search tree, and the multiset at the head of that subtree has been recursively failed then we cannot assume that it is unprovable. It may just be part of a subtree that provides a larger than necessary proof of an antecedent multiset.

Note 10.
Since our investigation dealt mainly with some specific fixed set of formulae we felt justified in carrying out this empirical research. This amounted to some general preprocessing of these formulae. It is with respect to these that the matrices are efficient and that the possible partitions of parameters are good. In general, of course, empirical research cannot solve the problem of selecting the right matrices as well as that of determining the likelihood of some formula being principal in a multiset. The beginning of a theoretical investigation of the matrices refuting strength can be found in [TMM88].

Note 11.
Note that the retrieval time is only constant for fixed-size, non-dynamic databases; and only if the number of data elements is less than or equal to the number of physical processors of course.

Note 12.
We can come back to the problem of the search strategy here. Most of our experiments relied on the depth-first strategy built into the Prolog engine. That means that, in general, no specific search strategy was specified in the program; the search is only directed by the use of heuristics which speed up the depth-first search in case some solution is found in this way. Otherwise, if no heuristics apply, the depth-first, left-to-right and backtracking on failure execution proceeds normally.

Contrary to widespread belief, other search strategies are relatively easy to implement in Prolog, at least in principle. Early attempts to implement breadth-first strategy (at each selection, make a list of all possible selections and try to prove them one level at a time) resulted in system overflow except in the case of the simple formulae of Standard. But, as expected, such a strategy required a much longer time than depth-first. Actually, using an ordering of the constituents of the multisets by cost and using some sort of evaluation function in the application of the heuristics (i.e. a proof will be found faster if such or such selection is made), we arrived at a program mixing several characteristics of other well-known search strategies like A* or Best-First.

It should be noted at this stage that the best evaluation function to use is not necessarily the selection of the heuristics which relies on the content of the databases or on some
approximate matching of the variables of principal and parameters, but it is rather to
calculate the McRobbie function which tells us the price to pay to open one or another
path in the proof search tree. This can be seen on a simple example:
prove((a \circ (c \circ \sim d)) + ((b \circ (d \circ \sim e)) + (\sim a + (\sim b + (\sim c + e))))
after translation, we get

Princ : +o(\sim e, o(b,d)), +o(\sim d, o(a,c)), +o(\sim c, +o(\sim b, +o(\sim a, e))))
Para : []

... 
Princ : +o(a, e) Para : [o(\sim d, o(a,c)), o(\sim e, o(b,d)), \sim b, \sim c]


> Princ : o(\sim e, o(b,d)) Para : [o(\sim d, o(a,c)), e, \sim a, \sim b, \sim c] McR: 243
  o prove [o(b,d), o(\sim d, o(a,c)), \sim a, \sim b, \sim c]
Princ : o(\sim d, o(a,c)) Para : [o(b,d), \sim a, \sim b, \sim c]
  o prove [o(b,d), \sim a, \sim b, \sim c]
Princ : o(b, d) Para : [\sim b, \sim d]
  o prove [o(a,c), \sim a, \sim c]
Princ : o(a, c) Para : [\sim a, \sim c]

Provable

Under the second selection of principal we obtain

> Princ : o(\sim d, o(a,c)) Para : [o(\sim e, o(b,d)), e, \sim a, \sim b, \sim c] McR: 243
  o prove [o(\sim e, o(b,d)), e, \sim a, \sim b, \sim d]
Princ : o(\sim e, o(b,d)) Para : [e, \sim b, \sim d]
  o prove [o(b,d), \sim b, \sim d]
Princ : o(b, d) Para : [\sim b, \sim d]
  o prove [o(a,c), \sim a, \sim c]
Princ : o(a, c) Para : [\sim a, \sim c]

Provable

In the first case the McRobbie function gives us 342 possible combinations to try and
the proof requires 57 tries. In the second case, we obtain respectively 288 combinations
and 48 tries. This is a toy example, but it shows clearly that the evaluation of the cost
given by the McRobbie function should be taken into account. Moreover we may note
that in this case the heuristics as they are implemented would make the wrong choice.
When o(\sim d, o(a,c)) is selected as principal, since there is no complement of \sim d in the
parameters, o(\sim e, o(b,d)) is tried instead.

Coming back to the search strategy, here is the simplest way to implement a bounded
search and an iterative deepening search. Both were tried and provided the shortest
proofs for Standard. In all cases -as well as for tests on Impset formulae-, it took longer
than with depth-first with heuristics and databases.
prove(Multiset) :- prove(Multiset, Bound). % Bound is whatever bound value chosen
prove(Multiset, Bound) :- prove_bounded_depth_first(Multiset, Bound).
prove_bounded_depth_first(Multiset, Bound) :- Bound >= 0,
                   provable(Multiset, Bound).
provable(Multiset, Bound) :- NewBound is Bound - 1, NewBound >= 0,
                         select(Multiset, Principal, Parameters),
                         apply_rules(Principal, Parameters, NewBound).

A simple bounded search will find a proof if it is inside the specified bounds. Otherwise no proof will be found. The solution is to increment the bound on failure. Iterative deepening does just that in adding a condition to increase the bound on failure due to the originally chosen bound being too small.

prove(Multiset, Bound) :- NewBound is Bound + 1,
                   prove(Multiset, NewBound).

If there is a proof or a refutation we are assured that it will now be found. But it may take time since on failure iterative deepening restarts from the beginning.

A better solution which was not implemented but that seems feasible is to keep the heuristically directed depth-first search and when some branch appears to be too hard, to allow back jumps to some former selection that created that branch. This implies to keep an agenda of all possible candidates at each selection (findall/3). In order to know when to jump back, as suggested by one examiner, we can check the selections made and if some constituent is repeatedly selected as principal -showing that branch to be too hard-, we can decide then to jump back. In order to know where to jump, since any new selection increases the depth of the tree by one, the depth can be used to flag the selections and we could then jump to the depth of selection of the recalcitrant principal and try another candidate which is recorded in the agenda.

**Note 13.**
The implementation of the updating of the databases at runtime was started in joint work with John Barlow who wrote the C* code. This is being completed and debugged at the time of finishing this thesis. The heuristic selection and partition was prototyped in Prolog and first results on a small sample of multisets allow us to predict important speed-ups on the CM2 provided all the appropriate information is already stored in memory. The C* code is also written by John Barlow. It is an adaptation of the subsumption test program but the code still requires debugging at this stage.