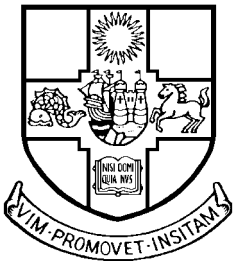
Analysis and Transformation of Proof Procedures

David André de Waal



A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

October 1994

Abstract

Automated theorem proving has made great progress during the last few decades. Proofs of more and more difficult theorems are being found faster and faster. However, the exponential increase in the size of the search space remains for many theorem proving problems. Logic program analysis and transformation techniques have also made progress during the last few years and automated theorem proving can benefit from these techniques if they can be made applicable to general theorem proving problems. In this thesis we investigate the applicability of logic program analysis and transformation techniques to automated theorem proving. Our aim is to speed up theorem provers by avoiding useless search. This is done by detecting and deleting parts of the theorem prover and theory under consideration that are not needed for proving a given formula.

The analysis and transformation techniques developed for logic programs can be applied in automated theorem proving via a programming technique called meta-programming. The theorem prover (or the proof procedure on which the theorem prover is based) is written as a logic meta-program and the theory and formula we wish to investigate becomes the data to the meta-program. Analysis and transformation techniques developed for logic programs can then be used to analyse and transform the meta-program (and therefore also the theorem prover).

The transformation and analysis techniques used in this thesis are partial evaluation and abstract interpretation. Partial evaluation is a program optimisation technique whose use may result in gains in efficiency, especially when used in meta-programming to "compile away" layers of interpretation. The theory of abstract interpretation provides a formal framework for developing program analysis tools. It uses approximate representations of computational objects to make program dataflow analysis tractable. More specifically, we construct a regular approximation of a logic program where failure is decidable. Goals in the partially evaluated program are tested for failure in the regular approximation and useful information is inferred in this way.

We propose a close symbiosis between partial evaluation and abstract interpretation and adapt and improve these techniques to make them suitable for our aims. Three first-order logic proof procedures are analysed and transformed with respect to twenty five problems from the Thousands of Problems for Theorem Provers Problem Library and two new problems from this thesis. Useful information is inferred that may be used to speed up theorem provers.

Declaration

The work in this thesis is the independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of education.

D.A. de Waal

Acknowledgements

This thesis is the realisation of a dream to become a *Doctor of Philosophy*. Numerous people and institutions provided support during this adventure into the unknown. First I would like to thank the Potchefstroomse Universiteit vir Christelike Hoër Onderwys that lured me into an academic career and has been supporting me ever since. Professor Tjaart Steyn needs special mention as he "looked after me" during my undergraduate and some of my postgraduate studies.

During 1988, Professor Ehud Shapiro offered me an opportunity to work at the Weizmann Institute of Science. There I met my advisor, John Gallagher, who made research look so interesting and relaxing that I had to follow. With his knowledge and leadership, doing research turned out to be truly interesting. However, it was not all that relaxing and I needed to be rescued several times from my Ph.D. blues.

Finally I would like to thank my family and friends who have all supported me during my stay in this foreign land. In particular, I would like to thank my wife Annetjie for being prepared to leave family and friends behind to follow our dreams.

This research was partly supported by the ESPRIT Project PRINCE(5246).

Contents

List of Figures

List of Tables

List of First-Order Theories

List of Programs

Chapter 1

Introduction

> The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.[1]
>
> K. GÖDEL

Progress towards the realisation of the statement above has been made during the last few decades. Automatic theorem provers are proving more and more complicated theorems and the execution times needed to prove established theorems are decreasing [45, 97, 12, 100, 7]. The reduction in search effort achieved leading to these improvements is the result of refinements in at least the following two areas: the proof methods (e.g. refutation theorem proving, Hilbert-style, sequent style, semantic tableaux, etc.) and the proof search strategies (e.g. for refutation theorem proving these include resolution, adding lemmas, forward chaining, backward chaining, etc.).

Many refutation theorem proving strategies for first-order logic produce search spaces of exponential size in the number of clauses in the theorem proving problem. As more difficult theorems are tackled, the time that these theorem provers spend tends to grow out of control very rapidly. Faster computers will allow a quicker search which may alleviate this problem. However, the inherent exponential increase in the size of the search space remains for many theorem proving problems. Refinements in the proof methods and/or the proof search strategies therefore also still need to be investigated. These refinements may allow even more theorems to be proved automatically, new theorems to be discovered and proofs to existing theorems to be found faster.

The approach taken in this thesis is to make automatic optimisations of general proof procedures with respect to a specific theorem proving problem. Given some theorem prover,

[1] Quoted from the QED Manifesto [3].

theory and formula (possibly a theorem), we wish to identify parts of the theorem prover and theory not necessary for proving the given formula. The identification of "useless" parts of the theorem prover and theory may be used to

- speed up the theorem prover by avoiding useless search and
- gain new insights into
  - the workings of the theorem prover and
  - the structure of the theory.

The layout of the rest of this chapter is as follows. First we define the theorem proving problem we wish to investigate. Second we indicate how we intend to identify "useless" parts of the theorem prover and theory and last we give the organisation of the rest of this thesis.

1.1 First-Order Logic and Refutation Theorem Proving

Before we introduce the theorem proving problem in which we are interested, we further motivate the need for the identification of useless parts of a theory and useless parts of a theorem prover. Let us for the moment assume that our theorem prover contains inference rules and our theory consists of axioms. These terms will be defined later on in this section.

Given some theory T, different versions (variations) of T are commonly used in theorem proving problems. These versions include:

1. taking some part of T, say T', targeted at proving certain theorems of T (we call this an incomplete axiomatisation of T);
2. adding lemmas to T, to simplify proofs of certain theorems (we call this a redundant axiomatisation of T).

The use of complete axiomatisations (in the sense that an axiomatisation captures some closed theory) is preferable to incomplete or redundant axiomatisations, as these axiomatisations may not reflect the "real world" we are trying to model. However, complete axiomatisations may increase the size of the search space as some axioms may not be needed to prove a particular theorem.

Automatic identification of axioms that are not needed to prove a theorem is desirable, as removal of these axioms may allow a more efficient search for a proof. Furthermore, removal of such axioms might indicate a minimal set of axioms required to prove the theorem (we usually have to be satisfied with a safe guess of which axioms are required as this property is in general undecidable).

The identification of useless inference rules may indicate that the theorem proving problem we are considering may be solved with a simpler (and perhaps more efficient) theorem prover or proof procedure. This may also indicate the existence of subclasses of theories where the full power of the proof procedure is not needed. Wakayama defined such a subclass of first-order theories, called CASE-Free theories, where an input-resolution proof procedure is sufficient to prove all theorems in theories that fall into this class. If such subclasses of a theory can be defined and instances of these subclasses occur regularly as theorem proving problems, it might be worthwhile to prove properties about the proof procedure for these subclasses. This may allow a simpler proof procedure to be automatically selected when a problem is recognised that is an instance of a defined and analysed subclass. This realisation that many problems do not always need the full power of the theorem prover led to the development of the near-Horn Prolog proof procedure of Loveland [69] and the Linear-Input Subset Analysis of Sutcliffe [103]. The methods we develop may also be used to do similar analyses. In a later chapter we give automatic proofs of some of the theorems about definite theories (Horn clause logic programs).

We now introduce the theorem proving problem we want to investigate in the rest of this thesis. The goals of automated theorem proving are:

- to prove theorems (usually non-numerical results) and
- to do it automatically (this include fully automatic proofs and "interactive" proofs where the user guides the theorem prover towards a proof).

The foundations of automated theorem proving were developed by Herbrand in the 1930s [49]. The first program that could prove theorems in additive arithmetic was written by M. Davis in 1954 [67]. Important contributions to the field were made by Newell, Shaw and Simon [81] in proving theorems in propositional logic. Gilmore [43] proved theorems in the predicate calculus using semantic tableaux. In the first part of the 1960s Wang [108] proved propositional theorems and predicate calculus theorems using Gentzen-Herbrand methods. Davis and Putnam [21] proved theorems in the predicate calculus and worked with conjunctive normal form. Prawitz [87] made further advances by replacing the enumeration of substitutions by matching. Most of the early work on resolution was done by Robinson [91] and on semantic tableaux by Smullyan [98], also in the 1960s. The use of rewrite rules

was developed by Knuth and Bendix in 1970 [58].

There are many textbooks and articles that can be consulted for an introduction to first-order logic and automated theorem proving; [91, 11, 67, 33, 17] amongst others give good introductions. We closely follow [91] in our presentation because it gives a concise and clear statement of the important issues surrounding the subject. There are more recent and more comprehensive introductions to automated theorem proving, but as the intention of this thesis is not to develop new theorem provers, but to optimise proof procedures on which existing theorem provers are based, we feel that a broad statement of the aims of automated theorem proving is in order.

We analyse and transform proof procedures for the first-order predicate calculus that can prove logical consequence. As the specialisation methods we are going to present in the following chapters do not depend on whether or not quantifiers are used in our first-order language, we present the quantifier free form to simplify the presentation. Davis and Putnam [21] introduced this "standard form" which preserves the inconsistency property of the theory. We return to this point later on in this section. The conversion of an arbitrary formula to a quantifier free form can be found in [63].

First, we introduce the *first-order predicate calculus* and second, we define a *theorem proving problem*.

DEFINITION 1.1.1 **alphabet**
An **alphabet** *is composed of the following:*

(1) variables: $X, Y, Z, \ldots$;

(2) constants: $a, b, c \ldots$;

(3) function symbols: $f, g, h, \ldots$, *arity* > 0;

(4) predicate symbols: $p, q, r, \ldots$, *arity* ≥ 0;

(5) connectives: $\neg$, $\vee$ *and* $\wedge$;

(6) punctuation symbols: "(" , ")" and ",".

This "non-standard" convention [64, 67, 33] for the representation of variables and predicate symbols was chosen so that we have only one representation throughout the whole thesis. Discussions, proof procedures presented as meta-programs, and programs generated by specialisation share the same representation. This simplifies the presentation and avoids errors in transforming programs generated by transformation and analysis to another representation used in discussions.

DEFINITION 1.1.2 **term**
A **term** *is defined inductively as follows:*

(1) a variable is a term;

(2) a constant is a term;

(3) if f is a function symbol with arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

DEFINITION 1.1.3 **formula**
A **formula** *is defined inductively as follows:*

(1) if p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula (called an atomic formula or atom);

(2) if f and g are formulas, then so are $(\neg f)$, $(f \wedge g)$ and $(f \vee g)$.

DEFINITION 1.1.4 **literal, clause**
A **literal** *is an atom or the negation of an atom. A* **clause** *is a formula of the form $L_1 \vee \ldots \vee L_m$,*[2] *where each L_i is a literal.*

DEFINITION 1.1.5 **first-order language**
The **first-order language** *given by an alphabet consists of the set of all formulas constructed from the symbols of the alphabet.*

It is important to work out the meaning(s) attached to the formulas in the first-order language to be able to discuss the truth and falsity of a formula. This is done by introducing the notion of an interpretation which we introduce informally (see [64, 67] for formal descriptions).

An interpretation (over D) consists of:

(1) some *universe of discourse* D (a non-empty set and also called a *domain of discourse*) over which the variables in our formulas range,

[2]Because we have a quantifier free language, there is an implicit universal quantifier in front of this formula for every variable $X_1, \ldots, X_s$ that occurs in $L_1 \vee \ldots \vee L_m$.

(2) the assignment to each constant in the alphabet of an element in D,

(3) the assignment of each n-ary function symbol in the alphabet of a mapping on $D^n \rightarrow D$ and

(4) the assignment of each n-ary predicate symbol in the alphabet of a relation on D^n.

The meaning of $\neg$ is negation, $\vee$ disjunction and $\wedge$ conjunction. Assuming every variable is universally quantified, every formula may be evaluated to either true or false. The first-order language can now be used to state our theorem proving problem.

DEFINITION 1.1.6 **logical consequence**
A statement B is a **logical consequence** *of a set of statements S if B is true whenever all the members of S are true, no matter what domain the variables of the language are thought to range over, and no matter how the symbols of the language are interpreted in that domain.*

DEFINITION 1.1.7 **theorem proving problem**
A **theorem proving problem** *has the form: show that B is a logical consequence of $\{C_1, \ldots, C_n\}$, where $C_1, \ldots, C_n$ and B are statements (formulas).*

$C = \{C_1, \ldots, C_n\}$ are also called axioms. If B is a logical consequence of C, B is a theorem of that theory. The problem now is to prove that the given theorem follows from the given axioms. For this we need a proof procedure. We want proof procedures that can deal with logical consequence and are suitable for computer implementation.

DEFINITION 1.1.8 **proof procedure**
A **proof procedure** *is an algorithm that can be used to establish logical consequence.*

A proof procedure contains a finite number of inference rules.

DEFINITION 1.1.9 **inference rule**
An **inference rule** *I maps a set of formulas into another set of formulas.*

Church [18] and Turing [105] independently showed that there is no general decision procedure (we call it a proof procedure) to check the validity of formulas for first-order logic (a formula is valid if it is true by logic alone: that is, it is true in all interpretations). However,

there are proof procedures that can verify that a formula is valid. For invalid formulas, these procedures in general never terminate.

In 1930 Herbrand [49] developed an algorithm to find an interpretation that can falsify a given formula. Furthermore, if the formula is valid, no such interpretation exists and his algorithm will halt after a finite number of steps. This idea forms the basis of proof procedures used for refutation theorem proving: the negation of the formula we want to prove $\neg B$ is added to our theory $(C_1 \wedge \ldots \wedge C_n)$ and if we can prove that $\neg(C_1 \wedge \ldots \wedge C_n \wedge \neg B)$ is valid, B is a logical consequence of our theory and therefore a theorem.

The specialisation methods we develop are primarily aimed at chain format linear deduction systems [67, 60, 68, 103]. In linear deduction systems the result of the previous deduction must always be used in the current deduction. In chain format systems, resolvents (see Section 3.2 for more detail) are augmented with information from deductions so far to make further deductions possible. However, our methods are not restricted to chain format linear deduction systems.

In addition to the restrictions given so far, we also assume that the first-order theory we are interested in consists of a finite number of axioms (formulas). Theories such as a theory of addition defined by an infinite number of equations over integers are therefore excluded. This restriction will be waived in a later chapter when we prove properties about proof procedures, first-order theories and formulas.

1.2 Specialising Proof Procedures

In the introduction we indicated that we would like to identify "useless" parts of the theorem prover and theory. The problem now is: how do we get the theorem prover in an analysable form so that we may use our analysis tools to identify useless inference rules in the proof procedure and useless formulas in the theory. One solution lies in the use of meta-programming.

A meta-program is a program that uses another program (the object program) as data [51]. Meta-programming is an important technique and many useful applications such as interpreters, knowledge based systems, debuggers, partial evaluators, abstract interpreters and theorem provers have been developed that use this technique with various degrees of success. The theorem prover is written down as a meta-program in some programming language L (we use a logic programming language in this thesis). The data for this meta-program is the theory (also called the object theory) we wish to consider. The meta-program may now be analysed with analysis tools for language L.

The theory of abstract interpretation provides a formal framework for developing program analysis tools. The basic idea is to use approximate representations of computational objects to make program dataflow analysis tractable [20]. However, meta-programs have an extra level of interpretation compared to "conventional" programs. This overhead caused by the extra level of interpretation can be reduced by a program specialisation technique called partial evaluation (also sometimes referred to as partial deduction) [54].

Partial evaluation takes a program P and part of its input In and constructs a new program P_{In} that given the rest of the input $Rest$, yields the same result that P would have produced given both inputs. The origins of partial evaluation can be traced back to Kleene's s-m-n theorem [57] and more recently to Lombardi [66], Ershov [31], Futamura [35] and Komorowski [59].

Some of the available transformation and analysis techniques need improving or changing to make them suitable for our aims. These improved partial evaluation and abstract interpretation techniques are developed in the following chapters.

We therefore propose a specialisation method based on global analysis and transformation of a proof procedure, object theory and formula. Our aim is to specialise a given proof procedure with respect to some object theory and formula. The diagram in Figure 1.1 illustrates our approach: proof procedures from automatic theorem proving, written as meta-programs in some programming language, are transformed and analysed with extended partial evaluation and abstract interpretation techniques. The transformation and analysis are done with respect to some chosen theory and formula taken from some calculus (the calculus has to be the calculus for which the proof procedure is written—we concentrate on first-order logic). The specialised proof procedure may be seen as compiling the object theory into the programming language in which the proof procedure is written. The central idea is to use information coming from the proof search itself to eliminate proof attempts not excluded by the proof procedure. This gives a problem-specific specialisation technique and contrasts with other problem-independent optimisations present in many theorem provers [100, 45, 97].

The proposed specialisation method can be elegantly expressed using the first Futamura projection [35], which is a well known transformation technique based on partial evaluation. Informally, it states that the result of restricting an interpreter for some language L to a particular program P, is a specialised interpreter that can only interpret one program P. In this thesis the interpreter will be a proof procedure. The specialised interpreter may be regarded as a compiled version of P. In our approach, partial evaluation is only one of the specialisation techniques used. It is augmented with renaming, abstract interpretation and a simple decision procedure to form a specialisation technique capable of powerful

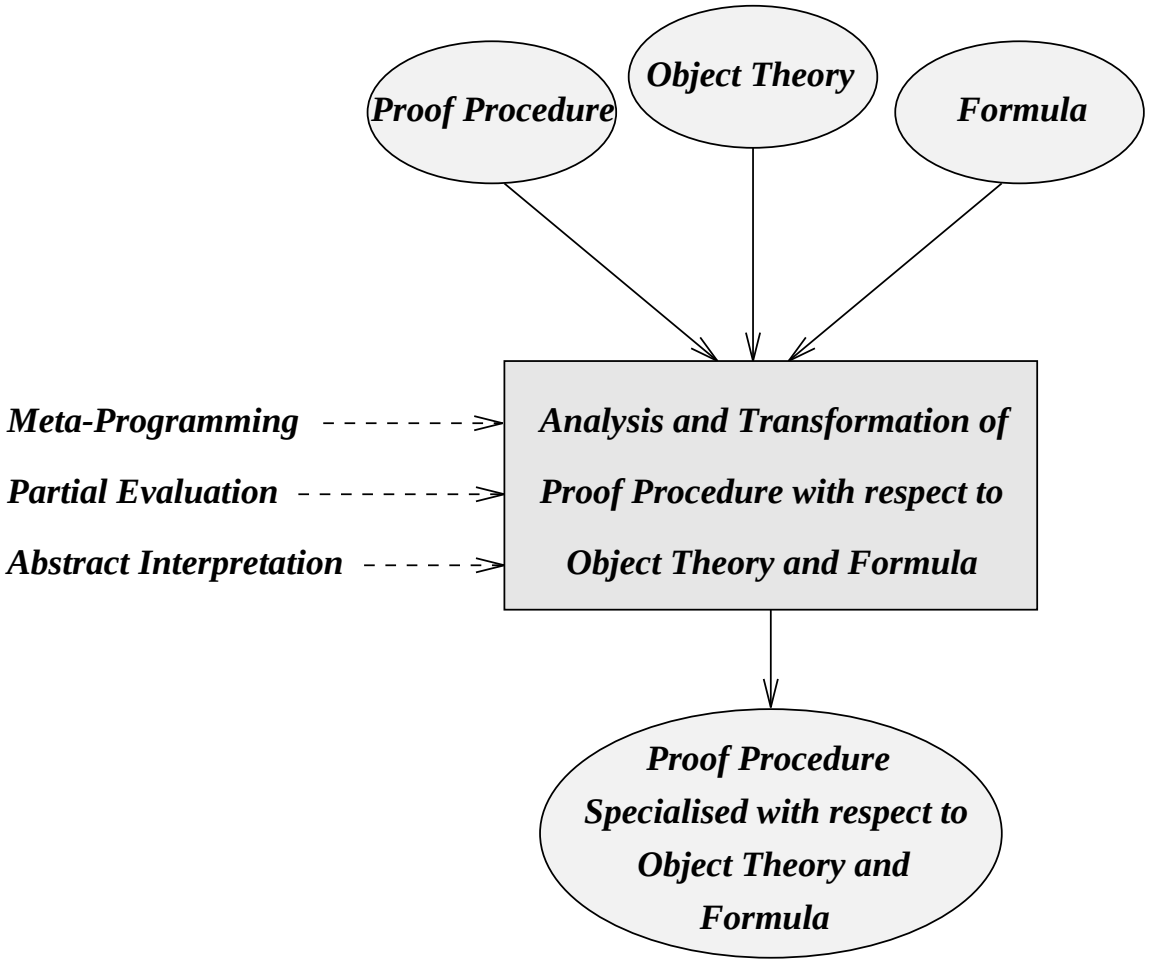


Figure 1.1: Specialisation of Proof Procedures

specialisations such as replacing infinitely failed deductions by finitely failed deductions.

Although similar research has been done that involved some of the components in Figure 1.1, we believe that the specialisation potential when combining theorem proving with meta-programming, partial evaluation and abstract interpretation has not been realised or demonstrated. The specialisation techniques may also pave the way to exploit more of the potential offered by global analysis and transformation techniques.

1.3 Organisation of Thesis

This thesis is organised as follows. First-order logic proof procedures presented as meta-programs are the subject of Chapter 2. Three proof procedures are presented: model elimination, naive near-Horn Prolog and the semantic tableau. Two versions of model elimination are presented: one version using the non-ground representation and the other using a ground representation. The other two proof procedures are represented using only a non-ground representation.

In Chapter 3 we review the current state of partial evaluation and point out some of the problems that have not been solved satisfactorily. We discuss some of the limitations of partial evaluation and propose a new unfolding rule that simplifies the analysis phase that is going to follow. This unfolding rule produces a partially evaluated program that is amenable to further specialisation. We then propose a simplified partial evaluator more suited for the specialisation of proof procedures.

Chapter 4 states the requirements needed for detecting "useless" clauses. We introduce regular approximations and give details of how to construct a regular approximation of a logic program. We then show how regular approximations can be used to detect useless clauses. This chapter ends with a discussion of "query-answer" transformations that are used to increase the precision of the regular approximation.

In Chapter 5 we combine the extended partial evaluation and abstract interpretation techniques developed in the previous chapters into a specialisation technique that reasons about safe over-approximations of proof procedures, object theories and queries. In Chapter 6 the use of these specialisation methods is demonstrated. The three procedures described in Chapter 2 are specialised and useful information is inferred.

Chapter 7 extends the proposed specialisation method with under-approximations. This results in an alternative approach to the specialisation of proof procedures that combines over- and under approximations. The rewriting of a proof procedure into a form more amenable to partial evaluation is described in Chapter 8. This is applied to one of the model elimination proof procedures presented in Chapter 2. Finally, Chapter 9 contains conclusions, further comparisons with other work and ideas for future research.

Chapter 2

Proof Procedures as Meta-Programs

In this chapter we present three first-order logic proof procedures as meta-programs, namely model elimination, naive near-Horn Prolog and semantic tableaux. The first two proof procedures will be used in the demonstration of our transformation and analysis techniques. A variation of the semantic tableau proof procedure will also be specialised to demonstrate the flexibility of our techniques. All or parts of each of these three proof procedures are used as the basis of one or more state-of-the-art theorem provers [100, 70, 12]. However, before we introduce the proof procedures, we discuss the meta-programming aspect of the proposed method in more detail.

2.1 Meta-Programming in Logic Programming

The specialisation techniques introduced in Figure 1.1 were not restricted to a particular programming language. However, for the proposed specialisation method to succeed, the partial evaluator has to be able to specialise programs in the language in which the meta-program is written (the meta-language). Furthermore the abstract interpreter has to be able to analyse programs generated by the partial evaluator. We chose "pure" logic programming as defined by Lloyd [64] as the common language for our methods. This frees us from most language dependent detail and "forces" us to develop general techniques not relying on any implementation for the success of our methods.

The theoretical foundations of meta-programming in logic programming did not receive much attention until it was given a sound theoretical foundation by Hill and Lloyd in

[52]. They showed that for meta-programs to have clear semantics, types (also called sorts) needed to be introduced into the language underlying the meta-language. In this thesis we do not give the type definitions relevant to our meta-programs, as our analysis and transformation techniques do not depend on them. However, we give declarative semantics to the procedures we develop. In any "real" implementation of our analysis and transformation techniques, for instance in the logic programming language Gödel, we expect a typed representation to be used. This will ensure a clear declarative semantics for all the meta-programs presented in this thesis. Martens [73] also investigated the semantics of meta-programming.

Various representation issues have also been clarified by Hill and Lloyd. There are two representations that can be used to represent object-programs: the non-ground representation, where object level variables are represented by variables at the meta-level, and the ground representation, where object level variables are represented by constants at the meta-level. We briefly discuss the two representations in the following sections.

2.1.1 Non-Ground Representation

The *non-ground* representation, where object level variables are represented by meta-level variables at the meta-level, was the first representation used for meta-programming in logic programming [51, 99, 22, 79]. The so called "vanilla" interpreters were then developed to demonstrate the power and usefulness of the meta-programming approach.

Certain meta-logical predicates, such as *var*, however presented semantic problems which made it impossible to give declarative semantics to these predicates. This complicated the analysis and transformation of logic programs using these non-declarative features to such an extent that the ground representation (discussed in the next section) was introduced into logic programming to overcome these semantic problems.

The semantic problems associated with the non-ground representation can be reduced by only using "pure" predicates (that is user-defined predicates and system predicates that do not destroy our declarative semantics). To make our specialisation techniques applicable to both representations, we therefore restrict our methods to "pure" logic programs.

Regardless of the semantic problems associated with the non-ground representation, it was an important stepping stone in the history of meta-programming in logic programming and is suited for the demonstration of analysis and transformation techniques, because of its simplicity.

2.1.2 Ground Representation

The ground representation was introduced in logic programming to overcome the semantic problems associated with some non-declarative predicates in the non-ground representation. Object level variables are represented by ground terms at the meta-level. It is then possible to give declarative definitions to the predicates that presented semantic problems in the non-ground representation.

The specialisation of meta-programs using the ground representation in Gödel was studied in detail by Gurr [46]. An important issue in the specialisation of meta-programs using a ground representation is the specialisation of a "ground unification algorithm" (an algorithm that unifies terms written in a ground representation). Two approaches to solving this problem were presented by Gurr [46] and de Waal and Gallagher [24]. The approach Gurr took towards the specialisation of a ground unification algorithm was to introduce WAM-like predicates [109] in the definition of *Resolve* (a Gödel predicate that resolves an atom in the current goal with respect to a statement selected from an object program) to facilitate effective specialisation of ground unification. De Waal and Gallagher used general analysis and transformation techniques to specialise a ground unification algorithm and the algorithm was treated just as another logic program (the fact that it did ground unification was of no importance to the specialiser). The ground unification algorithm was also written in a very intuitive and general way without using special predicates to aid specialisation. In this thesis we therefore assume that it is possible to specialise a ground unification algorithm with respect to all arguments that may occur in our meta-program effectively. The method by which this is achieved is not important.

2.1.3 Representation Independence

It is not the aim of this thesis to promote the aims of any logic programming language or representation and we therefore try to avoid representation issues as far as possible. We want to develop analysis and transformation techniques that are independent of the representation used to implement the meta-program: the techniques should therefore be general and not depend on any one feature of a representation for its success.

Unfortunately it is not always possible to avoid representation issues as a meta-program has to be presented using some representation. We will therefore specialise the model elimination proof procedure using both the ground and non-ground representations and show that the proposed specialisation methods can specialise the same program written in both representations giving similar specialisation results. However, we do not claim that the complexity of specialising the two representations is the same: the ground representation is

usually the more complex representation to specialise as a unification algorithm, renaming, substitutions handling, etc. has to be specialised as well. Only the non-ground representation will then be used further on in this thesis for the demonstration of the transformation and analysis techniques.

Our aim therefore is to build upon established work and extend current analysis and transformation techniques beyond the specialisation of unification and removing the overhead caused by the use of meta-interpreters. We will also show that by using a *depth abstraction* similar to that described in [39, 16], we can ignore the specialisation of unification of object level terms in the ground representation altogether. We lose some specialisation power by doing this, but are still able to achieve specialisation results that were previously only obtainable by problem-specific analysis techniques [85].

2.2 Representing a Proof Procedure as a Meta-Program

Many refinements of some basic strategies used in proof procedures may exist. For instance, the Prolog Technology Theorem Prover [100] has many refinements implemented to make the basic model elimination proof procedure on which it is based more efficient. Similarly, many refinements to resolution have been suggested. The appendix of [67] summarises twenty-five variations. In choosing our proof procedures for analysis and transformation, we have at least two options.

- Choose the simplest (shortest) sound and complete version of the basic procedure we are interested in that can be written as a declarative meta-program.
- Choose the most efficient (fastest) version of the proof procedure we are interested in that is sound and complete (and that can be written as a declarative meta-program).

Our experience with implementing and analysing various proof procedures, ranging from intuitionistic linear logic to constraint logic programming, has shown that analysing the simple basic version may hold the most promise. The reasons for this are as follows.

- Our analysis does a global analysis of the proof procedure, theory and formula. The inferred information allows optimisations different from that present in most proof procedures. The analysis of a complex version of the basic procedure may therefore be a waste of time.
- Because the meta-program is simple, it may allow a fast transformation and analysis. Furthermore, as our aim is to show an overall reduction in the time needed to prove

a theorem, a short analysis and transformation time is recommended.

- The inferred information is applicable to other variations of the proof procedure as well (because implementation dependent optimisations are not present).
- It may not be possible to write the complex version of the proof procedure as a declarative meta-program, because it may depend on non-declarative features of a particular language implementation for its success.

In this thesis we concentrate on the writing of the proof procedure as a definite logic program. The reason for this is that during the analysis phase we do not include analysis information from negation as failure steps in our approximation (this is to get a monotonic fixpoint operator for our approximation). All negation as failure steps are therefore ignored in the analysis phase and all information conveyed by negation as failure steps is lost. That is, normal programs can be analysed but not very precisely.

The first proof procedure we present is model elimination in the MESON format [67, 85]. It is an example of a chain format linear deduction system and is a sound and refutation complete proof procedure for clausal first-order logic. This proof procedure is presented using both the non-ground and ground representations respectively. In Chapter 6 we obtain new properties of this procedure.

The second proof procedure we present is naive near-Horn Prolog [69]. We present this "naive" version of near-Horn Prolog because it raises some interesting questions as it is sound, but incomplete. Naive near-Horn Prolog is also a linear deduction system.

The third proof procedure we present is semantic tableaux [33]. It is currently receiving a lot of attention as an alternative to resolution based systems. Two implementations of this proof procedure are presented: the first version is a recent version by Beckert and Possega [9] and the second a "compiled" graph version of the basic procedure developed by Possega [86]. The compiled graph version will be specialised in Chapter 6. This indirect route we take in specialising this procedure gives some indication of how other proof procedures not falling into the chain format linear deduction system category or that cannot be written as a declarative meta-program may be specialised.

2.2.1 Model Elimination

In this section we review a version of the model elimination proof procedure [67] described by Poole and Goebel in [85]. To make the discussion that follows more meaningful, we present this procedure in the form given by Poole and Goebel in Program 2.1. It is claimed

$\%$ $prove\ (G, A)$ is true if and only if $Clauses \models A \supset G$
$prove(G, A) \leftarrow member(G, A)$
$prove(G, A) \leftarrow clause(G, B),$
$\qquad\qquad neg(G, GN),$
$\qquad\qquad proveall(B, [GN|A])$

$\%$ $proveall(L, A) \leftarrow$ is true if and only if $Clauses \models A \supset L_i$ for each $L_i \in L$
$proveall([\,], A)$
$proveall([G|R], A) \leftarrow prove(G, A),$
$\qquad\qquad proveall(R, A)$

$\%$ $neg(X, Y)$ is true if X is the negative of Y, both in their simplest form
$neg(not(X), X) \leftarrow ne(X, not(Y))$
$neg(X, not(X)) \leftarrow ne(X, not(Y))$

$\%$ $clause(H, B)$ is true if there is the contrapositive form of an input clause
$\%$ such that H is the head, and B is the body
$\%$ in particular, we know $Clauses \models B \supset H$

Program 2.1: Full Clausal Theorem Prover in Prolog

that this program constitutes a sound and complete clausal theorem prover in Prolog [85]. The following points need mentioning.

- When run on a logic programming system, the above procedure is sound only if the system is sound and includes the occur check [64]. As the occur check relates to the underlying system, our analysis method is independent of whether the occur check is implemented or not.
- The above procedure is complete only when run on a logic programming system if the usual depth-first search used in many logic programming systems is replaced by a depth-first iterative deepening or similar search. However, for analysis purposes the search procedure may be ignored.
- Variable instantiations corresponding to disjunctive or indefinite answers may limit the usefulness of the returned substitutions. This is discussed next.

Consider the following object theory also discussed in [100],

$$p(a) \vee p(b)$$

with theorem to be proved

$$\exists X\, p(X).$$

$\leftarrow prove(p(X), [\,])$ represents the attempt to prove $\exists X\, p(X)$. The only answer that needs to be returned by any sound and complete theorem prover is *YES*. However, with only this one clause in our object theory this goal will fail. To get completeness, we have to add a negated renamed copy of the theorem $\neg p(Y)$ to our object theory. If this is done, the goal $\leftarrow prove(p(X), [\,])$ will return two proofs with substitutions $X = a$ and $X = b$. We now have to be careful how we interpret these results. When we add $\neg p(Y)$ to our theory $p(a) \vee p(b)$, we get an inconsistent theory $\{(p(a) \vee p(b)) \wedge \neg p(Y)\}$. Any clause and its negation occurring in our theory is therefore a logical consequence of this inconsistent theory. In particular, $\exists X\ p(X)$ is a logical consequence of the theory and so are $p(a)$ and $p(b)$. So, the "answer substitutions" mentioned above are sound with respect to the inconsistent theory.

The completeness result for this proof procedure presented by Poole and Goebel [85] states: given some theory T consisting of a consistent set of clauses, if a formula F is logically entailed by T, then there is a refutation proof for F. We therefore have to ignore the substitutions that are returned. We now state the soundness and completeness result for this procedure: $T \models F$ if and only if $M_{T \cup \{\neg F\}} \models prove(G, [\,])$ where G is any clause in $\neg F$. $M_{T \cup \{\neg F\}}$ is the model elimination proof procedure (Program 2.1) with clauses for $\{T \cup (\neg F)\}$ added.

This limitation with indefinite answers is not unique to model elimination: SATCHMO [70] has similar difficulties with uninstantiated variables in disjunctions. The notion of range-restricted clauses is introduced to deal with disjunctions. If this is not done the soundness of the model generation process for unsatisfiability for a set S of clauses cannot be guaranteed (see [70] for a detailed explanation of how this problem is handled).

Alternative ways of handling disjunctive logic programs were developed by Reed, Loveland and Smith [90] and Minker and Rajasekar [76]. Reed, Loveland and Smith developed a fixpoint characterisation of disjunctive logic programs that is based on case-analysis and Minker and Rajasekar developed a fixpoint semantics for disjunctive logic programs based on an extended concept of the Herbrand base of a logic program. Recently, Samaka and Seki [94] also investigated the partial deduction (partial evaluation) of disjunctive logic programs.

As the methods we develop in this thesis are general, we believe that given another theorem prover that does not suffer from the above limitations (even given other fixpoint semantics),

```
% Let T be a theory and F a literal. If T ⊨ ∃F then there exists a proof for F
% from the augmented theory T ∧ ¬F, that is ← solve(F, [ ]) will succeed.
solve(G, A) ←
        depth_bound(D),
        prove(G, A, D)

depth_bound(1) ←
depth_bound(D) ←
        depth_bound(D1),
        D is D1 + 1

prove(G, A, D) ← member(G, A)
prove(G, A, D) ← D > 1, D1 is D − 1,
        clause(G, B),
        neg(G, GN),
        proveall(B, [GN|A], D1)

proveall([ ], A, D) ←
proveall([G|R], A, D) ← prove(G, A, D),
        proveall(R, A, D)
```

Program 2.2: Clausal Theorem Prover Amenable to Partial Evaluation (written in a non-ground representation)

it will be possible to extend or modify our methods and ideas to achieve results similar to that achieved for the given procedure.

In Program 2.2 we present a slightly modified version of the model elimination proof procedure that will be used further on in this thesis for the demonstration of our transformation and analysis methods. A depth-first iterative deepening procedure is introduced to get a runnable procedure. However, the search procedure will be ignored if the aim is to derive only analysis information from this procedure. A top-level predicate $solve(G, A)$ has been added that will designate a unique entry point to the procedure. If the formula F we want to prove is a conjunction of literals, we can call *solve* repeatedly, once for every literal in our formula, or we can introduce a unique literal *goal* and clause $goal \vee \neg F$. Our top-level goal will therefore be $solve(goal, [\,])$ and the clause $goal \vee \neg F$ will be added to our theory. *goal* is a theorem if and only if F is a theorem. $neg(X, Y)$ is implemented as a data base of facts. The way this procedure is now used is as follows. If we are interested in disjunctive answers,

$\%$ $solve$ (G, A, P) is true if and only if $P \models A \supset \exists G$

$solve(Goal, Ancestor\_list, Program) \leftarrow$

$\quad depth\_bound(Depth),$

$\quad prove(Goal, Ancestor\_list, Depth, [\,], Substitution, 1000, N, Program)$

$prove(G, A, D, S, S1, N, N, P) \leftarrow$

$\quad member(G, A, S, S1)$

$prove(G, A, D, S, S2, N, N2, P) \leftarrow D > 1,\ D1\ \ is\ \ D - 1,$

$\quad clause(H, B, N, N1, P),$

$\quad unify(H, G, S, S1),$

$\quad neg(G, GN),$

$\quad proveall(B, [GN|A], D1, S1, S2, N1, N2, P)$

$proveall([\,], A, D, S, S, N, N, P) \leftarrow$

$proveall([G|R], A, D, S, S2, N, N2, P) \leftarrow prove(G, A, D, S, S1, N, N1, P),$

$\quad proveall(R, A, D, S1, S2, N1, N2, P)$

Program 2.3: Clausal Theorem Prover Amenable to Partial Evaluation (written in a ground representation)

we add the negation of the formula to our set of clauses and the semantics explained in this section applies. However, if we are not interested in disjunctive answers, we do not add the negation of the formula to our theory and the correctness results of Poole and Goebel apply.

Next we give a version of the above program using a ground representation similar to the one used by Defoort [30] and de Waal and Gallagher [24]. The program in Program 2.3 is equivalent to the program in Program 2.2 (in the sense that the two programs behave exactly the same on a given object theory taking the differences in representations into account).

$unify(H, G, S, S1)$ unifies a head of a clause H with a selected literal G by first applying the current substitution S with resulting substitution $S1$. $clause(H, B, N, N1, P)$ is true if H is the head of a contrapositive of a clause in the object theory P and B a list of the literals occurring in the body of the contrapositive. N and $N1$ are the indices for creating new variables names in the ground representation. The object theory is represented in the last argument to $solve$. For this program we also have a clear declarative semantics. Because substitutions for object variables are not returned, this meta-program using a

ground representation is slightly more "natural" than the previous program using the non-ground representation. This program can be transformed and analysed with the same tools as the meta-program in Program 2.2 (if the analysis and transformation tools are powerful and general enough, the representation used should not be important).

2.2.2 Naive Near-Horn Prolog

In this section we review the naive near-Horn Prolog proof procedure from [69]. This is a sound but incomplete proof procedure for first-order logic. The intuitive idea of this proof procedure is to try to solve the Horn part of a theorem first and to postpone the non-Horn part (which should be small for near-Horn problems) until the end. Each non-Horn problem that has to be solved is reduced to a series of simpler problems by applications of a splitting rule until only Horn problems remain. The procedure in Program 2.4 is for a positive implication logic. The conversion of any first-order logic formula in conjunctive normal form to this logic is described by Loveland [69]. Similar declarative semantics as for model elimination holds. Variable substitutions have to be ignored.

We assume the usual definition for *append*. $clause(G, D, B)$ is true if G is an atom from a multihead (a multihead is a clause with more than one atom in its head), D is a list of deferred heads and B a list of atoms corresponding to the atoms in the body of the clause (in the correct order). The definition for *depth_bound* is taken from the model elimination procedure in Program 2.2. $goal(G)$ is true if G is a renamed copy of the formula we want to prove. As in model elimination, it may be added to the set of clauses or kept separately as we have done.

The program in Program 2.4 is a complete specification of the naive nH-Prolog proof system, except for the omission of the exit condition for restart blocks (a cancellation operation must have occurred inside this block). The exit condition will prune the above procedure's search space but its omission does not effect soundness (see Loveland [69] for more details). The same remarks about the occur check stated in the previous section also hold for this proof system.

This program is a very intuitive statement of the naive nH-Prolog proof system. The many calls to *append* may look very inefficient, but the whole *append* procedure will disappear because it may be partially evaluated away (since the number of atoms in the body of an object theory clause is known at partial evaluation time).

Note also that indefinite answers are handled somewhat more naturally than in the previous proof procedure. Each solution (substitution) provided by the start and restart blocks must

```
% solve (G, A) is true if and only if Clauses ⊨ A ⊃ ∃G
solve(Goal, Auxiliary_list) ←
        depth_bound(Depth),
        prove([Goal], Auxiliary_list, Depth)

prove([ ], [ ], _) ←                                    % End
prove([ ], [A|As], D) ← goal([G]),                      % Restart
        restart([G], [A|As], D)
prove([G|Gs], A, D) ← D > 1, D1 is D − 1,               % Reduction
        clause(G, G1, B),
        append(B, Gs, Gs1),
        append(G1, A, A1),
        prove(Gs1, A1, D1)

restart([_], [ ], _)                                    % End restart block
restart([ ], [A|As], D) ← goal([G]),                    % Restart
        restart([G], As, D)
restart([G|Gs], [G|As], D) ← D > 1, D1 is D − 1,        % Cancellation
        restart(Gs, [G|As], D1)
restart([G|Gs], [A|As], D) ← D > 1, D1 is D − 1,        % Reduction
        clause(G, G1, B),
        append(B, Gs, Gs1),
        append(A, G1, A1),
        append(A1, As, As1),
        prove(Gs1, As1, D1)
```

Program 2.4: Naive Near-Horn Prolog

be considered in the answer. The disjunction of the query under all required substitutions is therefore the appropriate answer. If we do not "collect" substitutions during the proof, we have to ignore the substitutions returned. The same limitation with indefinite answers discussed in the previous section therefore also holds for this proof procedure.

2.2.3 Semantic Tableaux

As a third proof procedure for first-order logic, we look at semantic tableaux [98, 33]. Tableaux are, just like resolution, refutation systems. To prove a formula F, we begin

with $\neg F$ and produce a contradiction. The procedure for doing this involves expanding $\neg F$ with tableau expansion rules. In tableau proofs, such an expansion takes the form of a tree, where nodes are labelled by formulas. Each branch of the tree could be thought of as representing the conjunction of the formulas appearing on it and the tree itself as representing the disjunction of its branches. A branch in a tree is closed if it contains Z and $\neg Z$. Tableau proofs are closed trees, that is every branch in the tree must be closed. A proof for F amounts to constructing a closed tableau for $\neg F$.

Logic programs implementing tableau theorem provers have been written by Fitting [33]. A lean version of a tableau theorem prover was written by Beckert and Possega [9]. The idea of this theorem prover is to achieve maximal efficiency from minimal means. It is a sound and complete theorem prover for first-order logic based on free-variable tableaux [33]. The program is given in Program 2.5.

For the sake of simplicity, this theorem prover is restricted to formulas in Skolemised negation normal form. It can be extended by adding the standard tableau rules [8]. The standard Prolog syntax is used for first-order formulas (as presented by Beckert and Possega in their paper): atoms are Prolog terms, "-" is negation, ";" disjunction and "," conjunction. Universal quantification is expressed as $all(X, F)$, where X is a Prolog variable and F is the scope. For example, the first-order formula $(p(0) \wedge (\forall n(\neg p(n) \vee p(s(n)))))$ is represented by $(p(0), all(N, (-p(N); p(s(N)))))$.

This tableau implementation is not suitable for specialisation with the methods developed in this thesis, because declarative semantics cannot be given to *copy_term*. We therefore have to look at less "lean" versions of the semantic tableau proof procedure as for instance presented by Fitting [33]. However, there is another approach presented by Possega [86] that seems promising.

This alternative approach works by compiling a graphical representation of a fully expanded tableau into a program that performs the search for a proof at runtime. This results in more efficient proof search, since the tableau does not need to be expanded any more at runtime. The proof consists only of determining if the tableau can be closed or not. He shows how the method is applied to the target language Prolog. This step can be regarded as a partial evaluation step in the same spirit as the partial evaluation steps in our specialisation method. We investigate this approach further in Chapter 6.

This concludes the discussion of three first-order proof procedures. Although some of the presented proof procedures are very naively implemented, the aim of this thesis is not to reinvent well established optimisation techniques in theorem proving (such as regularity [6]), but to show how new information can be inferred that cannot be done by other techniques or that the inferred information is more precise than achievable with other techniques.

```
% prove (+Fml,+UnExp,+Lits,+FreeV,+VarLim)

% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% Fml: inconsistent formula in skolemized negation normal form.
% syntax: negation: '-', disj: ';', conj: ','
% quantifiers: 'all(X,<Formula> )', where 'X' is a Prolog variable.

% UnExp: list of formula not yet expanded
% Lits: list of literals on the current branch
% FreeV: list of free variables on the current branch
% VarLim: max. number of free variables on each branch
% (controls when backtracking starts and alternate
% substitutions for closing branches are considered)

prove((A,B),UnExp,Lits,FreeV,VarLim) <-!,
        prove(A,[B|UnExp],Lits,FreeV,VarLim)
prove((A;B),UnExp,Lits,FreeV,VarLim) <-!,
        prove(A,UnExp,Lits,FreeV,VarLim),
        prove(B,UnExp,Lits,FreeV,VarLim)
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) <-!,
        \+ length(FreeV,VarLim),
        copy_term((X,Fml),(X1,Fml1)),
        append(UnExp,[all(X,Fml)],UnExp1),
        prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim)
prove(Lit,_,[L|Lits],_,_) <-
        (Lit = -Neg;-Lit = Neg)->
        (unify(Neg,L);prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) <-
        prove(Next,UnExp,[Lit|Lits],FreeV,VarLim)
```

Program 2.5: Lean Tableau-Based Theorem Proving

In the next chapter, we refine partial evaluation for the specialisation of the proof procedures presented in this chapter.

Chapter 3

Partial Evaluation

Partial evaluation is an optimisation technique whose use may result in gains in efficiency, especially when used in meta-programming to "compile away" layers of interpretation. However, partial evaluation also has its limitations as a specialisation technique as it cannot take dynamic information that is dependent on the run time environment of the program it is specialising into account. After an introduction to partial evaluation and a review of its limitations, we present a new abstraction operation based on an extended notion of a characteristic path. This abstraction operation may allow the partial evaluator to make better use of the static information available from the first-order theories. A "minimal" partial evaluator is then defined, suitable for the specialisation of proof procedures, that preserves the structure of the meta-program. The terminology used is consistent with that in [65, 10, 36].

3.1 Introduction

Partial evaluation is ideally suited for removing most of the overheads incurred by the writing of meta-interpreters, especially when using the ground representation. It is debatable if many useful meta-interpreters can be written using this representation without the aid of a partial evaluator to remove the significant overheads of the ground representation. The aim of the developers of the logic programming language Gödel [51] is therefore to provide a partial evaluator with the Gödel system to facilitate efficient meta-programming in the Gödel ground representation.

The first Futamura projection, defining compilation as specialisation has been studied in great detail as it provides an alternative (although not yet practical) technique to conven-

tional compilation techniques [35, 59, 46]. Much effort has also been spent on developing a self-applicable partial evaluator [78, 46] in order to achieve the second and third Futamura projections. This is rightly so, as a practical self-applicable partial evaluator will open the doors to many new and exciting applications.

A criticism against the work done on producing partial evaluators is that the basic partial evaluation algorithms used so far are not powerful enough: the result of partial evaluation may contain many redundant computations that do not contribute to solving the given goals. As indicated by Gurr [46], the crucial step in obtaining a compiler-generator is a partial evaluator capable of producing efficient code upon the specialisation of arbitrary meta-programs. In this thesis we wish to show how significant improvements can be made on state of the art partial evaluations for the specialisation of meta-programs.

During the last few years many partial evaluators for various subsets of Prolog have been written [93, 36, 78, 62, 88]. Gurr presented a self-applicable partial evaluator for Gödel in [46]. Most of these partial evaluators are based on the theory developed by Lloyd and Shepherdson [65] and the algorithm by Benkerimi and Lloyd [10]. We repeat from [65] the basic definitions and main theorems that are used.

DEFINITION 3.1.1 **resultant**
A **resultant** *is a first-order formula of the form $Q_1 \leftarrow Q_2$, where Q_i is either absent or a conjunction of literals ($i = 1, 2$). Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.*

DEFINITION 3.1.2 **resultant of a derivation**
Let P be a normal program, G a normal goal $\leftarrow Q$ and $G_0 = G, G_1, \ldots, G_n$ an SLDNF-derivation of $P \cup \{G\}$, where the sequence of substitutions is $\theta_1, \ldots, \theta_n$ and G_n is $\leftarrow Q_n$. Let θ be the restriction of $\theta_1, \ldots, \theta_n$ to the variables in G. Then we say that the derivation has **length** *n with* **computed answer** *θ and* **resultant** *$Q\theta \leftarrow Q_n$. (If $n = 0$, then the resultant is $Q \leftarrow Q$).*

DEFINITION 3.1.3 **partial evaluation**
Let P be a normal program, A an atom, and T be an SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_r$ be (nonroot) goals in T chosen so that each nonfailing branch of T contains exactly one of them. Let R_i ($i = 1, \ldots, r$) be the resultant of the derivation from $\leftarrow A$ down to G_i given by the branch leading to G_i. Then the set of clauses $R_1, \ldots, R_r$ is called a **partial evaluation** *of A in P.*

DEFINITION 3.1.4 **independent**
Let A be a finite set of atoms. A is **independent** *if no pair of atoms in A has a common instance.*

DEFINITION 3.1.5 **closed**
*Let S be a set of first-order formulas and A a finite set of atoms. S is A-***closed** *if each atom in S containing a predicate symbol occurring in an atom A is an instance of an atom in A.*

We now present the correctness requirement for partial evaluation from Lloyd and Shepherdson [65].

THEOREM 3.1.6 **correctness requirement**
Let P be a normal program, G a normal goal, A a finite, independent set of atoms, and P' a partial evaluation of P with respect to A such that $P' \cup \{G\}$ is A-closed. Then the following holds:

(1) $P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ iff *$P \cup \{G\}$ does.*

(2) $P' \cup \{G\}$ has a finitely failed SLDNF-tree iff *$P \cup \{G\}$ does.*

The above correctness requirement inhibits specialisation: infinite failures may not be removed. We intend to relax the above requirement by replacing the iff's by if's. This gives us scope for removing infinite failures and adding solutions that we could not do if we adhered to the correctness requirements of [65]. Assuming a depth-first search, the specialised program P' might now behave differently from the original program P in the following ways:

1. P' might fail finitely where P failed infinitely,
2. P' might succeed where P failed infinitely,
3. P' might return more solutions than P.

These "improvements" are important in the theorem proving domain for the following reasons.

(1) Proving that a formula F is a theorem or is not a theorem is much more informative than not being able to prove anything about the status of F.

(2) Getting another (possibly shorter) proof may also be important.

Examples of these improvements are given in Chapter 6.

In Section 3.3.1 we refine partial evaluation for the specialisation of proof procedures. Our refinement is based on the notion of a *characteristic path* introduced by Gallagher and Bruynooghe [37]. This notion was introduced to improve the quality of the specialised code generated by program specialisation.

Informally, a characteristic path gives the structure of an SLDNF derivation. It records the clauses selected in a derivation and the position of the literals resolved upon. Information about the resolvents are abstracted away. The intuitive idea behind this abstraction (as used in the logic program specialisation system SP [36]) is that if two atoms B and C have the same characteristic path, they are regarded as equivalent from the point of view of partial evaluation. B and C are therefore represented by a single atom (which will lead to shared specialised code for B and C as they are indistinguishable to the partial evaluator).

It is important to realise the subtlety of this notion as used in the SP-system. We therefore introduce briefly the main aspects of the SP-system and indicate how the characteristic path is used in the generation of specialised code.

The SP-system takes as input a normal program P and goal G and generates a specialised more efficient program P'. The algorithm used in this system consists of three steps:

(1) Flow analysis.

(2) Construction of the specialised program.

(3) Cleaning up of the program constructed in step 2.

We concentrate on the first part of the algorithm as this is where we will introduce our refinements. The flow analysis algorithm from Gallagher [36] is reproduced in Figure 3.1. It bears a superficial resemblance to the partial evaluation algorithm by Benkerimi and Lloyd [10], but produces a set of atoms and not a partially evaluated program.

In the second step of the algorithm, each atom generated by this algorithm is renamed while constructing (by a process of folding and unfolding) the specialised program. Different renamed predicates will give rise to different pieces of specialised code.

$$
\begin{array}{l}
\textit{begin} \\
\quad A_0 \ := \ \textit{the set of atoms in } G \\
\quad i \ := \ 0 \\
\quad \textit{repeat} \\
\qquad P' \ := \ \textit{a partial evaluation of } A_i \textit{ in } P \textit{ using } U \\
\qquad S \ := \ A_i \ \cup \ \{p(t) \mid B \ \leftarrow \ Q, p(t), Q' \ \textit{ in } P' \\
\qquad\qquad\qquad\qquad OR \\
\qquad\qquad\qquad\qquad B \ \leftarrow \ Q, not(p(t)), Q' \textit{ in } P'\} \\
\qquad A_{i+1} \ := \ abstract(S) \\
\qquad i \ := \ i+1 \\
\quad \textit{until } A_i \ = \ A_{i-1} \ (\textit{modulo variable renaming}) \\
\textit{end}
\end{array}
$$

Figure 3.1: Basic Partial Evaluation Algorithm

The third step of the algorithm include optimisations such as:

- calls to procedures defined by a single clause are unfolded,
- common structure at argument level are factorised out and
- duplicate goals in the body of a clause are removed.

We also include the following definitions taken from the description of the SP-system.

DEFINITION 3.1.7 **unfolding rule**
An **unfolding rule** *U is a function which, given a program P and atom A, returns exactly one finite set of resultants, that is a partial evaluation of A in P. If A is a finite set of atoms and P a program, then the set of resultants obtained by applying U to each atom in A is called a partial evaluation of A in P using U.*

DEFINITION 3.1.8 **characteristic path**
Let P be a normal program and G be a goal. Assume that the clauses in P are numbered. Let $G, G_1, \ldots, G_n$ be an SLDNF derivation of $P \cup \{G\}$. The characteristic path of the derivation is defined as the sequence $(l_1, c_1), \ldots, (l_n, c_n)$, where l_i is the position of the selected literal in G_i and c_i is defined as:

- *if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i;*
- *if the selected literal is $\neg p(t)$, c_i is the literal $p(t)$.*

DEFINITION 3.1.9 **abstract(S)**
Let S be a set of atoms. An operation **abstract(S)** *is any operation satisfying the following condition. abstract(S) is a set of atoms A with the same predicates as those in S, such that every atom in S is an instance of an atom in A.*

It can be seen that the abstraction operation in the given algorithm ultimately determines the granularity of the specialised code: no abstraction will possibly give rise to the generation of infinitely many different specialised versions of the same predicate, while over-abstraction may give rise to few specialised versions of each predicate. If the abstraction operation is based on the notion of a characteristic path (as is done in the SP-system), only one different specialised definition is generated for each unique characteristic path.

One of the aims of the specialisation of proof procedures stated in the previous section was to distinguish the different applications of the same inference rule (which are expressed as one or more clauses in our meta-program). We therefore need a finer grained level of abstraction than that given by the notion of a characteristic path. This is developed in Section 3.3.1. In the next section we review the limitations of partial evaluation.

3.2 Limitations

In Chapter 1 we indicated that we are mainly interested in the specialisation of chain format linear deduction systems. This "restriction" to a particular type of deduction system we placed on ourselves is not a "hard" restriction. We interpret this class of systems loosely and proof procedures that include aspects of these systems may also be candidates for specialisation with the techniques we develop in this thesis. We therefore introduce chain format linear deduction system informally and indicate why partial evaluation may not be able to specialise such systems effectively with respect to first-order theories.

Given a set S of clauses and a clause C_0 in S, a linear deduction of C_n from S with top clause C_0 is a deduction of the form shown in Figure 3.2. For $i = 0, 1, \ldots, n-1$, C_{i+1} is a resolvent of C_i and B_i, and B_i is either in S or in C_j for some j, $j < i$. Each arc is labelled by an inference rule R_j, $1 \leq j \leq m$, where m is the number of inference rules in our proof

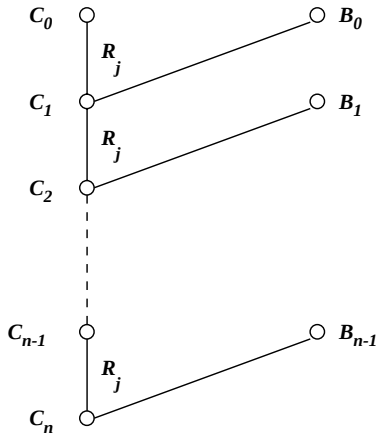


Figure 3.2: A Linear Deduction

procedure. n applications of one or more of the inference rules R_j give the resolvent C_n. Although an inference rule is applied to the whole resolvent C_i, usually only part of the resolvent is actively involved in the transformation to C_{i+1} (e.g. a literal in the resolvent may be deleted, replaced).

An inference rule in our proof procedure is implemented as one or more clauses in the corresponding meta-program. These clauses will now be applied to the predicates from our object theory in the search for a proof of a given formula. We call such an instance of a clause in the definition of an inference rule, *an instance of an inference rule*. In reality, one or more applications of perhaps several clauses may be needed to give a corresponding transformation specified by an inference rule and more than one predicate from the object theory may be involved. We therefore use this terminology loosely to indicate instances of clauses corresponding to inference rules in our proof procedure.

In the chain format linear deduction systems, the resolvents C_i are augmented with information from the deduction so far that may make further deductions possible. Such an augmented resolvent is called a chain. In Loveland [67] this extra information is identified by putting square brackets [...] around it. In the model elimination procedure presented in Program 2.2 the extra information is contained in the ancestor list. Chain format linear deduction systems can therefore be thought of as linear deduction systems with a method of recording information about deductions. This information may then be used to make further deduction possible.

There are three major reasons why partial evaluation may not be able to specialise such deduction systems effectively.

- Infinite linear deductions occur frequently.
- In the problem reduction format where a complex problem is divided into several subproblems (which may be less complex to solve), an infinite number of linear deductions may occur.
- The chain format introduces more complexity into the system.

Partial evaluation has to terminate to be of any use. It therefore cannot evaluate an infinite linear deduction or an infinite number of linear deductions (of which some may be infinite deductions as well). Partial evaluation therefore has to generalise at some stage during the specialisation to terminate. This generalisation may lose the information needed for effective specialisation.

Van Harmelen [106] also quoted the lack of static information as a possible reason for the generation of suboptimal partially evaluated code. In our application, the first-order theory that we are going to specialise the partial evaluator with respect to is known. This should therefore not be a problem. However, many partial evaluators may not be able to exploit the available static information fully. In a later section we show how to add predicates containing static information derived from the object theory to our meta-program to assist partial evaluation in generating better specialised code.

To complicate matters further, instances of inference rules in our proof procedure may succeed for some goals and fail for others. We would like partial evaluation to return specialised code for the successful goals and fail for the unsuccessful goals. This should be done in the presence of the complications introduced by chain format linear deduction systems described in the points above.

Although these problems cannot be solved completely (because of the undecidability of first-order logic), we might be able to alleviate some of the problems by the use of a combination of transformation and analysis tools refined for the specialisation of proof procedures. This may lead to useful specialisations not possible with individual transformation or analysis tools.

We propose the following plan of action to improve the specialised code generated by partial evaluation.

1. Extend or refine partial evaluation to make better use of the static information available.
2. Introduce a subsequent abstract interpretation phase to analyse the program generated by partial evaluation.

3. Optimise the code generated by partial evaluation. These optimisations will be based on a simple decision procedure that is able to exploit the information generated by abstract interpretation.

Although the proposed specialisation method consists of several phases, it is still an instance of the first Futamura projection: to specialise an interpreter (the meta-program) with respect to partially known input (the object theory and formula to be proved).

In the next sections we refine partial evaluation to generate specialised versions of meta-predicates. In the next chapter we develop the regular approximator. A simple decision procedure for deciding how to use this analysis information is also given. The extended partial evaluation, abstract interpretation and decision procedure are then combined into three specialisation methods in Chapter 5.

3.3 Partial Evaluation for Specialising Proof Procedures

3.3.1 Extending the Notion of a Characteristic Path

An extension of the notion of a characteristic path gives us a means to control the granularity of the specialised code generated by the given algorithm. As we want a finer grain, we must make the abstraction operation more refined. Also, we want different specialised versions of the same inference rule to be generated for different instances of the inference rule, so we have to extend the notion of a characteristic path with argument information. However, we have to be careful not to generate too many specialised definitions, as this may eventually annul the benefits derived from this refinement. A heuristic that has been found very useful to base our refined characteristic path notion on is to look at the different functors occurring inside atoms. We call this notion a *characteristic path with argument differentiation.*

DEFINITION 3.3.1 **characteristic path with argument differentiation**
Let P be a normal program and let G be a definite goal. Assume that the clauses in P are numbered. Let $G, G_1, \ldots, G_n$ be an SLDNF derivation of $P \cup \{G\}$. The **characteristic path with argument differentiation** *of the derivation is defined as:*

- *if the selected literal is an atom, the sequence $(l_1, c_1, A_{11}, \ldots, A_{1m}), \ldots, (l_n, c_n, A_{n1}, \ldots, A_{nm})$, where l_i is the position of the selected literal in G_i, c_i is defined as the number of the clause chosen to resolve with G_i and $A_{i1}, \ldots, A_{im}$ are defined as follows:*

$$
\begin{aligned}
&solve(A, B, C, D) \leftarrow \\
&\qquad member(A, B), \\
&\qquad member(C, D)
\end{aligned}
$$

$$
\begin{aligned}
&member(X, [X|Xs]) \leftarrow \\
&member(X, [Y|Xs]) \leftarrow \\
&\qquad member(X, Xs)
\end{aligned}
$$

Program 3.1: Example Illustrating the Need for a More Refined Abstraction Operation

$$
\begin{aligned}
&solve(1, X1, 2, X2) \leftarrow \\
&\qquad member\_1(1, X1), \\
&\qquad member\_1(2, X2)
\end{aligned}
$$

$$
\begin{aligned}
&member\_1(X1, [X1|X2]) \leftarrow \\
&member\_1(X1, [X2|X3]) \leftarrow \\
&\qquad member\_1(X1, X3)
\end{aligned}
$$

Program 3.2: A Partial Evaluation using Path Abstraction

  - *if the m-th argument of the selected atom is a constant, A_{im} is that constant;*
  - *if the m-th argument of the selected atom is a functor with arguments, A_{im} is that functor;*
  - *if the m-th arguments of the selected atom is a variable, A_{im} is the anonymous variable (underscore in Prolog notation).*

- *if the selected literal is not an atom, the sequence $(l_1, c_1), \ldots, (l_n, c_n)$, where l_i is the position of the selected literal in G_i and c_i is the literal $p(t)$.*

This definition can be generalised in an obvious way so that the argument abstraction corresponds to the well-known notion of a depth-k abstraction.

Consider the example in Program 3.1 and query $\leftarrow solve(1, A, 2, B)$ to this program. A partial evaluation using the characteristic path abstraction is given in Program 3.2. The partial evaluator is unable to distinguish between the two different calls to $member(X, Y)$ and one specialised definition is generated. This distinction might be important during the specialisation of proof procedures as important information may be lost if such a generalisation is made.

$solve(1, X1, 2, X2) \leftarrow$
$\quad member\_1(X1),$
$\quad member\_2(X2)$

$member\_1([1|X1]) \leftarrow$
$member\_1([X1|X2]) \leftarrow$
$\quad member\_1(X2)$

$member\_2([2|X1]) \leftarrow$
$member\_2([X1|X2]) \leftarrow$
$\quad member\_2(X2)$

Program 3.3: A Partial Evaluation using Path Abstraction Augmented with Argument Information

In a complex meta-program these two calls to $member\_1(X, Y)$ might be generated in different parts of the meta-program. If we are able to detect (by abstract interpretation or by partial evaluation itself) that one of the calls to $member\_1(X, Y)$ will always fail, we may not delete the $member\_1(X, Y)$ procedure from our program as we would then fail the other, possibly successful, call to $member\_1(X, Y)$ as well, which might be incorrect. A possibly infinite deduction might now be preserved which would lead to less optimal specialised code being generated. Incorporating the improved characteristic path operation into our partial evaluator rectifies this problem and gives the improved code in Program 3.3.

The differentiation up to first functor in each argument (depth-1 abstraction) is usually enough to force the required renamings in the partial evaluation algorithm. However, sometimes it is necessary to further distinguish atoms having the same top-level functors. For instance, many meta-programs have lists as arguments. If all arguments are lists, we are back to the original definition. If this improvement is however combined with other features of the SP-system, such as *most specific generalisation* and *factoring out common structure* [36], this is usually not a problem.

The termination conditions of the SP-system (see [36]) are not affected by replacing the characteristic paths in the abstraction operation by characteristic paths with argument differentiation. The relevant condition is that the abstraction operation, $abstract(S)$, should return a finite set of atoms. This is the case in the original algorithm as the number of generalisations of an atom (module variable renaming) is finite. This is also the case for the revised abstraction operation: we generalise less frequently.

3.3.2 Adding Static Information

As pointed out in Section 3.2, there may be static information available from our theory that a partial evaluator is unable to exploit. For instance, in the first-order theories considered in this thesis, we know that we have a finite number of predicate symbols, function symbols and constants (we restricted our theories to finite axiomatisations). It might be possible to infer automatically this information doing an analysis of the theory. However, we choose to add "type" atoms to our meta-program to convey this information to the partial evaluator. Kursawe [61] uses similar ideas to improve the quality of the code generated by "pure partial evaluation". He introduces the notion of a complete pattern set to assist partial evaluation in generating different specialised code for each pattern occurring in the pattern set.

For instance in the meta-program given in Program 2.2, we know that the first argument to solve will be a literal. If our theory contains two predicates p and q, we may add an atom

$$literal(Goal)$$

to the definition of solve. The definition for $literal(Goal)$ will be

$$literal(p(\ldots)) \quad literal(\neg p(\ldots)) \quad literal(q(\ldots)) \quad literal(\neg q(\ldots)).$$

The aim now is to "force" the partial evaluator to incorporate this information into the partially evaluated code. This can be done by unfolding. However, if the partial evaluator is restricted to determinate unfolding (to avoid adding choice points to the partially evaluated program), we may need to build a mechanism into the partial evaluator to force the unfolding of the added atoms. We therefore reconsider unfolding in the next section.

3.3.3 Reconsidering Unfolding

In Chapter 2 we assumed a unique predicate for representing the object program (theory) in the meta-program (e.g. $clause(X, Y)$). All calls to the object theory given by this literal should be unfolded. As we have only a finite number of clauses in our object theory, this will always terminate. Furthermore, all other calls to predicates that consist solely of a finite number of facts should also be unfolded. This corresponds to the unit-reduction operation by Bibel [11] and the motivation for this is similar to that given there. We call this *unit-unfolding*.

The addition of atoms conveying static information to the meta-program may also make unit-unfolding applicable. However, the partial evaluator may not be able to automatically determine which atoms are suitable for unit-unfolding. A directive to the partial evaluator

$$unfold(literal(\ldots))$$

$match(Pattern, Text) \leftarrow match(Pattern, Text, Pattern, Text)$

$match([\,], \_, \_, \_) \leftarrow$
$match([A|B], [C|D], P, T) \leftarrow A = C,$
$\quad match(B, D, P, T)$
$match([A|\_], [C|\_], P, [\_|T]) \leftarrow A \backslash == C,$
$\quad match(P, T, P, T)$

Program 3.4: Naive String Matching Algorithm

is therefore provided to force the unfolding of certain atoms. Because these atoms are unfolded away by unit-unfolding, they will not occur in the partially evaluated code, but the information conveyed by them will be incorporated into the partially evaluated program. This may allow the revised abstraction operation presented in Section 3.3.1 to improve further the specialised code by differentiating between different instances of inference rules. This in turn may allow a finer grained analysis with more precise information being inferred during the analysis phase.

A generalisation of unit-unfolding may be to unfold all non-recursive predicates in our program, not only those that have a corresponding unfold declaration. We chose to leave this option open to the user. Unfold declarations for all the non-recursive predicates will accomplish this.

3.3.4 A Minimal Partial Evaluator

With the above refinements incorporated into the SP-system, we still have a very general partial evaluator that can be applied to many logic programs. Consider the naive string matching algorithm in Program 3.4 used by Gallagher [36] and Martens [73] as an example.

The SP-system derives an optimised string matching program similar to the Knuth-Morris-Pratt algorithm from a naive algorithm given some ground pattern, for example $match([a, a, b], T)$. A partial evaluation of the program given in Program 3.4 using determinate unfolding with path abstraction is given in Program 3.5.

If we know that our text will be restricted to an alphabet with a finite number of letters, we might want to make this information available to the partial evaluator with the aim of getting a more restricted partial evaluation. This can be done by adding "type" atoms as

$match([a,a,b],X1) \leftarrow$
$\quad match1\_1(X1)$

$match1\_1([a|X1]) \leftarrow match1\_2(X1)$
$match1\_1([X1|X2]) \leftarrow a \backslash == X1,$
$\quad match1\_1(X2)$

$match1\_2([a|X1]) \leftarrow match1\_3(X1)$
$match1\_2([X1|X2]) \leftarrow a \backslash == X1, a \backslash == X1,$
$\quad match1\_1(X2)$

$match1\_3([b|X1]) \leftarrow$
$match1\_3([X1|X2]) \leftarrow b \backslash == X1,$
$\quad match1\_2([X1|X2])$

Program 3.5: Specialised Naive String Matching Algorithm using Path Abstraction

$match(Pattern,Text) \leftarrow match(Pattern,Text,Pattern,Text)$

$match([\,],\_,\_,\_) \leftarrow$
$match([A|B],[C|D],P,T) \leftarrow literal(A), literal(C), A = C,$
$\quad match(B,D,P,T)$
$match([A|\_],[C|\_],P,[\_|T]) \leftarrow literal(A), literal(C), A \backslash == C,$
$\quad match(P,T,P,T)$

$literal(a) \leftarrow$
$literal(b) \leftarrow$
$literal(c) \leftarrow$

Program 3.6: Naive String Matching Algorithm with Static Information

described in Section 3.3.2 to our program and unfolding away these atoms with the unfold directive or unit-unfolding of Section 3.3.3. The string matching program with added "type" atoms is given in Program 3.6. Partial evaluation of this program with all the improvements discussed so far, gives the program in Program 3.7.

In this case we restricted the whole alphabet to just a, b and c. In the previous program, the string that we are matching on may also have contained characters other than a, b and

$$match([a,a,b],X1) \leftarrow$$
$$\quad match1\_1(X1)$$

$$match1\_1([a|X1]) \leftarrow match1\_2(X1)$$
$$match1\_1([b|X1]) \leftarrow match1\_1(X1)$$
$$match1\_1([c|X1]) \leftarrow match1\_1(X1)$$

$$match1\_2([a|X1]) \leftarrow match1\_3(X1)$$
$$match1\_2([b|X1]) \leftarrow match1\_1(X1)$$
$$match1\_2([c|X1]) \leftarrow match1\_1(X1)$$

$$match1\_3([b|X1]) \leftarrow$$
$$match1\_3([a|X1]) \leftarrow match1\_3(X1)$$
$$match1\_3([c|X1]) \leftarrow match1\_1(X1)$$

Program 3.7: Specialised Naive String Matching Algorithm with Finite Alphabet

c. In many applications the alphabet is known. As is the case for our first-order theories, we should use this information to our advantage.

The difference between the partially evaluated code generated for the string matching programs without and with static information added (Program 3.7 and Program 3.5 respectively) is the grain of the partial evaluation code: $match1\_1$, $match1\_2$ and $match1\_3$ have three cases each instead of two. This may allow a more precise analysis and therefore better specialised code to be generated. The program in Program 3.7 is also more restricted than the program in Program 3.5.

This example also illustrates that if a sophisticated partial evaluator is used, it is difficult to control precisely what the partially evaluated program may look like. If our aim is to infer information about inference rules in the proof procedure, powerful transformations may generate a partially evaluated program with a structure completely different from that of the original program. This may make it difficult to relate information inferred for the specialised program back to the program we started with. It is therefore useful to have a "minimal" partial evaluator available that only performs simple transformations such as unfolding and renaming. First, we define what we understand by a renaming transformation and a determinate unfolding step.

DEFINITION 3.3.2 **renaming definition**
Let $a(c_1, \ldots, c_n)$ be an atom and c_i, $1 \leq i \leq n$ a term. Let p be a predicate symbol different from a. Then the clause $p(c_1, \ldots, c_n) \leftarrow a(c_1, \ldots, c_n)$ is called a renaming definition for $a(c_1, \ldots, c_n)$.
Let A be a set of atoms. Then the set of renaming definitions for A is the set of renaming definitions for the atoms in A.

DEFINITION 3.3.3 **complete renaming transformation**
Let P be a normal program, and $C = A \leftarrow Q_1, B\theta, Q_2$ be a clause in P, where $B\theta$ is a literal and Q_i are conjunctions of literals. Let R be a set of renaming transformations whose head predicates do not occur in P.

- *if B is an atom, let $D = B' \leftarrow B$ be in R. Then let $C' = A \leftarrow Q_1, B'\theta, Q_2$.*
- *if B is a literal $\neg N$, let $D = N' \leftarrow N$ be in R. Then let $C' = A \leftarrow Q_1, \neg N'\theta, Q_2$.*

A complete renaming of a clause (with respect to a set of renaming definitions) is obtained by renaming every atom in the clause body for which a renaming definition exists.

DEFINITION 3.3.4 **determinate unfolding step**
Let P be a normal program and let $C = A \leftarrow Q$ be a clause in P. Then a clause $A\theta \leftarrow R$ is obtained by unfolding C in P if there is exactly one SLDNF derivation of $P \cup \{\leftarrow Q\}$ of the form $\leftarrow Q, \leftarrow R$, with computed answer θ. The program $P' = P - \{C\} \cup \{C'\}$ is obtained from P by unfolding C in P. The clause $A\theta \leftarrow R$ is obtained by a **determinate unfolding** *step.*

We define our "minimal" partial evaluator as follows.

DEFINITION 3.3.5 **minimal partial evaluator**
A **minimal partial evaluator** *is a program transformation system restricted to the following transformations.*

(1) Unit-Unfolding.

(2) A finite number of deterministic unfolding steps.

(3) A complete renaming transformation.

The set of renaming definitions R is obtained by considering the atoms occurring in the bodies of clauses in P. The correctness of this transformation system follows from the correctness of the individual transformations. Gallagher and Bruynooghe [37] give an independent argument for the splitting of procedures into two or more versions handling different cases. Their correctness results can be independently applied if necessary. A general top-level goal is implicit in the transformation system.

Application of the minimal partial evaluator preserves the structure of the meta-program which makes it easier to relate inferred information back to the original program. In Chapter 6 we use this minimal partial evaluator for the specialisation of the model elimination proof procedure given in Program 2.2. However, the SP-system was not used in this specialisation. A transformation program giving equivalent specialisations to that described by the minimal partial evaluator for the model elimination proof was written.

This program is much simpler and faster than a general partial evaluator as several problem specific (proof procedure specific) optimisations can be made. It is only a few lines of code and will be used for all the model elimination results given further in this thesis (except where explicitly otherwise indicated).

Chapter 4

Abstract Interpretation for the Deletion of Useless Clauses

Abstract interpretation provides a formal framework for developing program analysis tools. The basic idea is to use approximate representations of computational objects to make program dataflow analysis tractable. In the first part of this chapter we introduce abstract interpretation. Useless clauses, that is clauses that yield no solutions are then defined. Next, we introduce regular approximations of logic programs and show how such an approximation can be used to detect useless clauses. In the final part of this chapter we discuss various versions of a query-answer transformation. This transformation can increase the precision of the regular approximation allowing a greater number of useless clauses to be detected.

4.1 Introduction

The objective of abstract interpretation is to infer run-time properties of programs by analysing the program at compile time. This information may be used for compiler optimisation or program transformation (as used in this thesis). But, for abstract interpretation to be of any use, it must terminate (this is also required of partial evaluation). An exact analysis is therefore usually impossible as most non-trivial program properties are either undecidable or very expensive to compute. A solution is to compute decidable approximations that are sound (anything that can happen at runtime is predicted by the analysis) and computationally less expensive. Furthermore, the approximation should capture "important" information.

The seminal paper on abstract interpretation is by Cousot and Cousot [20], with its applica-

tion to logic programming investigated by Mellish [75] and Bruynooghe [16], amongst others. A collection of papers on the abstract interpretation of declarative languages can be found in [1]. Informative tutorials on abstract interpretation were also given by Hermenegildo [50] and Debray [29]. We closely follow Marriott and Søndergaard [72] in our presentation.

It is necessary to establish relationships with formal semantics of the language to reason about the soundness of an analysis. There are at least three different approaches within the framework described by Cousot and Cousot [20] that can be used for the analysis of logic programs.

- Galois Connection Approach: Define a domain of "descriptions" and set up a Galois connection between the actual domain and description domain.
- Widening/Narrowing Approach: The analysis is carried out over the actual computational domain using "widening" and "narrowing" operators. There is no separate domain of descriptions.
- Hybrid Approach: Augment the Galois Connection approach with widening and narrowing operators to accelerate convergence.

The approach used in this thesis is based on an abstraction of the T_p operator described by Marriott and Søndergaard [72] (which is based on the Galois Connection Approach). First, we introduce some basic definitions taken from [72].

DEFINITION 4.1.1 **partial ordering, poset**
A **partial ordering** *is a binary relation that is reflexive, transitive and antisymmetric. A set equipped with a partial ordering is a* **poset**.

DEFINITION 4.1.2 **upper bound**
Let $(X, \leq)$ *be a poset. An element* $x \in X$ *is an* **upper bound** *for* $Y \subseteq X$ *iff* $y \leq x$ *for all* $y \in Y$.

DEFINITION 4.1.3 **least upper bound**
An upper bound x *for* Y *is the* **least upper bound** *for* Y *iff, for every upper bound* x' *for* Y, $x \leq x'$. *When it exists, we denote it by* $\sqcup Y$.

A **lower bound** and **greatest lower bound** can be dually defined. The greatest lower bound of Y is denoted by $\sqcap Y$.

DEFINITION 4.1.4 **complete lattice**
A poset for which every subset possesses a least upper bound and a greatest lower bound is a **complete lattice**.

Cousot and Cousot used a function γ to give the semantics of descriptions. γ is called the concretization function.

DEFINITION 4.1.5 **insertion**
An **insertion** *is a triple (γ, D, E) where D and E are complete lattices and the monotonic function $\gamma : D \to E$ is injective (one-to-one) and costrict.*

In Cousot and Cousot's theory of abstract interpretation, the existence of an adjoined, so called abstraction function $\alpha : E \to D$ is required, which we now define.

DEFINITION 4.1.6 **adjoined**
Let D and E be complete lattices. The (monotonic) functions $\gamma : D \to E$ and $\alpha : E \to D$ are **adjoined** *iff*

$$\forall d \in D,\ d = \alpha(\gamma d),\ \textit{and}$$

$$\forall e \in E,\ e \leq \gamma(\alpha e),$$

where $\leq$ is the ordering on E.

The abstraction function can be thought of as giving the best overall approximation to a set of objects. We now define a safe approximation.

DEFINITION 4.1.7 **safe approximation**
Let (γ, D, E) be an insertion. We define $appr_\gamma : D \times E \to Bool$ by $appr_\gamma(d, e)$ iff $e \leq \gamma d$, where $\leq$ is the ordering on E.

$appr_\gamma(d, e)$ denotes "d safely approximates e under γ". The subscript γ will be omitted as γ will always be clear from the context. The above definition is extended to cover functions.

DEFINITION 4.1.8 **safe approximation of functions**
Let the domain of appr be $(D \to D') \times (E \to E')$. *We define* $appr(F', F)$ iff $\forall (d, e) \in D \times E \quad appr(d, e) \Rightarrow appr(F'd, Fe)$.

The following result is also taken from Marriott and Søndergaard [72].

THEOREM 4.1.9 *Let* (γ, D, E) *be an insertion, and let* $F : E \to E$ *and* $F' : D \to D$ *be such that* $appr(F', F)$ *holds. Then*

- $appr(lfp\,(F'), lfp\,(F))$ *holds.*

The idea is to have the "standard" semantics of program P given as $lfp\,(F)$ for some function F, and to have dataflow analysis defined in terms of "nonstandard" functions F' that approximate F.

From the previous theorem we conclude that all elements of $lfp\,(F)$ have some property Q, provided all elements of $\gamma(lfp\,(F'))$ have property Q, where F and F' are the functions in which the "standard" and "non-standard" semantics are respectively defined. $lfp\,(F')$ provides us with approximate information about the standard semantics $lfp\,(F)$.

4.2 Useless Clauses

Given a logic program, it is desirable to detect clauses that yield no solutions. Normally a programmer would try not to write a program containing useless clauses, but if such useless clauses exist, it could indicate a bug, since these clauses contribute nothing to successful computations. Such clauses might also be considered "badly-typed". Related topics are discussed in [79, 71, 110, 111].

Useless clauses arise more frequently in programs generated by transformation systems. In the previous chapter our aim was to transform a program into an equivalent program, but with the different instances of inference rules separated out into different clauses. If this is not done, one successful instance of an inference rule may force a general version of the inference rule to be generated in the specialised program. Many unsuccessful instances of this inference rule may therefore be merged with the successful instance which may lead to many (possibly expensive) useless computations. If the different instances of an inference rule can be separated out and we have a way of identifying the unsuccessful instances,

removal of the unsuccessful instances may lead to a faster computation. In general, it is undecidable if such an instance of an inference rule is useless or not.

In the rest of this section we formally define a useless clause and give some results with respect to the procedural semantics of the original program and the program with useless clauses deleted.

DEFINITION 4.2.1 **useless clause**
Let P be a normal program and let $C \in P$ be a clause. Then C is useless if for all goals G, C is not used in any SLDNF-refutation of $P \cup \{G\}$.

A stronger notion of uselessness is obtained by considering particular computations and a fixed computation rule.

DEFINITION 4.2.2 **useless clause with respect to a computation**
Let P be a normal program, G a normal goal, R a safe computation rule and let $C \in P$ be a clause. Let T be the SLDNF-tree of $P \cup \{G\}$ via R. Then C is useless with respect to T if C is not used in refutations in T or any sub-refutation (for a negation as failure step) associated with T.

A clause that is useless by Definition 4.2.1 is also useless with respect to any goal by Definition 4.2.2.

Since loops may be eliminated by the deletion of useless clauses, the SLDNF procedural semantics of the original and specialised program is generally not preserved. The following correctness result is however proved by de Waal and Gallagher [26]: the results of all finite computations are preserved.

THEOREM 4.2.3 **preservation of all finite computations**
Let P be a normal program and G a normal goal. Let $C \in P$ be clause and let $P' = P - \{C\}$. If C is useless with respect to the computation of $P \cup \{G\}$, then

1. *if $P \cup \{G\}$ has an SLDNF-refutation with computed answer θ then $P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ; and*
2. *if $P \cup \{G\}$ has a finitely failed SLDNF-tree then $P' \cup \{G\}$ has a finitely failed SLDNF-tree.*

For the specialisation of proof procedures this is all that is needed. For general logic programs there may be cases where the original procedural semantics need to be preserved. However, this correctness result may also be useful for many general logic programs.

4.3 Regular Approximations

Regular structures have a number of decidable properties and associated computable operations [2], that make them suited for use as approximations to logic programs [53, 77, 110]. Comprehensive accounts of how to define regular safe approximations of logic programs can be found in [77, 110, 47, 34, 53]. Most of this work considered the definition and precision of various different approximations. In this section we concentrate on two implementations of regular safe approximations. First, we define a regular safe approximation and second, discuss the major differences between the two implementations.

The following definitions are taken from [42].

DEFINITION 4.3.1 **canonical regular unary clause**
A **canonical regular unary clause** *is a clause of the form*

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n) \qquad n \geq 0$$

where $x_1, \ldots, x_n$ *are distinct variables.*

DEFINITION 4.3.2 **canonical regular unary logic program**
A **canonical regular unary logic (RUL) program** *is a finite set of regular unary clauses, in which no two different clause heads have a common instance.*

The class of canonical RUL programs was defined by Yardeni and Shapiro [110]. More expressive classes of regular programs can be defined, but this one allows convenient and efficient manipulation of programs as we will show. In the rest of this thesis the term "RUL program" will be used to denote a "canonical RUL program".

DEFINITION 4.3.3 **regular definition of predicates**
Let $\{p_1/n_1, \ldots, p_m/n_m\}$ *be a finite set of predicates. A* **regular definition** *of the predicates is a set of clauses* $Q \cup R$*, where*

- *Q is a set of clauses of form $p_i(x_{1^i}, \ldots, x_{n^i}) \leftarrow t_{1^i}(x_{1^i}), \ldots, t_{n^i}(x_{n^i})$ $(1 \leq i \leq m)$, and R is an RUL program defining $t_{1^i}, \ldots, t_{n^i}$ such that no two heads of clauses in $Q \cup R$ have a common instance.*

A RUL program can now be obtained from a regular definition of predicates by replacing each clause $p_i(x_{1^i}, \ldots, x_{n^i}) \leftarrow B$ by a clause

$$approx(p_i(x_{1^i}, \ldots, x_{n^i})) \leftarrow B$$

where *approx* is a unique predicate symbol not used elsewhere in the RUL program. In this clause the functor p_i denotes the predicate p_i. Most of the time the *approx* will be absent from our RUL programs as it simplifies the discussion. When it is clear from the context what is meant the *approx* is omitted from our RUL programs. We now define a safe approximation in terms of a regular definition of predicates.

DEFINITION 4.3.4 **regular safe approximation**
Let P be a definite program and P' a regular definition of predicates in P. Then P' is a regular safe approximation of P if the least Herbrand model of P is contained in the least Herbrand model of P'.

We now link regular approximations as defined in this section to the framework of Marriott and Søndergaard [72] described in Section 4.1. The concretisation function γ giving semantics of RUL programs is first defined.

DEFINITION 4.3.5 $\gamma(R)$
Let R be an RUL program and U_R its Herbrand Universe. The concretisation of R, $\gamma(R)$ is the set $\{t \in U_R \mid R \cup \{\leftarrow approx(t)\}$ has a refutation*$\}$.*

The abstraction function α used to compute RUL programs is also called $\mathcal{T}_P$ and is a function that maps one RUL program into another, such that

1. $\forall n\ T_P^n(\emptyset) \subseteq \gamma(\mathcal{T}_P^n(\emptyset))$ and
2. $\mathcal{T}_P^m(\emptyset)$ is a fixed point for some m.

This guarantees that $lfp(T_P) \subseteq \gamma(lfp(\mathcal{T}_P))$. More details about the exact operations used in $\mathcal{T}_P$ can be found in [42]. The concrete meaning of our program P is given by the least

fixed point of the function T_P (which corresponds to the least Herbrand model of P) and the abstract meaning of P is given by the least fixed point of the abstract semantic function $\mathcal{T}_P$ (which corresponds to the least Herbrand model of P').

Two practical systems that can compute regular safe approximations automatically have been discussed by Gallagher and de Waal (we call this system GdW) [42] and Van Hentenryck, Cortesi and Le Charlier (we call their system VHCLC) [48]. Implementation details can be found in both papers, so they are not repeated here. However, we give a brief account of the GdW system for completeness of this chapter. Any one of these two systems can be used in our proposed methods to compute a regular safe approximation (called a safe approximation from now on) of a logic program. The two implementations are however very different and we therefore discuss the differences in some detail. First, we give a summary of the GdW system.

The GdW regular approximation system is based on the framework of bottom-up abstract interpretation of logic programs [72]. We define an abstraction of the T_P function, $\mathcal{T}_P$ that maps one regular program to another. By computing the least fixed point of $\mathcal{T}_P$ we obtain a description of some superset of the least fixed point of T_P.

A number of operations and relations on RUL programs are used. We repeat the following definitions from [42].

DEFINITION 4.3.6 *$success_R(t)$*
Let R be an RUL program and let U_R be its Herbrand universe. Let t be a unary predicate. Then $success_R(t) = \{s \in U_R \mid R \cup \{\leftarrow t(s)\}$ has a refutation$\}$.

DEFINITION 4.3.7 **inclusion**
Let R be an RUL program, and let t_1 and t_2 be unary predicates. Then we write $t_1 \subseteq t_2$ if $success_R(t_1) \subseteq success_R(t_2)$.

Definitions for intersection and upper bound of unary predicates can be found in [42]. The intersection of two predicates t_1 and t_2 is denoted by $t_1 \cap t_2$ and their upper bound by $t_1 \sqcup t_2$.

DEFINITION 4.3.8 **calls, call-chain, depends on**
Let R be a program containing predicates t and s ($t \neq s$). Then t calls s if s occurs in the body of clause whose head contains t. A sequence of predicates $t_1, t_2, \ldots, t_i, \ldots$ where t_i calls t_{i+1} ($i \geq 1$) is a call-chain. We say t depends on s if there is a call-chain from t to s.

DEFINITION 4.3.9 $D(t,s)$
Let R be an RUL program containing predicates t and s ($t \neq s$). Then the relation $D(t,s)$ is true if t depends on s and the set of function symbols appearing in the heads of clauses in the procedure for t is the same as the set of function symbols appearing in the heads of clauses in the procedure for s.

DEFINITION 4.3.10 **shortening** *(adapted from GdW)*
Let R be an RUL program, and let t and s be unary predicates defined in R such that $D(s,t)$ holds. Then a program $Sh(R)$ is obtained from R as follows:

- *Let $r = t \sqcup s$. Replace all occurrences of t in clause bodies by r. Then replace all occurrences of s that depend on r by t.*

DEFINITION 4.3.11 **shortened RUL program**
A **shortened** *program R is one in which there are no two predicates t and s such that $D(s,t)$ holds.*

A shortened program can be obtained from R by performing a finite number of applications of Sh to R, i.e. computing $Sh^n(R)$ for some finite n. This is proved in [40].

DEFINITION 4.3.12 **short**(R)
Let R be an RUL program. Then **short**$(R) = Sh^n(R)$ *such that $Sh^n(R)$ is shortened.*

Shortening is used to limit the number of RUL programs and thus ensure termination of our approximation procedure. A shortened RUL program corresponds to an automaton in which no two connected states have the same set of labels on transitions leading from them. Shortening also puts RUL definitions into a minimal form.

Before we can define $\mathcal{T}_P$, we need to define the regular solution of a definite clause. This is done informally by means of an operator *solve*. The procedure $solve(C,R)$ takes a definite clause C and a regular definition of predicates, R. First, it solves the body of C in R and then produces a regular definition of the predicate in the head of the clause. If *solve* returns $\emptyset$ this is interpreted as failure. Otherwise, the procedure returns a unique result which is an RUL program. The full definition is given in [41].

DEFINITION 4.3.13 $\mathcal{T}_P$
Let P be a definite program. Let D a regular definition of predicates in P. A function $\mathcal{T}_P(D)$ is defined as follows:

$$\mathcal{T}_P(D) = \mathbf{short}(\bigsqcup \left\{ \mathbf{solve}(C, D) \;\middle|\; C \in P \right\} \bigsqcup D)$$

That is, **solve** *is applied to each clause, their upper bound with D is computed and the whole result shortened.*

$\mathcal{T}_P$ has a least fixed point which is finitely computable as $\mathcal{T}_P^n(\emptyset)$ and is a safe abstraction of T_P.

The major difference between the GdW system and the VHCLC system is that VHCLC uses a top-down (query dependent) framework while GdW use a bottom-up (query independent) framework (see Debray [29] for further details). To take query information into account GdW was extended with query-answer transformations (which are discussed in detail in the next section).

A second difference is that VHCLC uses type-graphs and a widening operator to construct the approximation while GdW uses a shortening operator incorporated into $\mathcal{T}_P$. Widening is used to avoid the result of a procedure being refined infinitely often and to avoid an infinite sequence of procedure calls. The widening operator tries to prevent the type graph from growing by introducing loops. The justification for shortening is to limit the number of RUL programs that can be generated and thus ensure termination of the approximation procedure. Shortening also has the effect of introducing loops. The design of widening and shortening operators is an active area of research and new operators may assist in decreasing the approximations times and therefore make these methods more attractive for practical use. VHCLC also uses two graphs (the previous and newly constructed graph) to decide when to introduce a loop while GdW uses only one RUL program (the current one) to decide when to do folding. In VHCLC the concept of a topological clash is used to decide this while the notion of a call-chain is used for RUL programs in GdW. In very special cases this makes GdW slightly inferior to VHCLC (e.g. when approximating lists of lists of lists etc.) as GdW cannot check the growth of certain parts of the RUL program against the RUL from the previous iteration of the fixpoint algorithm (less information is available to base the decision on). However, GdW's shortening operation is less complex than VHCLC's widening operation.

A third difference is that VHCLC is implemented in C while GdW is implemented in Prolog. There is roughly an order of magnitude difference in computation times between

$perm(X,Y) \leftarrow intlist(X), permutation(X,Y)$

$permutation([\,],[\,]) \leftarrow$
$permutation(X,[U|V]) \leftarrow delete(U,X,Z), permutation(Z,V)$

$delete(X,[X|Y],Y) \leftarrow$
$delete(X,[Y|Z],[Y|W]) \leftarrow delete(X,Z,W)$

$intlist([\,]) \leftarrow$
$intlist([X|Xs]) \leftarrow integer(X), intlist(Xs)$

Program 4.1: Permutation

the two systems. Naturally, the C implementation is the faster implementation. Most of this difference can be attributed to the use of the two different implementation languages (see [42] and [48] for performance figures and complexity analyses). However, the existence of faster systems than the one we used for our experiments only strengthens our claims about the usefulness of our approach.

4.4 A Regular Approximation Example

We now give an example of a regular approximation computed by GdW. Consider a variation of the well-known permutation program, but with a call in which the first argument is a list of integers, expressed by the predicate $intlist(X)$. The program is given in Program 4.1. A regular approximation of Program 4.1 computed by the GdW implementation (fully described in [42]) is given in Program 4.2. The t-predicates, *any* and *numeric* are generated by the approximation procedure. *numeric* is a predefined approximation for $integer(X)$. Similarly, other approximations for other built-ins can also be defined. Note that we should actually have $approx(perm(\ldots))$ instead of $perm(\ldots)$ to conform with our formal description of RUL programs. However, no confusion is possible, so we regard the generated program as representing the RUL program. Although the second argument of *perm* is shown to be a list, it cannot be inferred to be a list of integers. This result is imprecise, but no other method of regular approximation in the literature based on bottom-up approximation would give a better result without argument dependency information.

In Section 4.6 we introduce a query-answer transformation to increase the precision of the approximation. A decision procedure for detecting useless clauses using regular approxima-

$delete(X1, X2, X3) \leftarrow any(X1), t11(X2), any(X3)$
$t11([X1|X2]) \leftarrow any(X1), any(X2)$
$perm(X1, X2) \leftarrow t21(X1), t28(X2)$
$t21([X1|X2]) \leftarrow any(X1), any(X2)$
$t21([\,]) \leftarrow$
$t28([\,]) \leftarrow$
$t28([X1|X2]) \leftarrow any(X1), t28(X2)$
$intlist(X1) \leftarrow t42(X1)$
$t42([\,]) \leftarrow$
$t42([X1|X2]) \leftarrow numeric(X1), t42(X2)$
$numeric(numeric) \leftarrow$
$permutation(X1, X2) \leftarrow t42(X1), t28(X2)$
$t42([\,]) \leftarrow$
$t42([X1|X2]) \leftarrow numeric(X1), t42(X2)$
$t28([\,]) \leftarrow$
$t28([X1|X2]) \leftarrow any(X1), t28(X2)$
$numeric(numeric) \leftarrow$

Program 4.2: Bottom-Up Regular Approximation of Permutation

tions is given in the next section.

4.5 Detecting Useless Clauses Using Regular Approximations

In this section we give a decision procedure for deciding if a clause in a normal program P is useless or not. We first need a procedural definition of safe approximation and need to make the connection between safe approximations and useless clauses.

DEFINITION 4.5.1 **safe approximation (procedural definition)**
Let P and P' be normal programs. Then P' is a safe approximation of P if for all definite goals G,

- *if $P \cup \{G\}$ has an SLDNF-refutation then $P' \cup \{G\}$ has an SLDNF-refutation.*

For our purposes we are interested in using approximations to detect useless clauses in P, and for this the following weaker condition is adequate.

- if $P' \cup \{G\}$ has a finitely failed SLDNF-tree then $P \cup \{G\}$ has no SLDNF-refutation.

This definition is equivalent to saying that if a definite goal G succeeds in P then it succeeds, loops or flounders in P'.

DEFINITION 4.5.2 G^+
Let G be a normal goal. Then G^+ denotes the definite goal obtained by deleting all negative literals from G.

THEOREM 4.5.3 **safe approximation for detecting useless clauses**
Let P be a normal program, and P' a safe approximation of P. Let $A \leftarrow B$ be a clause in P. Then $A \leftarrow B$ is useless with respect to P if $P' \cup \{\leftarrow B^+\}$ has a finitely failed SLDNF-tree.

PROOF. Assume $P' \cup \{\leftarrow B^+\}$ has a finitely failed SLDNF-tree. From Definition 4.5.1 $P \cup \{\leftarrow B^+\}$ has no SLDNF-refutation, hence $P \cup \{\leftarrow B\}$ has no SLDNF-refutation, hence $A \leftarrow B$ is not used in any refutation, hence by Definition 4.2.1 it is useless in P. □

This theorem shows that useless clauses are detectable if failure in the approximation is decidable. Useless clauses can be deleted from a program, without losing the results of any terminating computation.

The regular approximation procedure can now be applied to a normal program as follows:

1. Let P be a normal program and P' the program P with all negative literals deleted. P' is a definite program that is a safe approximation of P.
2. Compute a regular approximation of P'.

Useless clauses can now be detected in P'.

Given some program P and an approximation P', the simplest and most straightforward decision procedure for deciding failure in the approximation is a special case of Theorem 4.5.3.

- *Does a definition for some predicate $p(\ldots)$ exist in P but not in the approximation P'? If the answer is yes, all clauses containing positive occurrences of $p(\ldots)$ in the body of a clause in P are useless clauses and can be deleted.*

This can be checked finitely. The resulting program, after deleting useless clauses by Theorem 4.5.3, preserves all finite computations. More complex decision procedures for deciding whether a clause is useless can be constructed by examining arguments to predicates in the regular approximation. Since failure is decidable in regular approximations, our decision procedures will always terminate when we check (using Theorem 4.5.3) if a clause in our program is useless. Note that negative literals cannot be taken into account when constructing a safe approximation of a program, as this would destroy the monotonicity of the $\mathcal{T}_P$ operator.

As the $\mathcal{T}_P$ operation is a bottom-up fixpoint operation, it cannot take information about the queries to a program into account. A query-answer transformation that allows a bottom-up approximation framework to take information about queries into account is described in the next section. This transformation may increase the precision of the approximation and may therefore contribute to the detection and deletion of more useless clauses.

4.6 Query-Answer Transformations

The motivation for introducing this transformation is the following: we want to examine particular computations rather than the success set of a program. If we analyse a program (as it is) with the bottom-up regular approximation procedure from the previous section, we get a safe approximation of the success set of the program, which is in general larger than the information we want for a particular call to the program. By transforming a program with the "magic set" transformation, we restrict attention to certain calls to predicates rather than their general models. This analysis information will in general be more precise and therefore more useful. This may be the case even when the supplied call to a predicate is unrestricted, since calls from within clause bodies may be instantiated.

The oddly-named "magic set" and "Alexander" transformations on which this transformation is based were introduced as recursive query evaluation techniques [4, 92], but have since been adapted for use in program analysis [19, 28, 55, 95, 40]. We prefer the name

$reverse([\,],[\,]) \leftarrow$
$reverse([X|Xs],Ys) \leftarrow$
$\quad reverse(Xs,Zs),$
$\quad append(Zs,[X],Ys)$

$append([\,],Ys,Ys) \leftarrow$
$append([X|Xs],Ys,[X|Zs]) \leftarrow$
$\quad append(Xs,Ys,Zs)$

Program 4.3: Naive Reverse

$reverse(X1,X2) \leftarrow t38(X1), any(X2)$
$t38([\,]) \leftarrow$
$t38([X1|X2]) \leftarrow any(X1), t38(X2)$
$append(X1,X2,X3) \leftarrow t22(X1), any(X2), any(X3)$
$t22([\,]) \leftarrow$
$t22([X1|X2]) \leftarrow any(X1), t22(X2)$

Program 4.4: Bottom-Up Regular Approximation of Naive Reverse

query-answer transformations and will introduce several versions of the basic transformation capturing query and answer patterns to various levels of precision. In general, the more precise analysis requires a more complex query-answer transformation.

The intuitive idea behind this method will be explained with the well-known naive reverse program given in Program 4.3. A bottom-up regular approximation using GdW gives an approximation of the success set of the program. The approximation is given in Program 4.4. The t-predicates and *any* are generated by the regular approximation procedure. As can be seen, the approximation is imprecise and has lost all information about the second argument to *reverse* and the second and third arguments to *append*. This is all a bottom-up approximation can do as it does not know the arguments to *append*. This then propagates up into *reverse* and we get *any* for the second argument to reverse.

Assuming a left-to-right computation rule and a top-level goal $\leftarrow reverse(\_,\_)$, we can restrict the calls to *append* to those calls that will be generated by *reverse* using a left-to-right computation rule. This should give us a more precise approximation for *append* and therefore also for *reverse*.

We reason as follows (assuming a left-to-right computation rule):

(i) for $reverse([\,],[\,])$ to succeed, $reverse([\,],[\,])$ must be called.

(ii) For $reverse([X|Xs],Ys)$ to succeed, $reverse([X|Xs],Ys)$ must be called, $reverse(Xs,Zs)$ must succeed and $append(Zs,[X],Ys)$ must succeed.

(iii) For $append([\,],Ys,Ys)$ to succeed, $append([\,],Ys,Ys)$ must be called.

(iv) For $append([X|Xs],Ys,[X|Zs])$ to succeed, $append([X|Xs],Ys,[X|Zs])$ must be called and $append(Xs,Ys,Zs)$ must succeed.

(v) For $reverse(Xs,Zs)$ to be called $reverse([X|Xs],Ys)$ must be called.

(vi) For $append(Zs,[X],Ys)$ to be called, $reverse([X|Xs],Ys)$ must be called and $reverse(Xs,Zs)$ must succeed.

(vii) For $append(Xs,Ys,Zs)$ to be called, $append([X|Xs],Ys,[X|Zs])$ must be called.

The program in Program 4.5 gives a "query-answer" transformed version of the program in Program 4.3. A meta-program describing the above query-answer transformation was described by Gallagher and de Waal [40]. We added the clause

$$reverse\_query(X,Y)$$

to the program to restrict the query and answer patterns to those generated when *reverse* is called as the top-level goal.

A regular approximation of the query-answer transformed program given in Program 4.5 is given in Program 4.6. The regular approximation of the second argument to *append* is now a singleton list and the second argument to *reverse* is a list. This is much more precise than the bottom-up regular approximation of the original program.

For some programs, the regular approximation procedure might still be unable to infer optimal results even for a query-answer transformed program. An example of such a program is the *permutation* program with the second argument to *permutation* specified to be a list of integers. We want the regular approximation to detect that the first argument to *permutation* is also a list of integers. The program is similar to the program given in Program 4.1, but with the second argument of *permutation* specified to be a list of integers, instead of the first.

The reason for inferring non-optimal results is that the recursive call to *perm* in the definition of *perm* is not taken into account when an approximation of *delete* is computed,

$reverse\_ans([\,],[\,]) \leftarrow reverse\_query([\,],[\,])$
$reverse\_ans([X1|X2],X3) \leftarrow reverse\_query([X1|X2],X3),$
$\quad reverse\_ans(X2,X4),$
$\quad append\_ans(X4,[X1],X3)$
$append\_ans([\,],X1,X1) \leftarrow append\_query([\,],X1,X1)$
$append\_ans([X1|X2],X3,[X1|X4]) \leftarrow append\_query([X1|X2],X3,[X1|X4]),$
$\quad append\_ans(X2,X3,X4)$
$append\_query(X1,[X2],X3) \leftarrow reverse\_query([X2|X4],X3),$
$\quad reverse\_ans(X4,X1)$
$reverse\_query(X1,X2) \leftarrow reverse\_query([X3|X1],X4)$
$append\_query(X1,X2,X3) \leftarrow append\_query([X4|X1],X2,[X4|X3])$
$reverse\_query(X1,X2) \leftarrow$

Program 4.5: Query-Answer Transformed Program

because *perm* is executed after *delete* using a left-to-right computation rule. But, as we are interested only in successful computations of *perm*, we do not need to include in the approximation instances of answers to *delete* that can never occur in any successful query to *permutation*. This is exactly what we do with the query-answer transformation described earlier.

As we are interested only in an approximation of the success set of a program, the left-to-right computation rule we used is no longer a prerequisite. An analysis using a right-to-left (or any other) computation rule is just as valid. We now also generate a query-answer transformation using a right-to-left computation rule. We therefore have two answer and query predicates for each predicate occurring in our program, one for the left-to-right and one for the right-to-left computation rule. The intersection of the regular approximations of these two computation rules may now give us even more precise information, as information from the right of a clause may now be propagated to the literals to its left.

To capture all the information possible, we need to consider all possible computation rules. This is clearly very inefficient and we have found that analysing the above two computation rules gives near optimal or optimal results for most programs. Also, the two approximations using the two computation rules can be performed independently (this is detected by the decomposition of the predicate call graph into strongly connected components [42]) and are combined only at the final step. The approximation time will therefore approximately double, but it might be an acceptable price to pay for the increase in precision.

A regular approximation of the *permutation* program with a list of integers as the second

$reverse\_query(X1, X2) \leftarrow any(X1), any(X2)$
$append\_query(X1, X2, X3) \leftarrow t66(X1), t69(X2), any(X3)$
$t66([\,]) \leftarrow$
$t66([X1|X2]) \leftarrow any(X1), t66(X2)$
$t69([X1|X2]) \leftarrow any(X1), t70(X2)$
$t70([\,]) \leftarrow$
$append\_ans(X1, X2, X3) \leftarrow t84(X1), t69(X2), t85(X3)$
$t84([\,]) \leftarrow$
$t84([X1|X2]) \leftarrow any(X1), t84(X2)$
$t69([X1|X2]) \leftarrow any(X1), t70(X2)$
$t85([X1|X2]) \leftarrow any(X1), t88(X2)$
$t70([\,]) \leftarrow$
$t88([X1|X2]) \leftarrow any(X1), t88(X2)$
$t88([\,]) \leftarrow$
$reverse\_ans(X1, X2) \leftarrow t34(X1), t66(X2)$
$t34([\,]) \leftarrow$
$t34([X1|X2]) \leftarrow any(X1), t34(X2)$
$t66([\,]) \leftarrow$
$t66([X1|X2]) \leftarrow any(X1), t66(X2)$

Program 4.6: Regular Approximation of Query-Answer Transformed Program

argument using the improved query-answer transformation enables us to infer the most precise RUL approximation possible for this program. The approximation is given in Program 4.7. We do not give the complete approximation, but only the intersection of regular approximations using a left-to-right and a right-to-left computation rule.

The overhead introduced by the query-answer transformation has been found to be substantial, usually between a factor of three to seven times slower for most reasonable programs. The reduction of this overhead is the subject of ongoing research and we briefly sketch possible ways to reduce this overhead.

- Construct an "ideal" computation rule that captures the flow of information in a program. The query-answer transformation is then based on this one computation rule. A reduction in complexity (and precision) compared to a query-answer transformation based on more than one computation rule may result during the approximation.
- Do a fast bottom-up approximation first and use this as a starting point for a more

$perm\_ans(X1, X2) \leftarrow t114(X1), t91(X2)$
$t114([\,]) \leftarrow true$
$t114([X1|X2]) \leftarrow numeric(X1), t114(X2)$
$t91([\,]) \leftarrow true$
$t91([X1|X2]) \leftarrow numeric(X1), t91(X2)$
$numeric(numeric) \leftarrow true$
$delete\_ans(X1, X2, X3) \leftarrow numeric(X1), t125(X2), t114(X3)$
$numeric(numeric) \leftarrow true$
$t125([X1|X2]) \leftarrow numeric(X1), t114(X2)$
$t114([\,]) \leftarrow true$
$t114([X1|X2]) \leftarrow numeric(X1), t114(X2)$
$intlist\_ans(X1) \leftarrow t49(X1)$
$t49([\,]) \leftarrow true$
$t49([X1|X2]) \leftarrow numeric(X1), t49(X2)$
$numeric(numeric) \leftarrow true$

Program 4.7: Regular Approximation of Permutation with Second Argument a List of Integers

complex analysis based on one or more computation rules. This can all be done by writing variations of the query-answer transformation introduced at the beginning of this section.

Chapter 5

Specialisation Methods for Proof Procedures

The improved partial evaluation procedures may be combined with the the detection of useless clauses to form specialisation methods that reason about safe approximations of proof procedures, theories and formulas (queries). This combination is presented in this chapter, Section 5.2. Before that, we review the research of Brüning [14] which, as far as we know, is the only research with directly similar aims to those of this thesis. The chapter ends with an automatic proof of one of the theorems by Poole and Goebel [85] stating that: given a first-order theory T consisting of a set of definite clauses, ancestor resolution (search) is never needed to prove any theorem in T.

5.1 Detecting Non-Provable Goals

In Brüning's work, "Detecting Non-Provable Goals" [14], he tries to identify goals which cannot contribute to a successful derivation (i.e. refutation). The way in which he approaches the problem is very different from the approach presented in this thesis. However, there are also some similarities.

The approach presented by Brüning has its roots in cycle unification [13]. It deals with the problem of how to compute a complete set of answer substitutions for a goal $\leftarrow p(s_1, \ldots, s_n)$ with respect to a logic program consisting of one fact $p(t_1, \ldots, t_n)$ and a recursive two-literal clause $p(l_1, \ldots, l_n) \leftarrow p(r_1, \ldots, r_n)$. Variable dependencies are studied to determine the maximum number of required iterations through the cycle. However, even for this small class of programs this is in general undecidable if the cycle is unrestricted, i.e. $l_1, \ldots, l_n, r_1, \ldots, r_n$

are arbitrary terms.

The idea of trying to determine what may happen to a goal G during some derivation (given some clause set) was adapted for the detection of non-provable goals. The approach analyses how a term may change during a derivation by "adding" and "deleting" function symbols. Using this information makes it possible to detect derivation chains that cannot contribute to a refutation and can therefore be pruned. This idea is similar to ours as we try to detect clauses that can never contribute to a successful derivation in a proof.

The first step in his approach is to construct so-called position graphs that encode the possible changes to terms. These position graphs are then translated to monadic clause sets. Ordered resolution (with an ordering imposed on terms) [32] are then used as a decision procedure to detect non-provable goals. Instead of monadic clause sets and ordered resolution we use regular approximations and a simple decision procedure based on the existence of a regular approximation.

Sharing information is not in general taken into account in our method: $t_1, \ldots, t_n$ are different variables in $approx(p(t_1, \ldots, t_n))$. This is one area where we may potentially lose specialisations compared to the system proposed by Brüning. However, the incorporation of sharing information into our regular approximation procedure is an area of ongoing research and we hope to incorporate it into our procedure in the near future.

Both methods seem to be tractable for the examples studied so far and experimental results from experiments performed at Bristol on interesting problems suggested by Brüning [15] have not provided conclusive evidence that one system is more accurate than the other.

Although both methods are aimed at first-order logic, the methods developed in this thesis are more general as we can analyse any logic with our methods, as long as a proof procedure for it can be written as a logic program. The current implementation of the method described in this thesis is also more advanced and we have analysed many problems from the TPTP Problem Library and shown the usefulness of our method (see Chapter 6 and [27]).

5.2 Specialisation Method

We present three versions of a basic specialisation method based on the techniques developed in the previous chapters. The first version is very general and should give the best specialisations possible. The second version is tailored towards users of existing theorem provers and users not having access to a sophisticated partial evaluator. The third version

Given some proof procedure P (implemented as a meta-program), a representation of an object theory, T (a finite set of clauses), and a representation of a formula, F, the procedure for creating a version of P specialised with respect to T and F is as follows:

1. **Perform a partial evaluation of P with respect to T and F', where F is an instance of F'. Let the result be P_1.**
2. **Generate a query-answer version of P_1 and F'. Let this be P_2.**
3. **Derive a regular approximation A of P_2.**
4. **Use the decision procedure given in the previous chapter to delete useless clauses from P_1 using A, giving P_3.**

Figure 5.1: GENERAL

is interesting from a logic programming point of view, as it combines partial evaluation of a program written in a ground representation with abstract interpretation. However, this last method is also the most complicated of the three.

5.2.1 General Method—GENERAL

It is important to state that the proposed specialisation methods are independent of the logic of the proof system. As long as the chosen proof procedure can be written as a logic program the specialisation methods can be applied. This makes our specialisation methods more general than some of the other methods designed for one specific proof system [85, 107, 103, 56].

Any approximation where failure is decidable (with corresponding decision procedure) can be substituted for our regular approximations to give a less precise or more precise specialisation method based on the precision of the approximation. A general version of the specialisation method is shown in Figure 5.1. By the correctness of partial evaluation and by Theorem 4.5.3, we have the following properties.

1. If $P \cup \{G\}$ has an SLDNF-refutation with computed answer θ then $P_3 \cup \{G\}$ has an SLDNF-refutation with computed answer θ; and

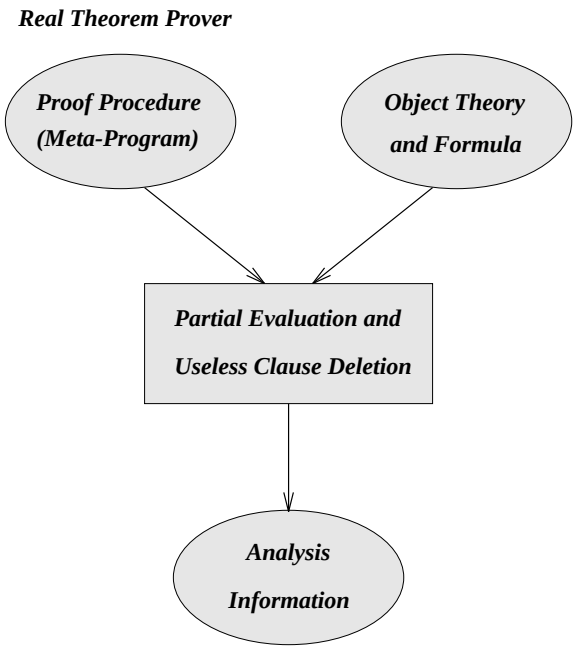


Figure 5.2: Inferring Analysis Information for Theorem Provers

2. If $P \cup \{G\}$ has a finitely failed SLDNF-tree then $P_3 \cup \{G\}$ has a finitely failed SLDNF-tree.

The specialisation thus preserves all the results of terminating computations. The effectiveness of the second stage depends partly on the partial evaluation procedure employed. Some renaming of predicates to distinguish different calls to the same procedure is desirable, since some calls to a procedure may fail or loop while other calls succeed (see Chapter 3). Renaming allows the failing branches to be removed while retaining the useful ones. If partial evaluation (and renaming) is not done then useless clauses are much less likely to be generated.

5.2.2 Analysis for Theorem Provers—KERNEL

The user who is interested only in deriving analysis information for his or her theorem prover (possibly implemented in another programming language) might not want an efficient runnable program from the proposed specialisation method. Analysis information that can be used to "tune" some other theorem prover may be all that is desired from this method. We envisage the situation depicted in Figure 5.2. The proof procedure that we are going to

Given some specification of a proof procedure P, a representation of an object theory, T, and a representation of a formula, F, the procedure for inferring analysis information about P, T and F is:

1. **Apply the minimal partial evaluator to P and T. Let the result be P_1.**
2. **Generate a query-answer version of P_1 and F', where F is an instance of F'. Let this be P_2.**
3. **Derive a regular approximation A of P_2.**
4. **Use the decision procedure given in the previous chapter to record useless clauses in P_1 .**

Figure 5.3: KERNEL

analyse may be regarded as a specification of the "real" theorem prover. Analysis results inferred by using this method on the specification of the theorem prover, object theory and formula may then be used to optimise the real theorem prover. The method given in the previous section may then be revised to make the analysis as fast as possible and we get the revised specialisation method given in Figure. 5.3.

It should now be straightforward to use this analysis information to set some flags in the "real" theorem prover to exploit these results (examples of the application of this method will be given in the next chapter). By the correctness of partial evaluation and by Theorem 4.5.3, we have the same properties as for the general method given in the previous section.

The reason for using the minimal partial evaluator is to preserve the basic structure of the meta-program. This makes it easier to relate information inferred for the partially evaluated program P_1 back to the meta-program (and therefore also the inference rules in the proof procedure). It also assists in the identification of clauses in the object theory that are not needed to prove a given theorem. If the structure of the meta-program is not preserved, it may be difficult to draw conclusions from the detection of useless clauses in P_1 as we may not know from which meta-program clauses and object theory clauses these useless clauses are derived.

This method should give a fast analysis as the creation of renamed versions of specialised inference rules are tightly controlled by the application of the minimal partial evaluator. The need for a general partial evaluator is also absent which may make this method widely usable in many different theorem proving environments.

Given some proof procedure P, a representation of an object theory, T, and a representation of a formula, F, the procedure for creating a version of P specialised with respect to T and F is as follows:

1. **Apply the minimal partial evaluator to P and T. Let the result be P_1.**
2. **Generate a query-answer version of P_1 and F', where F is an instance of F'. Let this be P_2.**
3. **Derive a regular approximation A of P_2.**
4. **Use the decision procedure given in the previous chapter to delete useless clauses from P_1, giving P_3.**
5. **Partially evaluate P_3 with respect to F'. Let the result be P_4.**

Figure 5.4: MARSHAL

5.2.3 Specialising the Ground Representation—MARSHAL

The previous methods are both correct for a theorem prover implemented using a ground representation. But it may be more efficient to leave the specialisation of ground unification until last (approximation may delete many useless clauses which could have been unnecessarily partially evaluated). An example illustrating this point will be given in the next chapter. We therefore propose the following third version of the specialisation method especially suited to the specialisation of proof procedures implemented using a ground representation. It is given in Figure 5.4.

Step 5 specialises ground unification. A general partial evaluator may be used for this task or a specialised "ground unification specialiser". If a general partial evaluator is used, some specialisations may be "redone" which could be inefficient. The same properties hold as given for the previous methods.

We presented three methods for analysing and transforming proof procedures for first-order logic with respect to some object theories and formulas. The precision of the above methods can be improved as follows:

- increase the amount of static information added to the meta-program by adding more "type" atoms to the meta-program;

- regular approximation can be made more precise or replaced by a more precise approximation;
- the decision procedure of the previous chapter can be improved by looking at arguments of the regular approximation definitions;
- the query-answer transformation can be improved;
- the partial evaluation can be improved.

The implementation used in this thesis, based on the SP-system by Gallagher and the regular approximation system GdW by Gallagher and de Waal, can therefore be regarded as a first realisation of the use of approximation to detect useless clauses in theories and useless inference rules in proof procedures. Much research however still remains to be done to exploit and improve this basic idea. Examples of the applications of all three methods are given in the next section and the next chapter.

5.3 Proving Properties about Proof Procedures with respect to Classes of Theories

In Chapter 1 we indicated that it might be useful to analyse the behaviour of a proof procedure on well identified classes of theories. The analysis information may indicate that certain inference rules are not needed to prove theorems belonging to a certain class.

The way in which this is done is as follows.

- The problem is stated as a first-order theory.
- The model elimination proof procedure, Program 2.2, is specialised with respect to this theory and a formula with KERNEL.
- The inferred information about useless clauses is examined. This information should contain the proof we are interested in.

We illustrate how this may be done by giving an automatic proof of the theorem by Poole and Goebel that ancestor resolution is never needed to prove any theorem in a definite (Horn) theory [85].

THEOREM 5.3.1 **redundant ancestor resolution**

Let T be a first-order theory expressed as a possibly infinite set of definite clauses. Let P be the model elimination proof procedure consisting of two inference rules input resolution (I) and ancestor resolution (A). For any theorem F of T, where F is an atom, A is never used successfully in any proof of F. Furthermore, every application of I resolves upon the positive literal in a clause in T.

PROOF. We give an automatic proof of the theorem using KERNEL. The proof is done in six steps. First, write down the problem as a first-order theory. The domain of interpretation D is the set of predicates in T. The predicate and function symbols have obvious meanings. *anc* is the definition for ancestor resolution, *clause* for a definite clause and *contr* for a contrapositive of a definite clause. We assumed that a clause is a contrapositive of itself. $+(\_)$ indicates a positive literal (an atom) and $-(\_)$ a negative literal (a negated atom).

(1) $atom(+(\_))$

(2) $not\_atom(-(\_))$

(3) $clause(P) \vee \neg atom(P)$

(4) $clause(or(P,D)) \vee \neg atom(P) \vee \neg body(D)$

(5) $body(D) \vee \neg not\_atom(D)$

(6) $body(or(D,R)) \vee \neg not\_atom(D) \vee \neg body(R)$

(7) $contr(P,P) \vee \neg atom(P)$

(8) $contr(or(P,D),or(X,Y)) \vee \neg clause(or(P,D)) \vee \neg pick\_a\_head(X,or(P,D),Y)$

(9) $pick\_a\_head(X,or(X,Y),Y) \vee \neg not\_atom(Y)$

(10) $pick\_a\_head(Y,or(X,Y),X) \vee \neg not\_atom(Y)$

(11) $pick\_a\_head(X,or(X,or(Z,W)),or(Z,W))$

(12) $pick\_a\_head(X,or(Y,or(Z,W)),or(Y,R)) \vee \neg pick\_a\_head(X,or(Z,W),R)$

(13) $not(+(\_),-(\_))$

(14) $not(-(\_),+(\_))$

(15) $infer(X,Y) \vee \neg anc(X,Y)$

(16) $infer(X,Y) \vee \neg clause(X) \vee \neg contr(C,X) \vee \neg atom(X)$

(17) $infer(X,Y) \vee \neg clause(C) \vee \neg contr(C,X) \vee \neg not\_atom(X)$

(18) $infer(X,Y) \vee \neg clause(Z) \vee \neg contr(Z,or(X,W)) \vee$
$\neg not(X,A) \vee \neg inferall(W,and(A,Y))$

(19) $inferall(W,A) \vee \neg not\_atom(W) \vee \neg not(W,Z) \vee \neg infer(Z,A)$

(20) $inferall(W,A) \vee \neg atom(W) \vee \neg not(W,Z) \vee \neg infer(Z,A)$

(21) $inferall(or(W,Y),A) \vee \neg not(W,Z) \vee \neg infer(Z,A) \vee \neg inferall(Y,A)$

(22) $anc(X,and(X,Y))$

(23) $anc(X,and(Y,Z)) \vee \neg atom(Y) \vee \neg anc(X,Z)$

(24) $anc(X,and(Y,Z)) \vee \neg not\_atom(Y) \vee \neg anc(X,Z)$

(25) $\neg infer(+(\_),empty)$

Clauses 1 to 12 are the axioms defining definite clauses and contrapositives of definite clauses while clauses 13 to 24 are the axioms of the model elimination proof procedure. Clause 25 is the negation of the theorem we want to prove. Although we are specialising a version of the model elimination proof procedure with respect to itself, the model elimination proof procedure as a meta-program should not be confused with the occurrence of the same proof procedure in the object theory. They are written in different languages: a logic programming language and a first-order clausal language.

Second, the model elimination theorem prover P given in Program 2.2 is partially evaluated with respect to the theory T giving P_1 (P_1 is given in Appendix A).

Third, a query-answer transformed version of P_1 with respect to the goal $\leftarrow solve(infer(+(\_), empty), [])$ is generated (we wish to approximate with respect to any theorem p that can be expressed as an atom, and no ancestor resolution steps indicated by $empty$). This gives P_2.

Fourth, a regular approximation A of P_2 is computed[1] (A is given in Appendix B). Fifth, the decision procedure of Chapter 5 is used to identify useless clauses.

Applying the decision procedure on A and P_1 concludes that all clauses of the form

$$prove\_91(++anc(+(\_),\_),\_) \leftarrow \ldots$$
$$prove\_92(++anc(-(\_),\_),\_) \leftarrow \ldots$$
$$prove\_191(--anc(+(\_),\_),\_) \leftarrow \ldots$$
$$prove\_192(--anc(-(\_),\_),\_) \leftarrow \ldots$$

in P_1 are useless clauses. This implies the following:

- The atom $anc(X,Y)$ and literal $\neg anc(X,Y)$ can never contribute to a proof. Object theory clauses 15,22,23 and 24 can therefore never contribute to a proof of any theorem $+(\_)$.

By the correctness properties given in Section 5.2.1, we have therefore proved that ancestor resolution can never successfully be used in the proof of any formula F, where F is an atom.

Sixth, the regular approximation for *contr* ($prove\_5$) in Appendix B is inspected. From the regular definition we are able to infer that the definite clauses in our theory are the only clauses that need to be considered in any proof. QED □

[1] It took 141.34 seconds on a Sun Sparc-10 running SICStus Prolog to compute the regular approximation.

The identification of clauses 15,22,23 and 24 as useless, indicates that we have redundant clauses in our theory. For this particular problem, these clauses are associated with inference rules. Their removal may make faster proof possible.

THEOREM 5.3.2 **redundant ancestor and input resolution**
Let T be a first-order theory expressed as a possibly infinite set of definite clauses. Let P be the model elimination proof procedure consisting of two inference rules input resolution (I) and ancestor resolution (A). For any theorem F, where F is the negation of an atom, I and A are never used successfully in any proof of F.

PROOF. *An automatic proof similar to the previous proof may be given.* □

These proofs generalise the proofs given by Poole and Goebel in [85]. These are non-trivial theorems that have now been proved automatically in just over two minutes.

These proofs show how a description of a subclass of a problems (e.g. a problem expressed as definite clauses) may be used to identify a simpler proof procedure that will be sound and complete for this subclass. This may be used to speed up the proof process for instances of this subclass.

Chapter 6

Transformation and Analysis Results

"Kind of crude, but it works, boy, it works!"[1]

In this chapter we present our specialisation results for the three proof procedures introduced in Chapter 2. These proof procedures are specialised with respect to twenty five problems from the Thousands of Problems for Theorem Proving (TPTP) Problem Library [104], two problems from the literature and two interesting new problems.

Our main aim is to infer results that may be used in state-of-the-art theorem provers to obtain faster proofs. We therefore do not generate runnable programs in the first part of this chapter where we use KERNEL as our specialisation method. However, in the latter part of this chapter we demonstrate how the other two specialisation methods, GENERAL and MARSHAL may be used to generate runnable programs.

All the analysis times given for the chosen benchmarks are in seconds obtained on a Sun Sparc-10 running SICStus Prolog. The partial evaluation times are not included in the analysis times as theorem independent partial evaluations are done (e.g. we partially evaluate with respect to $\leftarrow solve(\_, \_)$ for model elimination). The partial evaluation time can therefore be divided between all the theorems proved for a particular theory and are therefore ignored. Furthermore, it is also insignificant compared to the regular approximation times.

[1] Alan Newell to Herb Simon, Christmas 1955.

In the next section we infer analysis results for the model elimination proof procedure specialised with respect to at least one problem from each of the domains in the TPTP Problem Library (e.g. Henkin Models, Number Theory, etc.). The benchmarks used are listed in Table 6.1. Two new problems from this thesis are also included. The results are then briefly summarised and complexity issues considered. In Section 6.2 we specialise the nH-Prolog proof procedure with respect to a problem first described by Loveland [69]. An infinitely failed computation is turned into a finitely failed computation. Two problems generated by the "compiled graph" variation of the semantic tableau proof procedure are optimised in Section 6.3. The application of MARSHAL to the model elimination proof procedure (using the ground representation, Program 2.3) and a problem from the TPTP Problem Library listed in Table 6.1 is next described. The results are compared with those obtained in Section 6.1.

6.1 Analysis Results for Model Elimination

The model elimination proof procedure can be specialised with respect to a given theorem (e.g. queries of the form $solve(c(a), [\,])$, with respect to a specific class of theorems (e.g. queries of the form $solve(c(\_), [\,])$ or with respect to all theorems (e.g. queries of the form $solve(\_, [\,])$). The more limited the theorems (queries), the more precise the results are likely to be for a given specialisation method. In the benchmarks that follow, we generally approximate with respect to a given theorem. The analysis results therefore hold only for the given theorem used in the approximation. If we want to prove a theorem different from that used in the approximation, we have to redo the approximation with respect to the new theorem.

As indicated in the introduction, at least one result from each of the domains appearing in the TPTP Problem Library is presented. Some of the typical benchmark problem sets [67, 82] used in other papers could also have been chosen, but we chose not to do this for the following reasons.

- Some of the problems in these benchmark sets have the same structure or are related and good (or bad) analysis results inferred for one of the problems will usually also hold for the others.
- The "ideal" is to analyse *all* 2295 problems in the TPTP Problem Library and make the analysis results available for use within the theorem proving community. This however is a massive task as analysis of some of the more complex problems may take hours. The problems were therefore chosen to show a wide applicability of the developed techniques to automated theorem proving problems and should be regarded

as the first steps in an exhaustive analysis of the problems in the TPTP Problem Library. This is therefore an attempt to get away from current benchmark problem sets and use the TPTP Problem Library as a problem set.

All the analysis and transformation results that we present in Table 6.1 were obtained with KERNEL, the improved query-answer transformation using a left-to-right and right-to-left computation rule and an implementation of the regular approximations [42] based on the description given in Chapter 4.

In all cases the regular approximation was restricted to the theorem indicated in the TPTP Problem Library. If the theorem represented more than one clause in the problem, any one of the clauses can be chosen as a starting point for a proof. We chose the last clause (in the order given in the TPTP Problem Library). Also, when the goal to be proved consisted of more than one literal (e.g. $\leftarrow c(a), c(b)$), we introduced a unique literal $fail$ as top-level goal and added the clause $fail \vee \neg c(a) \vee \neg c(b)$ to our object theory. This gave us a single literal, $fail$, to use in the approximation.

In order to show the significance of the results and facilitate a more meaningful discussion, we will reprint some of the problems from the TPTP Problem Library. In all the following problems, ++ as used in the TPTP Library (which indicates a positive literal (atom)) is deleted and --, which indicates a negative literal (negated atom) is replaced by $\neg$. This is just for ease of presentation and ++ and -- are used during the actual specialisation process (see Appendix A and Appendix B). The TPTP input clauses are therefore used as they appear in the TPTP Problem Library, except for the possible introduction of an extra ++ $fail \vee \ldots$ clause as indicated above. The clause order is as in the TPTP Problem Library.

Table 6.1 contains the TPTP-number, problem description, number of clauses, whether it is a Horn problem or not, analysis time and an indication of whether interesting results were inferred for the problem or not. The benchmark table is sorted by the significance of the results: the non-Horn problems are listed before the Horn-problems (as they are assumed to be more complex to analyse). If interesting results were inferred, the results will be explained in a following section in the order on which they appear in Table 6.1. The caption *Table* is used to indicate a table of results, *Program* a logic program and *Theory* a first-order theory.

For eleven of the twenty seven analysed problems interesting results were inferred. We now discuss each result in some detail (except the model elimination example discussed in detail in Section 5.3). We also present a second new example (Theorem 5.3.1 was the first new example) as a first-order object theory and formula and put it as a challenge to other

Analysis and Transformation Results						
	TPTP Syntactic Name	Description	Clauses	Horn	Time	Results
1.	MSC002-1	Blind Hand	14	×	66.1	√
2.	MSC008-1	Graeco-Latin Square	17	×	33.7	√
3.	ANA003-4	Continuous Functions	12	×	34.6	√
4.	TOP001-2	Topology	13	×	50.0	√
5.	PUZ033-1	Winds and Windows	13	×	21.8	√
6.	PUZ031-1	Steamroller	26	×	104.5	√
7.	NUM014-1	Prime Number	7	×	6.0	√
8.	COM002-2	Program Correctness	19	×	8.2	√
9.	CAT007-3	Domain = Codomain	12	×	12.3	√
10.	*N/A*	Model Elimination	25	√	141.3	√
11.	*N/A*	List Membership	10	√	12.5	√
12.	ALG002-1	Ordered Field	14	×	18.9	×
13.	GEO078-5	3 Noncollinear Points	61	×	> 1000	×
14.	GRA001-1	Labelled Graph	12	×	31.8	×
15.	PRV009-1	Hoare's FIND Program	9	×	29.2	×
16.	RNG041-1	Ring Equation	41	×	80.1	×
17.	SET001-1	Superset	9	×	19.3	×
18.	SYN001-4	Signed Propositions	16	×	60.9	×
19.	BOO001-1	B3 Algebra	13	√	1.3	×
20.	COL001-1	Weak Fixed Point	8	√	0.8	×
21.	GRP001-5	Commutativity	7	√	1.4	×
22.	HEN001-1	X/identity = zero	20	√	8.3	×
23.	LAT005-1	Sam's Lemma	31	√	6.7	×
24.	LCL001-1	Heredith Axiom	28	√	8.3	×
25.	LDA001-1	$3 * 2 * U = UUU$	10	√	0.9	×
26.	PLA001-1	Bread	16	√	4.1	×
27.	ROB001-1	Robins Algebra	10	√	1.9	×

Table 6.1: Benchmarks

theorem provers to show that the presented formula is not a theorem of the given theory.

As many of the problems are stated as Horn clauses, the lack of interesting results for these problems should be cautiously interpreted. The reason for this is that for these problems nothing can be inferred about ancestor resolution as it is redundant. This places a limit on the interesting results than can be inferred for these problems[2]. However, the two new problems (indicated by N/A's in Table 6.1) are Horn clause problems. Significant results are inferred for these two problems.

6.1.1 Detailed Explanations of Results

MSC002-1—DBA BHP—A Blind Hand Problem

Consider the object theory given in Theory 6.1. The first thirteen clauses are hypotheses and the last clause is the negation of the formula to be proved. There are seven predicate symbols.

Our aim is to detect unnecessary inference rules and object theory clauses for proving the above formula (theorem). Since this is not a definite clause theory (and cannot be turned into one by reversing signs on literals) it is not clear which ancestor resolutions and object theory clauses are needed.

Step 1: Partially evaluate the model elimination theorem prover with respect to the object theory (the goal $\leftarrow solve(\_, \_)$ (any goal) is implicit). This gives P_1. Renaming is done to distinguish the different *prove* procedures corresponding to the literals that may occur in the object theory. This step is not compulsory, but makes the analysis results much more precise. Experimental evidence has shown that such a renaming strikes a good balance between usefulness and complexity. A less detailed renaming (e.g. only one specialised procedure is created for each positive and negative occurrence of a literal) sometimes fails to capture important information to make the analysis useful. A more precise renaming taking functors in the object theory into account may take a very long time to analyse, without a substantial increase in the precision of the analysis results. Twenty eight different partially specialised procedures, one for every positive and negative literal appearing in the object theory are generated.

Step 2: Generate a query-answer version of P_1 (with respect to a left-to-right and right-

[2]There is an Afrikaans proverb saying: "Jy kan nie bloed uit 'n klip tap nie". Directly translated this means that pouring blood from a stone is impossible: we should not try to infer results where they do not exist.

1. $at(something, here, now)$
2. $\neg hand\_at(Place, Situation) \vee hand\_at(Place, let\_go(Situation))$
3. $\neg hand\_at(Place, Situation) \vee hand\_at(Another\_place, go(Another\_place, Situation))$
4. $\neg hold(Thing, let\_go(Situation))$
5. $\neg at(Thing, here, Situation) \vee red(Thing)$
6. $\neg at(Thing, Place, Situation) \vee at(Thing, Place, let\_go(Situation))$
7. $\neg at(Thing, Place, Situation) \vee at(Thing, Place, pick\_up(Situation))$
8. $\neg at(Thing, Place, Situation) \vee grab(Thing, pick\_up(go(Place, let\_go(Situation))))$
9. $\neg red(Thing) \vee \neg put(Thing, there, Situation) \vee answer(Situation)$
10. $\neg at(Thing, Place, Situation) \vee \neg grab(Thing, Situation) \vee$
    $put(Thing, Another\_place, go(Another\_place, Situation))$
11. $hold(Thing, Situation) \vee \neg at(Thing, Place, Situation) \vee$
    $at(Thing, Place, go(Another\_place, Situation))$
12. $\neg hand\_at(One\_place, Situation) \vee \neg hold(Thing, Situation) \vee$
    $at(Thing, Place, go(Place, Situation))$
13. $\neg hand\_at(Place, Situation) \vee \neg at(Thing, Place, Situation) \vee$
    $hold(Thing, pick\_up(Situation))$
14. $\neg answer(Situation)$

Theory 6.1: MSC002-1—DBA BHP—A Blind Hand Problem

to-left computation rule) and the goal $\leftarrow solve(answer(S), [\,])$, where $answer(S)$ is the formula we want to prove and the empty negative ancestor list $[\,]$ indicates the start of the proof. This gives P_2. P_2 contains three hundred and seventy clauses. We refer the reader to the program given in Program 4.5 for an example of what the query-answer transformed program P_2 will look like.

Step 3: Compute a regular approximation of P_2. This gives A. The regular approximation for $solve(answer(S), [\,])$ is given in the program in Program 6.1 (this is only part of the approximation generated). It is the intersection of the two regular approximations corresponding to the two different computation rules. The structure of the theorem can be clearly seen.

Step 4: Empty approximations for all the different *member* procedures were generated. This indicates that no ancestor resolutions are needed when trying to prove $answer(S)$. Furthermore, empty approximations were also generated for the following *prove* procedures, $hand\_at$, $hold$, $\neg at$, $\neg hand\_at$, $\neg answer$, $\neg put$, $\neg grab$ and $\neg red$.

The deletion of all ancestor resolutions is a surprising result as it is not at all obvious from

$solve\_ans(X1, X2) \leftarrow t2024(X1), t13(X2)$
$t2024(X1) \leftarrow t2025(X1)$
$t13([\,]) \leftarrow true$
$t2025(answer(X1)) \leftarrow t2014(X1)$
$t2014(go(X1, X2)) \leftarrow t1933(X1), t2015(X2)$
$t1933(there) \leftarrow true$
$t1933(here) \leftarrow true$
$t2015(pick\_up(X1)) \leftarrow t2014(X1)$
$t2015(let\_go(X1)) \leftarrow t1938(X1)$
$t1938(go(X1, X2)) \leftarrow any(X1), t1937(X2)$
$t1938(pick\_up(X1)) \leftarrow t1938(X1)$
$t1938(now) \leftarrow true$
$t1938(let\_go(X1)) \leftarrow t1938(X1)$
$t1937(let\_go(X1)) \leftarrow t1938(X1)$

Program 6.1: DBA BHP—Regular Approximation

Theory 6.1 that ancestor resolution is never needed to prove $answer(S)$. Clauses 2,3,12 and 13 in Theory 6.1 also need not be considered in the proof of this theorem as all *prove* procedures in P_1 corresponding to literals occurring in these clauses have been proved useless (they have empty approximations in A). These four clauses are redundant for the proof of $answer(S)$.

We therefore have the information in Table 6.2 available that can be fed back into the "real" theorem prover (see Section 5.2.2). This information holds only for model elimination with

Formula:	$answer(Situation)$
Input Resolution:	Only at, $answer$, put, $grab$, red and $\neg hold$ can contribute to a proof.
Ancestor Resolution:	Redundant.
Object Theory Clauses:	Clauses 2,3,12 and 13 are redundant.

Table 6.2: DBA BHP—Analysis Information

top-level query $\leftarrow solve(answer(Situation), [\,])$. For any other literal used as a starting point, the analysis has to be redone with the appropriate top-level query.

Deletion of the corresponding useless clauses in the object theory results in specialised theorem provers with a reduced search space which allows a faster proof. The times in

Table 6.3 give an indication of the speedups possible with current state-of-the-art theorem provers. All timings are overall times in seconds measured on a Sun Sparc-10 with the "time" command [102]. These figures show that the detection of redundant clauses is potentially

Speedups: DBA BHP			
Prover	Original Problem	Reduced Problem	Speedup
Killer	4.6	0.5	9.2
Otter	0.7	0.5	1.4
Protein	1.1	0.9	1.2
Setheo	0.3	0.2	1.5
Theme	2.3	1.4	1.6

Table 6.3: Possible Speedups

a very useful analysis. No significant difference was found for this example when only eliminating ancestor resolution.

The transformation and analysis of this example has been shown in great detail. The rest of the examples will be done in less detail.

MSC008-1.002—LatSq—The Inconstructability of a Graeco-Latin Square

The first fifteen clauses in Theory 6.2 are axioms of the theory and the last two clauses the negation of the formula (theorem) to be proved. There are three predicate symbols.

Analysis gives the information in Table 6.4. It is interesting to note that the reflexivity axiom is redundant. This is an unexpected result and also indicates the existence of a non-minimal clause set for this particular theorem. The removal of this axiom only gives a small decrease in the runtime of the theorem provers listed in Table 6.3 (approximately five present, from 22.8 to 21.8 seconds using a model elimination theorem prover [5]). The speedup is not dramatic, but again shows that the detection and removal of redundant clauses is significant and is not included in current theorem provers.

1. $\neg eq(p1, p2)$
2. $eq(A, A)$
3. $\neg eq(A, B) \vee eq(B, A)$
4. $\neg latin(A, B, C) \vee \neg latin(A, B, D) \vee eq(D, C)$
5. $\neg latin(A, B, C) \vee \neg latin(A, D, C) \vee eq(D, B)$
6. $\neg latin(A, B, C) \vee \neg latin(D, B, C) \vee eq(D, A)$
7. $\neg greek(A, B, C) \vee \neg greek(A, B, D) \vee eq(D, C)$
8. $\neg greek(A, B, C) \vee \neg greek(A, D, C) \vee eq(D, B)$
9. $\neg greek(A, B, C) \vee \neg greek(D, B, C) \vee eq(D, A)$
10. $latin(E, F, p1) \vee latin(E, F, p2)$
11. $latin(G, p1, H) \vee latin(G, p2, H)$
12. $latin(p1, I, J) \vee latin(p2, I, J)$
13. $greek(K, L, p1) \vee greek(K, L, p2)$
14. $greek(M, p1, N) \vee greek(M, p2, N)$
15. $greek(p1, O, P) \vee greek(p2, O, P)$
16. $\neg greek(Q, R, S) \vee \neg latin(Q, R, T) \vee \neg greek(U, V, S) \vee \neg latin(U, V, T) \vee eq(V, R)$
17. $\neg greek(Q, R, S) \vee \neg latin(Q, R, T) \vee \neg greek(U, V, S) \vee \neg latin(U, V, T) \vee eq(U, Q)$

Theory 6.2: MSC008-1.002—LatSq—The Inconstructability of a Greaco-Latin Square

Formula:	$greek(Q, R, S) \wedge latin(Q, R, T) \wedge greek(U, V, S) \wedge$ $latin(U, V, T) \wedge \neg eq(U, Q)$
Input Resolution:	$eq(A, A)$ can never contribute to a proof.
Ancestor Resolution:	Redundant for $eq(X, Y)$ and $\neg eq(X, Y)$.
Object Theory Clauses:	Clause 2 (reflexivity axiom) is redundant.

Table 6.4: LatSq—Analysis Information

ANA003-4—SumContFuncLem1—Lemma 1 for the Sum of Continuous Functions is Continuous

The first five clauses in Theory 6.3 are the axioms of the theory, the next six clauses hypotheses and the last clause the negation of the formula (theorem) to be proved. There is only one predicate symbol $less\_or\_equal$.

Analysis gives the information in Table 6.5. Although the analysis information indicates that ancestor resolution is redundant, because of the depth of search needed to find a proof, it is very difficult to exploit this information in a successful proof. Ancestor resolution

1. $\neg less\_or\_equal(Z, minimum(X,Y)) \vee less\_or\_equal(Z,X)$
2. $\neg less\_or\_equal(Z, minimum(X,Y)) \vee less\_or\_equal(Z,Y)$
3. $less\_or\_equal(X,0) \vee less\_or\_equal(Y,0) \vee \neg less\_or\_equal(minimum(X,Y),0)$
4. $\neg less\_or\_equal(X, half(Z)) \vee \neg less\_or\_equal(Y, half(Z)) \vee$
   $less\_or\_equal(add(X,Y),Z)$
5. $less\_or\_equal(X,0) \vee \neg less\_or\_equal(half(X),0)$
6. $less\_or\_equal(Epsilon,0) \vee \neg less\_or\_equal(delta\_1(Epsilon),0)$
7. $less\_or\_equal(Epsilon,0) \vee \neg less\_or\_equal(delta\_2(Epsilon),0)$
8. $less\_or\_equal(Epsilon,0) \vee$
   $\neg less\_or\_equal(absolute(add(Z, negate(a\_real\_number))), delta\_1(Epsilon)) \vee$
   $less\_or\_equal(absolute(add(f(Z), negate(f(a\_real\_number)))), Epsilon)$
9. $less\_or\_equal(Epsilon,0) \vee$
   $\neg less\_or\_equal(absolute(add(Z, negate(a\_real\_number))), delta\_2(Epsilon)) \vee$
   $less\_or\_equal(absolute(add(g(Z), negate(g(a\_real\_number)))), Epsilon)$
10. $\neg less\_or\_equal(epsilon\_0, 0)$
11. $less\_or\_equal(Delta,0) \vee$
    $less\_or\_equal(absolute(add(xs(Delta), negate(a\_real\_number))), Delta)$
12. $less\_or\_equal(Delta,0) \vee$
    $\neg less\_or\_equal(add(absolute(add(f(xs(Delta)), negate(f(a\_real\_number)))),$
    $absolute(add(g(xs(Delta)), negate(g(a\_real\_number))))), epsilon\_0)$

Theory 6.3: ANA003-4—SumContFuncLem1—Lemma 1 for the Sum of Continuous Functions is Continuous

Formula:	$\neg less\_or\_equal(Delta,0) \wedge$ $less\_or\_equal(add(absolute(add(f(xs(Delta)),$ $negate(f(a\_real\_number)))),$ $absolute(add(g(xs(Delta)), negate(g(a\_real\_number))))),$ $epsilon\_0)$
Input Resolution:	No information.
Ancestor Resolution:	Redundant.
Object Theory Clauses:	No information.

Table 6.5: SumContFuncLem1—Analysis Information

can also be efficiently implemented and its removal may speed up a proof, but usually the improvement is small. Other optimisation techniques may play an important role in finding a proof quickly.

For the examples that follow, we only give the analysis information. The object theory can be found in the TPTP Problem Library.

TOP001-2—BasisTplgLem1—Topology Generated by a Basis Forms a Top

The first eleven clauses are the axioms of the theory and the last two clauses are the negation of the formula (theorem) to be proved. There are five predicate symbols. Analysis gives the information in Table 6.6.

Formula:	$subset\_sets(union\_of\_members(top\_of\_basis(f)), cx)$
Input Resolution:	$\neg basis(X,Y)$ and $\neg equal\_sets(X,Y)$ can never contribute to a proof.
Ancestor Resolution:	Redundant for $\neg basis(X,Y)$, $basis(X,Y)$, $\neg equal\_sets(X,Y)$ and $equal\_sets(X,Y)$.
Object Theory Clauses:	No information.

Table 6.6: BasisTplgLem1—Analysis Information

The indication that input resolution is not needed for certain literals may allow the overhead caused by the need for contrapositives from the object theory to be minimised. Contrapositives of clauses whose heads correspond to literals that cannot succeed with input resolution may be deleted. Exploiting this information in current theorem provers may not be possible, as in general no mechanisms (e.g. flags) exist that could be used to selectively switch input resolution on and off for the indicated literals. We hope that the availability of such detailed information will change the current state of affairs and that it will be possible to use this information constructively in the near future.

PUZ033-1—Winds—The Winds and the Windows Puzzle

The first eleven clauses are the axioms of the theory, the twelfth clause a hypothesis and the last clause the negation of the formula (theorem) to be proved. There are twelve predicate symbols. Analysis gives the information in Table 6.7

Formula:	$window\_is\_shut$
Input Resolution:	Not necessary for $\neg headache$, $\neg i\_feel\_rheumatic$, $\neg neighbor\_practices\_flute$, $\neg sunshine$, $\neg wind\_in\_east$ and $headache$.
Ancestor Resolution:	Only necessary for $neighbor\_practices\_flute$, $cold$, $\neg door\_is\_open$ and $\neg window\_is\_shut$.
Object Theory Clauses:	No information.

Table 6.7: Winds—Analysis Information

Note that 20 out of 24 possible ancestor checks were eliminated. Also, because we have a propositional object theory, the approximation is very precise as variables in the object theory are absent (which may have caused a loss of information during the approximation phase).

PUZ031-1—SteamR—Schubert's Steamroller

The first twenty five clauses are the axioms of the theory and the last clause the negation of the formula (theorem) to be proved. There are ten predicate symbols. Analysis gives the information in Table 6.8.

Formula:	$animal(Animal) \wedge animal(Grain\_eater) \wedge grain(Grain) \wedge eats(Animal, Grain\_eater) \wedge eats(Grain\_eater, Grain)$
Input Resolution:	Redundant for all negative literals except $\neg eats(X, Y)$.
Ancestor Resolution:	Redundant for all negative and all positive literals except $\neg eats(X, Y)$ and $eats(X, Y)$.
Object Theory Clauses:	No information.

Table 6.8: SteamR—Analysis Information

NUM014-1—VallsPrm—Prime Number Theorem

The first eleven clauses are the axioms of the theory, the next two hypothesis and the last clause the negation of the formula (theorem) to be proved. There are three predicate symbols. Analysis gives the information in Table 6.9.

Formula:	$divides(a, b)$
Input Resolution:	Not necessary for $\neg prime(X)$ and $\neg product(X, Y, Z)$.
Ancestor Resolution:	Only necessary for $\neg divides(X, Y)$.
Object Theory Clauses:	No information.

Table 6.9: VallsPrm—Analysis Information

COM002-2—8StSp—A Program Correctness Theorem

The first four clauses are the axioms of the theory, the next fourteen hypothesis and the last clause the negation of the formula (theorem) to be proved. There are four predicate symbols. Analysis gives the information in Table 6.10

Formula:	$\neg fails(p3, p3)$
Input Resolution:	Redundant for $fails(X, Y)$, $\neg labels(X, Y)$, $\neg has(X, Y)$ and $\neg follows(X, Y)$.
Ancestor Resolution:	Redundant.
Object Theory Clauses:	No information.

Table 6.10: 8StSp—Analysis Information

CAT007-3—DomEqCod—If domain(x)=codomain(y) then xy is defined

The first eight clauses are the axioms of the theory, the next three hypotheses and the last clause the negation of the formula (theorem) to be proved. There are two predicate symbols. Analysis gives the information in Table 6.11.

Formula:	$there\_exists(compose(c2, c1))$
Input Resolution:	No information.
Ancestor Resolution:	Redundant for $there\_exist(X)$.
Object Theory Clauses:	No information.

Table 6.11: DomEqCod—Analysis Information

List Membership

We conclude this section with Theory 6.4. Informally, this theory describes the following interesting problem: prove that an element p can never occur in any list with elements q. We would like to prove that the formula $test(p,q)$ *IS NOT* a theorem of the theory. This is different from what is usually asked of a theorem prover, namely to prove that a formula *IS* a theorem.

1. $test(E1,E2) \vee \neg gen(E2,L) \vee \neg member(E1,L) \vee \neg pred(E1) \vee \neg pred(E2)$
2. $gen(X,cons(X,nil)) \vee \neg pred(X) \vee \neg pred\_list(cons(X,nil))$
3. $gen(X,cons(X,Xs)) \vee \neg gen(X,Xs) \vee \neg pred(X) \vee \neg pred\_list(cons(X,Xs))$
4. $member(X,cons(X,Xs)) \vee \neg pred(X) \vee \neg pred\_list(cons(X,Xs))$
5. $member(X,cons(Y,Xs)) \vee \neg member(X,Xs) \vee \neg pred(X) \vee \neg pred\_list(cons(Y,Xs))$
6. $pred\_list(nil)$
7. $pred\_list(cons(X,Xs)) \vee \neg pred(X) \vee \neg pred\_list(Xs)$
8. $pred(p)$
9. $pred(q)$
10. $\neg test(p,q)$

Theory 6.4: A Challenging Problem for Theorem Proving—List Membership

The first nine clauses are the axioms of the theory and the last clause the negation of the formula to be proved. The analysis information that should be inferred is given in Table 6.12.

Formula:	$test(p,q)$
Input Resolution:	Redundant.
Ancestor Resolution:	Redundant.
Object Theory Clauses:	Redundant.

Table 6.12: List Membership—Analysis Information

This example shows how an infinitely failed deduction could be turned into a finitely failed deduction. We know now that $test(p,q)$ is not a theorem of the given theory. Most current theorem provers are unable to finitely fail this deduction.

6.1.2 Summary of Results and Complexity Considerations

In the previous section we have shown the following:

- ancestor resolution is redundant for some problems;
- ancestor resolution is redundant for particular literals in some problems;
- input resolution is redundant for some problems;
- input resolution is redundant for particular literals in some problems;
- infinitely failed deductions can be turned into finitely failed deductions;
- some contrapositives are redundant.
- some clauses in our first-order theories are redundant;

From the experiments performed so far, it seems that the detection of useless clauses in first-order theories is the most promising analysis. Although we have not inferred any useful information for a problem from the set theory domain, this domain seems particularly suitable for the detection of useless set theory axioms. However, to achieve this satisfactorily, we might have to do a more precise analysis. This is the subject of current research.

As indicated at the beginning of this chapter, we do not include the partial evaluation times in the overall analysis and transformation times as partial evaluation is done only once for every theory (given some proof procedure): we partially evaluate with respect to all theorems. The restriction to a particular formula or class of formulas is done in the approximation phase. The same partial evaluation can therefore be used in all the possible approximations of formulas and the partial evaluation time should therefore be divided between all the formulas we want to prove and becomes insignificant. It is therefore better to ignore it.

Experiments performed with regular approximations reported by Gallagher and de Waal for the GdW system [42] indicated a worst-case complexity that is exponential on the average number of function symbols that can appear in each argument position of the program we approximate. The number of predicate symbols and function symbols in the object theory therefore has a direct influence on the performance of the regular approximation algorithm. However, in practice average cases are quite tractable and we have found that the performance of the algorithm is affected roughly linearly by the number of strongly connected components in the predicate call-graph, the average length of clauses and the average size of the strongly connected components. The recursive structure of programs

has also been found to have an effect on the number and size of the strongly connected components.

The speedups given in Table 6.3 do not justify the 66.1 seconds spent on the analysis of the proof procedure, theory and formula. However, as indicated in Chapter 4, there exists at least one other implementation of regular approximations that will give an order of magnitude decrease in approximation times. A further improvement of an order of magnitude may make the analysis worth considering as a preprocessing phase in theorem provers. However, for failing non-theorems the analysis time is not that important. If a previously infinitely failed deduction can be turned into a finitely failed deduction, the analysis time is obviously not important. The list membership example is such an example. Also, for the proof of properties about theorems, theories and formulas, the analysis time is not critical.

6.2 Optimising Naive Near-Horn Prolog

In this section we show how an infinite failed deduction can be turned into finite failure. Consider the object theory given in Theory 6.5 from Loveland [69] in conjunction with the naive nH-Prolog proof procedure given in Program 2.4. This object theory was used to demonstrate the incompleteness of the naive nH-Prolog proof system. The deduction given

$$
\begin{array}{l}
q \leftarrow a, b \\
a \leftarrow c \\
a \leftarrow d \\
c \vee d \\
b \leftarrow e \\
b \leftarrow f \\
e \vee f
\end{array}
$$

Theory 6.5: Incompleteness of the Naive Near-Horn Prolog Proof System

in Figure 6.1 showed that naive nH-Prolog would never terminate when trying to prove the formula q.

In general it is not so easy to establish nontermination of a proof system, because it is not known how many deductions are needed to detect a loop. Hundreds of deductions may be necessary to identify such a property. Even worse, a proof system may not terminate and there may be no recurring pattern of deductions. In this case, it is impossible to detect

$$
\begin{array}{lll}
(0) & ?-q. & \\
(1) & :-a,b & \\
(2) & :-c,b & \\
(3) & :-b\ \#[d] & \\
(4) & :-e\ \#[d] & \\
(5) & :-\ \#[f,d] & \\
(6) & :-q\ \#f[d] & \{restart\} \\
(7) & :-a,b\ \#f[d] & \\
 & \vdots & \\
(17) & :-\ \#d\ [f,d] & \\
(18) & :-q\ \#f[d] & \{restart\} \\
 & \vdots &
\end{array}
$$

Figure 6.1: Non-Terminating Deduction

nontermination with a scheme as presented in the example. It is however possible to detect failure of this proof system to prove the given query by applying the method described in Section 5.1.

Note that in this case, we can apply GENERAL, KERNEL or MARSHAL with equal success as we have a propositional object theory. We get the following analysis in Table 6.13. An empty approximation for *solve* resulted which indicates that the above proof

Formula:	q
Input Resolution	Redundant.
Ancestor Resolution:	Redundant.
Object Theory Clauses:	Redundant.

Table 6.13: Incompleteness Turned into Finite Failure—Analysis Information

system is unable to prove the above theorem starting from query q. We still do not know if it is because there does not exist a proof or because of the incompleteness of the proof system. However, we have removed some indecision with respect to the above proof system and can now move on to another proof system (that may be complete) and try to prove the above query.

It is worth emphasising that this method may indicate failure for more formulas (non-theorems) than is possible with the original proof system. This is because the analysis

method may approximate an infinitely failed tree by a finitely failed tree (the finite failure set of the original prover may not be preserved) [25]. This increase in power of the prover provided by the analysis method may be very important.

There exist more powerful theorem provers that are able to prove failure of the above query given this object theory. In general, the existence of more powerful theorem provers that are able to terminate all non-terminating deductions cannot be assumed. The only way to identify failure for some non-terminating deductions may be to use methods such as those described in this thesis.

6.3 Specialising "Compiled Graphs"

An alternative approach to "ordinary" semantic tableaux (e.g. as described by Fitting [33]) presented by Possega [86] generates a compiled graph version of an expanded tableau and only leaves the proof search at runtime. This step can be compared to a "partial evaluation" step or the application of the minimal partial evaluator presented earlier in this section. This compilation method differs from partial evaluation in that renaming and deterministic unfolding are not done. Partial evaluation can therefore still improve on these programs. Our aim in this section is to specialise these programs with the specialisation methods presented in Chapter 5.

To illustrate the specialisation of this method, consider the following theory from Possega [86]

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

A graph representing a fully expanded tableau is now computed and then compiled into a logic program that is equivalent to testing if the original tableaux can be closed. If this is the case then we have an inconsistent formula as no model can be constructed. The compiled graph of the above formula is given in Program 6.2. For this program, partial evaluation is powerful enough to detect failure. We have thus proved that the given formula $(a \leftrightarrow b) \wedge a \wedge \neg b$ is inconsistent and do not need to run the above program. GENERAL is therefore powerful enough to detect failure.

The propositional case is fairly straightforward to specialise. We now present a first-order example also from Possega [86] in Theory 6.6 (also known as Pelletier 30—SYN060-1 in the TPTP Problem Library) which is more of a challenge to our specialisation techniques. The graph compiled version of Theory 6.6 is given in Program 6.3. After applying GENERAL to the program in Program 6.3 we get the specialised program with useless clauses deleted in Program 6.4.

$satisfy(Path) \leftarrow functor(Path, path, 2), node(5, Path)$

$node(5, Path) \leftarrow (node(3, Path); node(1, Path))$
$node(3, Path) \leftarrow arg(1, Path, -), node(4, Path)$
$node(4, Path) \leftarrow arg(2, Path, -), node(6, Path)$
$node(6, Path) \leftarrow arg(1, Path, +), node(7, Path)$
$node(7, Path) \leftarrow arg(2, Path, -)$
$node(1, Path) \leftarrow arg(1, Path, +), node(2, Path)$
$node(2, Path) \leftarrow arg(2, Path, +), node(6, Path)$

Program 6.2: Compiled Graph

$\forall x(f(x) \vee g(x) \rightarrow \neg h(x))$
$\forall x((g(x) \rightarrow \neg i(x)) \rightarrow (f(x) \wedge h(x)))$
$\neg \forall x \ i(x)$

Theory 6.6: First-Order Tableau Example

The deletion of useless clauses achieved the following:

- 8 out of 10 use_gamma_1 clauses were deleted.
- 8 out of 10 specialised *member* definitions were deleted.
- 7 out of 8 specialised *close* definitions were deleted.
- 2 out of 8 specialised *node* definitions were specialised away by deterministic unfolding.

For the first-order case, a call to the compiled graph program will succeed if the formula we are considering is inconsistent (this is different from the propositional case presented earlier—see Possega [86] for further detail). The goal $\leftarrow prove(2)$ succeeds for the original and specialised programs which indicates that the formula in Theory 6.6 is inconsistent. $i(x)$ therefore is a logical consequence of the theory. The specialised program given in Program 6.4 exhausts the search space in less than half the time of the original program (with $\leftarrow prove(2)$ as goal: $\leftarrow prove(1)$ fails).

For this example the result of our analysis is a specialised program and not analysis results that can directly be used by other tableau theorem provers. However, if the relationship between the compiled program and inference rules in a tableau theorem prover can be

```
prove (Limit) ← node(gamma1, Limit, [ ], _)

use_gamma(0, _, _) ← !, fail.
use_gamma(Limit, Path, VarBnd) ← NewLimit is Limit − 1,
        member(gamma(N), Path),
        node(N, NewLimit, Path, VarBnd)

close(++ L1, [H|Path]) ←
        H = −− L2, unify(L1, L2)); close(++ L1, Path)
close(−− L1, [H|Path]) ←
        H = ++ L2, unify(L1, L2)); close(−− L1, Path)

node(gamma1, Limit, Path, VarBnd) ←
        node(gamma2, Limit, [gamma(4)|Path], VarBnd)
node(gamma2, Limit, Path, VarBnd) ←
        node(10, Limit, [gamma(9)|Path], VarBnd)
node(1, Limit, Path, bind(A, B)) ←
        close(−− f(A), Path); node(2, Limit, [−− f(A)|Path], bindA, B)))
node(2, Limit, Path, bind(A, B)) ←
        close(−− f(A), Path); use_gamma(Limit, [−− g(A)|Path], bind(A, B)))
node(3, Limit, Path, bind(A, B)) ←
        close(−− f(A), Path); use_gamma(Limit, [−− h(A)|Path], bind(A, B)))
node(4, Limit, Path, bind(A, B)) ← node(1, Limit, [−− f(A)|Path], bindA, B))),
        node(3, Limit, [−− f(A)|Path], bindA, B)))
node(5, Limit, Path, bind(A, B)) ←
        close(++ g(B), Path); node(6, Limit, [++ g(B)|Path], bindA, B)))
node(6, Limit, Path, bind(A, B)) ←
        close(++ i(B), Path); use_gamma(Limit, [++ i(B)|Path], bind(A, B)))
node(7, Limit, Path, bind(A, B)) ←
        close(++ f(B), Path); node(8, Limit, [++ f(B)|Path], bindA, B)))
node(8, Limit, Path, bind(A, B)) ←
        close(++ h(B), Path); use_gamma(Limit, [++ h(B)|Path], bind(A, B)))
node(9, Limit, Path, bind(A, B)) ← node(5, Limit, [−− f(A)|Path], bindA, B))),
        node(7, Limit, [−− f(A)|Path], bindA, B)))
node(10, Limit, Path, bind(A, B)) ←
        close(−− f(A), Path); use_gamma(Limit, [−− i(sk0)|Path], VarBnd))
```

Program 6.3: Compiled First-Order Graph

$prove\ (X1) \leftarrow node10\_1(X1, gamma(9), gamma(4), [\,], X2)$

$use\_gamma\_1(0, \_, \_, \_, \_, \_) \leftarrow !, fail$
$use\_gamma\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow X1 > 0, X8\ \ is\ \ X1 - 1,$
$\quad member4\_1([X2, X3, X4|X5]),$
$\quad node1\_1(X8, X2, X3, X4, X5, bind(X9, X7)),$
$\quad node3\_1(X8, X2, X3, X4, X5, bind(X9, X7))$
$use\_gamma\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow X1 > 0, X8\ \ is\ \ X1 - 1,$
$\quad member9\_1([X2, X3, X4|X5]),$
$\quad node5\_1(X8, X2, X3, X4, X5, bind(X6, X9)),$
$\quad node7\_1(X8, X2, X3, X4, X5, bind(X6, X9))$

$node1\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close1\_1(X6, [X2, X3, X4|X5])$
$node1\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad node2\_1(X1, -- (f(X6)), X2, X3, [X4|X5], bind(X6, X7))$
$node2\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close2\_1(X6, [X2, X3, X4|X5])$
$node2\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad use\_gamma\_1(X1, -- (g(X6)), X2, X3, [X4|X5], bind(X6, X7))$
$node3\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close3\_1(X6, [X2, X3, X4|X5])$
$node3\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad use\_gamma\_1(X1, -- (h(X6)), X2, X3, [X4|X5], bind(X6, X7))$
$node5\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close6\_1(X7, [X2, X3, X4|X5])$
$node5\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad node6\_1(X1, ++ (g(X7)), X2, X3, [X4|X5], bind(X6, X7))$
$node6\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close8\_1(X7, [X2, X3, X4|X5])$
$node6\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad use\_gamma\_1(X1, ++ (i(X7)), X2, X3, [X4|X5], bind(X6, X7))$
$node7\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close5\_1(X7, [X2, X3, X4|X5])$
$node7\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad node8\_1(X1, ++ (f(X7)), X2, X3, [X4|X5], bind(X6, X7))$
$node8\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow close7\_1(X7, [X2, X3, X4|X5])$
$node8\_1(X1, X2, X3, X4, X5, bind(X6, X7)) \leftarrow$
$\quad use\_gamma\_1(X1, ++ (h(X7)), X2, X3, [X4|X5], bind(X6, X7))$
$node10\_1(X1, X2, X3, X4, X5) \leftarrow use\_gamma\_1(X1, -- (i(sko)), X2, X3, X4, X5)$

$member4\_1([gamma(4)|X1]) \leftarrow$
$member4\_1([X1|X2]) \leftarrow member4\_1(X2)$

$member9\_1([gamma(9)|X1]) \leftarrow$
$member9\_1([X1|X2]) \leftarrow member9\_1(X2)$

$close4\_1([++ (i(sko))|X1]) \leftarrow$
$close4\_1([X1|X2]) \leftarrow close4\_1(X2)$

Program 6.4: Compiled First Order Graph (Improved)

established, it should be possible to use this compilation method using graphs to infer information in a similar way as was done for model elimination in the previous sections.

6.4 Meta-Programming Results for Model Elimination

In this section we show how to transform a chosen proof procedure and object theory into a more efficient runnable program using MARSHAL developed in Chapter 5. We specialise the model elimination proof procedure written in a ground representation with respect to one of the problems from the previous sections. We also compare the results with that computed in previous sections.

In Section 2.2.1 we argued that specialising the non-ground and ground representations using the methods developed in this thesis gives similar specialisation results. From the examples given in the previous section it can easily be seen how to specialise the non-ground representation. Instead of representing the analysis information in a table, we just delete the corresponding clauses from the program generated by partial evaluation.

For example, the program in Appendix A can be specialised by deleting all useless clauses as specified by the analysis information. This gives our final specialised program (in this case we should have added extra arguments for the search procedure as presented in the prover in Program 2.2 to get a runnable program).

To demonstrate the application of MARSHAL, we specialise the model elimination theorem prover using a ground representation, Program 2.3, with the SP-system modified with an unfolding rule based on the ideas of Chapter 3.

Consider the proof procedure in Program 2.3 and the Prime Number Problem (NUM014-1) from the previous section. The prover is now partially evaluated with respect to the object theory and goal $\leftarrow solve(\_, \_, P)$, where P is the object theory represented using a ground representation similar to the one used by Defoort [30, 24]. Part of the partially evaluated program is given in Program 6.5. We partially unfolded $unify(H, G, S, S1)$, but left unification of object level arguments for later (one of our assumptions in the Chapter 1 was that we have a procedure available, possibly a partial evaluator such as SAGE [46], that can specialise unification for us efficiently).

We have the following comments.

- There is a direct correspondence between this program and the program generated when using a non-ground representation.

$solve(struct(p,[X1]),X2,X3) \leftarrow$
$\quad depth\_bound(X4),$
$\quad prove\_1(X1,X2,X4,[\,],X5,1000,X6,X3)$
$solve(struct(m,[X1,X2,X3]),X4,X5) \leftarrow$
$\quad depth\_bound(X6),$
$\quad prove\_2(X1,X2,X3,X4,X6,[\,],X7,1000,X8,X5)$
$solve(struct(d,[X1,X2]),X3,X4) \leftarrow$
$\quad depth\_bound(X5),$
$\quad prove\_3(X1,X2,X3,X5,[\,],X6,1000,X7,X4)$
$solve(struct(\neg p,[X1]),X2,X3) \leftarrow$
$\quad depth\_bound(X4),$
$\quad prove\_4(X1,X2,X4,X5,X6,X3)$
$solve(struct(\neg m,[X1,X2,X3]),X4,X5) \leftarrow$
$\quad depth\_bound(X6),$
$\quad prove\_5(X1,X2,X3,X4,X6,X7,X8,X5)$
$solve(struct(\neg d,[X1,X2]),X3,X4) \leftarrow$
$\quad depth\_bound(X5),$
$\quad prove\_6(X1,X2,X3,X5,X6,X7,X4)$

$prove\_1(X1,X2,X3,X4,X5,X6,X6,X7) \leftarrow$
$\quad member\_1(X1,X2,X4,X5)$
$prove\_1(X1,X2,X3,X4,X5,X6,X6,X7) \leftarrow$
$\quad X3 > 1,$
$\quad X8 \;\; is \;\; X3 - 1,$
$\quad unify1(struct(a,[\,]),X1,X4,X5)$
$\vdots$

$member\_1(X1,[struct(p,[X2])|X3],X4,X5) \leftarrow$
$\quad unify1(X1,X2,X4,X5)$
$member\_1(X1,[X2|X3],X4,X5) \leftarrow$
$\quad member\_1(X1,X3,X4,X5)$
$\vdots$

Program 6.5: Specialised Theorem Prover using a Ground Representation

- Because this program was generated by a sound partial evaluator, the correctness result of partial evaluation holds for this program.
- The choice whether unification should be specialised before or after the approximation phase is the user's. However, this may influence the analysis results as we will show.
- If the specialisation of unification is left until after the approximation, we may choose to ignore approximation of unification altogether or we may approximate the unification with our regular approximator.
- This program may be approximated as usual. Because a ground representation was used, the depth of terms is now greater than for the non-ground representation and the regular approximation might be less precise.

For Program 6.5 (augmented with the other renamed definitions), approximation gives the information in Table 6.14. We chose to ignore approximation of the unification algorithm.

Formula:	$divides(a, b)$
Input Resolution:	Not necessary for $\neg prime(X)$ and $\neg product(X, Y, Z)$.
Ancestor Resolution:	Not necessary for $\neg prime(X)$ and $\neg product(X, Y, Z)$.
Object Theory Clauses:	No information.

Table 6.14: VallsPrm—Analysis Information Using a Ground Representation

These results are less precise than that reported previously for the same problem (especially the ancestor resolution results). The ground unification algorithm was only partially specialised. We now have the choice of further specialising it either before or after we do the approximation phase. If it is done after the approximation phase and we choose to ignore the approximation of unification, we cannot expect to improve much on the analysis results given in Table 6.14. However, unification will be efficient and our resulting program should be at least as efficient as that achievable by partial evaluation alone (if we take away the approximation phase, we get just a partial evaluation phase).

If partial evaluation of the ground representation is done before approximation, we can expect similar results as those presented for the non-ground representation (depending on the sophistication of the specialisation of unification). The propagation of object theory constants and function symbols will be the main improvement of the specialisation of unification. We suspect that this improvement will bring us close to the analysis results presented in the first section of this chapter. However, as indicated, we might have to make do with a less specialised program because of the complexities associated with the ground representation.

The deletion of useless clauses, even with the limited information inferred when using a ground representation, still gives a more efficient program than is achieved by just partial evaluation alone. The results may, however, not be as good as we would like when compared to the results reported before.

In [27] de Waal and Gallagher reported specialisations resulting in speeds-ups of between 2.2 and 42.9 for some of the problems in Table 6.1. These results were for the specialisation of a model elimination theorem prover written in a non-ground representation, Program 2.2. Most of these improvements were not due to partial evaluation, but were a result of the deletion of useless clauses. Although the prover is very naive compared to some state of the art theorem provers, it gives some indication of the order of speedups that should be achievable for meta-programs. This may have to be revised downwards for meta-programs using a ground representation as explained in this section.

For the naive nH-Prolog proof procedure presented in this Chapter, the results for the chosen object theory are independent of the representation chosen (it is a propositional theory). Infinite failure is turned into finite failure. This is the best any method can achieve.

Chapter 7

Combining Safe Over-Approximations with Under-Approximations

The motivation for this chapter is the following: given a proof procedure, object theory and formula, we would like to know whether the formula can be proved using a simpler, but incomplete proof procedure. We accomplish this by making an educated guess followed by a test of our hypothesis using safe over-approximations and under-approximations.

The use of approximations will assist in identifying cases where the use of an incomplete proof procedure will not suffice to prove the given formula. This can eliminate useless proof attempts and indicate when the full power of the complete proof procedure is needed.

In the previous chapters we reasoned with safe approximations to eliminate useless inference rules and object clauses that are not needed to prove a given theorem. In contrast, in this chapter we use both over- and under-approximation of a proof procedure to identify necessary (actually, not unnecessary) inference rules for proving a given theorem in some object theory. The diagram in Figure 7.1 illustrates the idea: C is the set of all theorems in a given object theory (that is, theorems provable by some sound and complete proof procedure). R is the set of theorems provable in the same object theory, but with a reduced (incomplete) proof procedure. Our aim is to decide if a given formula T is in R. This is usually undecidable. We now compute a safe over-approximation A of R, given some formula T, in which membership is decidable. If T is not in the approximation A, T is definitely not in R and therefore not provable using the incomplete proof procedure. However if T is in A, then T might also be in R and we might be able to prove the theorem T using only

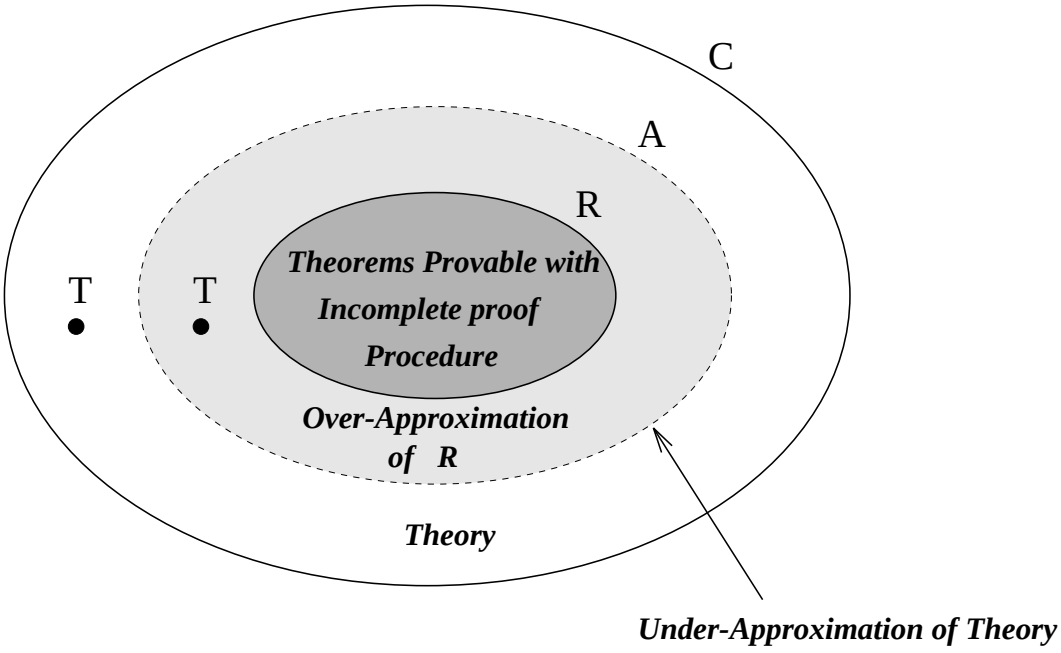


Figure 7.1: Under-Approximations of Proof Procedures

the incomplete proof procedure.

This method depends directly on the precision of the approximation A. If the approximation loses too much information, useless work may be done in unsuccessful proofs and nothing will be gained. However, if a precise approximation can be constructed, there will be fewer successful proofs for T when the approximation indicates that a proof may exist.

The aim therefore is to show how reasoning with both over- and under-approximations may assist in exploiting the speed and simplicity of incomplete proof procedures to find proofs quickly for difficult theorems.

An interesting application is the class of CASE-Free theories defined by Wakayama [107]. The class of CASE-Free theories is the class of first-order clausal theories where all theorems are provable using only input resolution. An input resolution proof procedure corresponds to model elimination [67] without ancestor resolution. It is thus of interest to know whether a given theorem is provable by an input resolution prover. We are considering a larger class of theories than the CASE-Free ones, since we consider a specific theorem (rather than all theorems for a given theory).

Although in this chapter we focus only on checking this particular restriction of a proof

procedure, this method is general and does not rely on any proof procedure or set of inference rules. Regular approximations as described in Chapter 4 will provide the vehicle to achieve this aim. We will illustrate this process on two combinatorially difficult first-order theories, namely Schubert's Steamroller Problem (PUZ031-1 in the TPTP Problem Library) and another Blind Hand Problem (APA BHP) (MSC001-1 in the TPTP Problem Library).

7.1 Incomplete Proof Procedures

Soundness and completeness are usually prerequisites when considering the usefulness of a proof procedure. Relaxing the soundness requirement does not make much sense. However, there is scope for relaxing the completeness requirement.

If some relation between a widely occurring subclass of problems and an incomplete version of a complete proof procedure can be established, and we have some means of identifying instances of the subclass, the corresponding performance increase obtainable by only considering the incomplete version of the proof procedure for instances of this subclass may justify the analysis time spent. In other words, the analysis time needed to determine whether the theory is in a subclass or not, should not exceed the time saved in achieving a faster proof.

An incomplete proof procedure P' will be obtained from a complete proof procedure P by deleting some clauses representing inference rules from P. In this chapter we consider only clause deletion for derivations satisfying Lemma 7.1.1. This will ensure that if a theorem is provable using P', it will also be provable using P. If this restriction is not adhered to, we may prove fictitious theorems in P' that are not theorems in P (because negation as failure steps may now succeed that should fail).

LEMMA 7.1.1 *Let P be a normal program and let P' be the result of deleting one or more clauses from P. Suppose $P' \cup \{G\}$ has an SLDNF-refutation containing no negation-as-failure steps. Then $P \cup \{G\}$ has an SLDNF-refutation.*

This is correct since any derivation in P' not using negation-as-failure is also a derivation in P. In particular if P and P' are definite programs then all derivations in P' are also derivations in P.

The object theory is represented in contrapositive form [85]. That is, an n-literal clause is represented by n contrapositives. This representation allows for better specialisation of the object theory, as we may be able to detect that a certain contrapositive is useless with

Given a sound and complete proof procedure (implemented as a meta-program P), a representation of an object theory T and a representation of a formula F, perform the following four steps.

1. **Identify an incomplete proof procedure P_1 from P.**
2. **Partially evaluate P_1 with respect to T and F', where F is an instance of F'. Call the partially evaluated program P_2.**
3. **Compute a regular approximation of the clauses in P_2 with respect to F. This yields A.**
4. **If $A \cup \{\neg F\}$ fails, the incomplete proof procedure is insufficient for proving the given theorem. Otherwise, optimise by deleting useless clauses from P_2 giving P_3. The approximation A is used to detect useless clauses. P_3 is the optimised incomplete proof procedure and will be used to search faster for a possible proof to the original theorem.**

Figure 7.2: INSPECTOR

respect to the proof of some theorem. This may not have been possible if we worked with clauses from conjunctive normal form as all contrapositives derived from the clause need to be useless before we could delete this clause. A finer grained analysis is therefore possible which may lead to a better final specialisation.

7.2 A Four-Stage Specialisation Procedure—INSPECTOR

A four-stage specialisation procedure is given in Figure 7.2. The procedure first constructs an under-approximation P_1 of a proof procedure P and then an over-approximation of A of P_1 to test our hypotheses.

Firstly, it is used to identify the possible existence of a proof given a theorem and some object theory. Secondly, it is used to optimise the proof procedure P_2 by deleting useless clauses.

The effectiveness of the above procedure depends directly on the precision of the regular approximation. If the approximation is not precise enough, the above method may indicate

that proofs exist for many unsuccessful proofs. This may lead to a lot of wasted effort.

As we can now compute regular approximations automatically [42] the results of the above process may be that we can prove some difficult theorems much faster than was previously possible. In the next section we will apply the above process to two combinatorially difficult theories, namely Schubert's Steamroller Problem and the Blind Hand Problem(APA BHP).

7.3 Analysis Results

7.3.1 Schubert's Steamroller Problem

Consider the version of Schubert's Steamroller Problem given in Theory 7.1, which is the standard formulation proposed by Stickel [101] from the TPTP Problem Library. A regular approximation of a partially evaluated proof procedure consisting of only input resolution and the above object theory (with contrapositives) with respect to this theory and the theorem (clause 26) is computed (the goal to the partial evaluator was $\leftarrow solve(\_)$). A non-empty regular approximation results. The constructed regular approximation is given in Program 7.1 (we list only the approximation for the predicate $eats(X,Y)$). This is the most precise approximation possible (given the restrictions imposed by our regular approximations) and indicates that there may possibly exist a proof for the above theorem using only input resolution.

The similarity between the this program and the original unfolded theory is obvious and startling. The above program corresponds closely to a version of Schubert's Steamroller Problem, but sharing information between variables has been lost. For instance, in the theorem to be proved the variables *Grain_eater*, *Animal* and *Grain* occur in more than one literal. The approximation has lost this connection. Nevertheless, Program 7.1 is a precise approximation of the original theory and we can predict that if a proof exists in the approximation, we have a good chance of finding a proof using the incomplete proof procedure from which the approximation is derived.

As a possible proof exists, we continue and optimise the partially evaluated incomplete proof procedure using the rest of the information contained in the regular approximation. For space reasons, the whole approximation cannot be given, but it contains the following information.

- All contrapositives with a negative literal as the head of a clause can be deleted, except for those whose head is the literal $\neg eats(X,Y)$.

1. $animal(X) \vee \neg wolf(X)$
2. $animal(X) \vee \neg fox(X)$
3. $animal(X) \vee \neg bird(X)$
4. $animal(X) \vee \neg caterpillar(X)$
5. $animal(X) \vee \neg snail(X)$
6. $wolf(a\_wolf)$
7. $fox(a\_fox)$
8. $bird(a\_bird)$
9. $caterpillar(a\_caterpillar)$
10. $snail(a\_snail)$
11. $grain(a\_grain)$
12. $plant(X) \vee \neg grain(X)$
13. $eats(Animal, Plant) \vee eats(Animal, Small\_animal) \vee \neg animal(Animal) \vee$
    $\neg plant(Plant) \vee \neg animal(Small\_animal) \vee \neg plant(Other\_plant) \vee$
    $\neg much\_smaller(Small\_animal, Animal) \vee \neg eats(Small\_animal, Other\_plant)$
14. $much\_smaller(Catapillar, Bird) \vee \neg caterpillar(Catapillar) \vee \neg bird(Bird)$
15. $much\_smaller(Snail, Bird) \vee \neg snail(Snail) \vee \neg bird(Bird)$
16. $much\_smaller(Bird, Fox) \vee \neg bird(Bird) \vee \neg fox(Fox)$
17. $much\_smaller(Fox, Wolf) \vee \neg fox(Fox) \vee \neg wolf(Wolf)$
18. $\neg wolf(Wolf) \vee \neg fox(Fox) \vee \neg eats(Wolf, Fox)$
19. $\neg wolf(Wolf) \vee \neg grain(Grain) \vee \neg eats(Wolf, Grain)$
20. $eats(Bird, Catapillar) \vee \neg bird(Bird) \vee \neg caterpillar(Catapillar)$
21. $\neg bird(Bird) \vee \neg snail(Snail) \vee \neg eats(Bird, Snail)$
22. $plant(caterpillar\_food\_of(Catapillar)) \vee \neg caterpillar(Catapillar)$
23. $eats(Catapillar, caterpillar\_food\_of(Catapillar)) \vee \neg caterpillar(Catapillar)$
24. $plant(snail\_food\_of(Snail)) \vee \neg snail(Snail)$
25. $eats(Snail, snail\_food\_of(Snail)) \vee \neg snail(Snail)$
26. $\neg animal(Animal) \vee \neg animal(Grain\_eater) \vee \neg grain(Grain) \vee$
    $\neg eats(Animal, Grain\_eater) \vee \neg eats(Grain\_eater, Grain)$

Theory 7.1: PUZ031-1—SteamR—Schubert's Steamroller

- All contrapositives with a negative literal in the body of a clause can be deleted, except for those containing the literal $\neg eats(X, Y)$.

The above information can now be used to delete useless clauses from P_2 to give the spe-

$prove\_10\_ans(X1) \leftarrow t730(X1)$
$t730(++\ X1) \leftarrow t732(X1)$
$t732(eats(X1, X2)) \leftarrow t705(X1), t727(X2)$
$t705(a\_wolf) \leftarrow$
$t705(a\_fox) \leftarrow$
$t705(a\_bird) \leftarrow$
$t705(a\_snail) \leftarrow$
$t705(a\_caterpillar) \leftarrow$
$t727(snail\_food\_of(X1)) \leftarrow t53(X1)$
$t727(a\_grain) \leftarrow$
$t727(caterpillar\_food\_of(X1)) \leftarrow t41(X1)$
$t727(a\_fox) \leftarrow$
$t727(a\_bird) \leftarrow$
$t727(a\_caterpillar) \leftarrow$
$t727(a\_snail) \leftarrow$
$t53(a\_snail) \leftarrow$
$t41(a\_caterpillar) \leftarrow$

Program 7.1: SteamR—Regular Approximation for $eats(X, Y)$

cialised program P_3. This removes all unnecessary contrapositives from P_2 which removes another inefficiency of these proof procedures. This leaves us with a final proof procedure with one inference rule and only two added contrapositives (each clause in the above theory gives rise to only one clause in the specialised procedure, except clause 13 which becomes three clauses).

A speedup of approximately 33 times over running standard model elimination results. However, there is hardly any overall gain as the approximation time takes up most of the time saved. We should view this speedup figure however in the right context: we started from a naive version of model elimination. The speedup is unrealistic if a more sophisticated version of model elimination will be used. However, some of the optimisations present in the more sophisticated version of model elimination will also be present in the specialised program and the runtime of the specialised program will therefore improve as well. As pointed out in the previous chapter, we also believe that some of the optimisation achieved cannot be easily expressed in the meta-level proof procedure.

Although the specialised procedure is incomplete the possibility of finding a solution, as indicated by the regular approximation, gives us confidence to proceed with the specialisation

1. $\neg at(A, there, B) \vee \neg at(A, here, B)$
2. $\neg hold(thing(A), do(let\_go, B))$
3. $\neg red(hand)$
4. $at(hand, A, do(go(A), B))$
5. $at(thing(s), here, now)$
6. $\neg at(thing(A), B, do(go(B), C)) \vee at(thing(A), B, C) \vee hold(thing(A), C)$
7. $\neg at(hand, A, B) \vee \neg at(thing(C), A, B) \vee hold(thing(taken(B)), do(pick\_up, B))$
8. $\neg hold(thing(A), B) \vee \neg at(hand, C, B) \vee at(thing(A), C, B)$
9. $\neg hold(thing(A), B) \vee \neg at(thing(A), C, B) \vee at(hand, C, B)$
10. $\neg red(A) \vee \neg at(A, there, B) \vee answer(B)$
11. $\neg at(thing(A), B, C) \vee at(thing(A), B, do(go(B), C))$
12. $\neg hold(thing(A), B) \vee hold(thing(A), do(go(C), B))$
13. $\neg at(hand, A, B) \vee at(thing(taken(B)), A, B)$
14. $\neg at(A, B, C) \vee at(A, B, do(pick\_up, C))$
15. $\neg at(A, B, C) \vee at(A, B, do(let\_go, C))$
16. $\neg at(A, B, do(let\_go, C)) \vee at(A, B, C)$
17. $\neg at(A, here, now) \vee red(A)$
18. $\neg answer(A)$

Theory 7.2: MSC001-1—BHand1—A Blind Hand Problem

process. A point of concern is: what happens in the case when the regular approximation falsely indicates that a possible proof exists. The incomplete proof system is also most likely to be undecidable.

A solution to this problem might be to run a coroutining scheme between the complete proof procedure and the incomplete proof procedure. In the worst case double the amount of work will now be done, but with the possibility of finding some proofs to difficult theorems extremely quickly. However, out of experimental evidence gathered so far, we do not believe that the regular approximation will let us down in many cases. We feel that the risk is worth taking and may provide significant speedups.

7.3.2 A Blind Hand Problem

Consider as a second example the Blind Hand Problem(APA BHP) (MSC001-1, in [104]) given in Theory 7.2.

Repeating INSPECTOR for this Blind Hand Problem (again using only input resolution as the incomplete proof procedure) gives us a non-empty approximation. A possible proof therefore exists using only input resolution and we continue with the proposed optimisations. This time the speedup is approximately 25 times. We also do not show an overall gain, although the specialised incomplete provers is very efficient. As we have indicated in the previous chapters, we feel that it might be possible to decrease the analysis time so as to make the whole process worthwhile.

7.4 Discussion

The two examples analysed in the previous sections are examples from combinatorially difficult problems from the literature of first-order theorem proving. When we started this experiment we did not know that Schubert's Steamroller Problem and the Blind Hand Problem were provable using only an input resolution theorem prover. The simple proof obtained for Schubert's Steamroller Problem came as a pleasant surprise. As we have shown, you do not need a full first-order theorem prover to prove the given theorems or more generally, to prove inconsistency of the object theory with the theorem added in negated form. We suspect that there are many more first-order problems in the TPTP Problem Library that belong to this category.

The goal of the theorem prover will dictate if such a process is applicable or not. If all possible proofs to a given theorem need to be found, this process is obviously not suitable as completeness of the original proof procedure is not preserved. However if the aim of the theorem prover is to find a proof of a given theorem quickly, then the described process can be considered.

Although we lose completeness in the specialised proof procedure, we always have the original proof procedure to fall back on and therefore do not lose overall completeness. This will happen when we have spent too much time in the specialised procedure (without finding a proof) or the approximation indicates that no proof exists with the specialised procedure. However, as the specialised proof procedure is much simpler than the original proof procedure we started off with, we can search to a greater depth for a possible proof than would have been possible using the original proof procedure and some time limit.

In this chapter we have shown only two "positive" examples where the approximation indicated that a proof might exist. An example where the approximation is empty when the proof procedure consists of only input resolution, is the prime number theorem (NUM014-1, [104]): if a is prime and $a = b^2/c^2$ then a divides b. This theorem can be proved using only input resolution, but then signs of predicates needs to be reversed which we do not

consider. Combining our analysis with methods which transform programs by reversing signs on literals might lead to further improvements.

In Section 7.1 we indicated that sharing information has been lost during the approximation. It is possible to recover the sharing information with a more complicated analysis. This will make the approximation even more precise and eliminate more unsuccessful proof attempts using incomplete proof procedures. We are currently working on such an extension. However, as the analysis will now be more complicated and will take longer to compute, the disadvantage of spending more time on analysis must be weighed up against the possible gains.

Chapter 8

Rewriting Proof Procedures

An alternative method of deleting unnecessary inference rules using only partial evaluation and a well-designed reimplementation of the theorem prover is presented. This alternative method is less general than the methods presented in the previous chapters, but is nevertheless an important way of specialisation that has a methodological flavour and puts more emphasis on the ingenuity of the programmer than on the power of the analysis and transformation tools.

The aim in this chapter is to show how partial evaluation of a well-designed proof procedure with respect to some object theory can produce results comparable to that obtained by a special purpose static analysis applied to a more naive version of the proof procedure and object theory.

We have already argued in Chapter 3 that partial evaluation alone is not able to specialise effectively some proof procedures. The most we can expect from partial evaluation is the deletion of the overhead incurred by the meta-program and some local optimisations [36, 38].

We give a version of the model elimination proof procedure that is amenable to partial evaluation and show how specialisation of this procedure with respect to some object theory can give results comparable to the results obtained by the special purpose analysis developed by Poole and Goebel [85].

This rewriting method could be very useful when an implementation of the extended analysis and transformation techniques is not available. Some progress in the specialisation of the proof procedure might then be made by an intelligent rewrite of the proof procedure.

The fact that the result of partial evaluation depends directly on the way the meta-program

is written is not encouraging at all. However, this comparison between two apparently very similar programs that give notably different specialisation results, may suggest ways to make meta-programs more amenable to specialisation.

8.1 Limitations of Partial Evaluation

From Theorem 5.3.1 and Theorem 5.3.2 it is known that for any definite object theory the negative ancestor check is redundant. To reinforce the point that partial evaluation is not able to specialise effectively the model elimination proof procedure, we give a partial evaluation of the prover given in Program 2.1 with respect to naive reverse (a definite theory) with top-level goal $prove(reverse(\_, \_), [\,])$ in Program 8.1.

Most of the overheads of the meta-program have been eliminated. All that remains is the specialised *prove* procedures for *reverse* and *append* and a general member procedure, *member_1*. The negative ancestor check stays intact, although it is not needed. Partial evaluation was unable to delete the negative ancestor check because it is unable to detect that all calls to *member_1* will fail finitely (all ancestor lists will be of finite length). The negative ancestor list will only contain instances of $not(reverse)$ and $not(append)$ and the first argument of all calls to *member_1* will never contain a negated atom. Because partial evaluation has to terminate (to be of any use), partial evaluation cannot evaluate all possible ancestor checks. It therefore has to generalise at some stage to terminate and to guarantee that a successful branch has not been pruned from the search tree.

The "problem" with the prover given in Program 2.1 is that the occurrence (and absence) of all instances of negated ancestor literals are kept in one list. It is therefore impossible to distinguish between the absence of a negated ancestor literal in the ancestor list and the generalisation introduced by partial evaluation for termination purposes.

Although it might seem unreasonable at this stage to expect partial evaluation to do any more than what was achieved in Program 8.1, we will now show that the negative ancestor check can be deleted by rewriting the model elimination theorem prover and partially evaluating the improved prover with respect to naive reverse.

$prove(reverse(X0, X1), [X2|X3]) \leftarrow member\_1(reverse(X0, X1), X2, X3)$
$prove(reverse([\,], [\,]), X0) \leftarrow$
$prove(reverse([X0|X1], X2), X3) \leftarrow prove\_2(X1, X4, X0, X2, X3),$
$\quad prove\_3(X4, X0, X2, X1, X3)$

$member\_1(X0, X0, X1) \leftarrow$
$member\_1(X0, X1, [X2|X3]) \leftarrow member\_1(X0, X2, X3)$

$prove\_2(X0, X1, X2, X3, [X4|X5]) \leftarrow member\_1(reverse(X0, X1), X4, X5)$
$prove\_2([\,], [\,], X0, X1, X2) \leftarrow$
$prove\_2([X0|X1], X2, X3, X4, X5) \leftarrow$
$\quad prove\_2(X1, X6, X0, X2, [\neg reverse([X3, X0|X1], X4)|X5]),$
$\quad prove\_3(X6, X0, X2, X1, [\neg reverse([X3, X0|X1], X4)|X5])$

$prove\_3(X0, X1, X2, X3, [X4|X5]) \leftarrow member\_1(append(X0, [X1], X2), X4, X5)$
$prove\_3([\,], X0, [X0], X1, X2)$
$prove\_3([X0|X1], X2, [X0|X3], X4, X5) \leftarrow$
$\quad prove\_4(X1, X2, X3, X0, reverse([X2|X4], [X0|X3]), X5)$

$prove\_4(X0, X1, X2, X3, X4, [X5|X6]) \leftarrow member\_1(append(X0, [X1], X2), X5, X6)$
$prove\_4([\,], X0, [X0], X1, X2, X3) \leftarrow$
$prove\_4([X0|X1], X2, [X0|X3], X4, X5, X6) \leftarrow$
$\quad prove\_4(X1, X2, X3, X0, append([X4, X0|X1], [X2], [X4, X0|X3]), [not(X5)|X6])$

Program 8.1: "Uninteresting" Partial Evaluation

8.2 Clausal Theorem Prover Amenable to Partial Evaluation

A possible way to rewrite the model elimination proof procedure is to have a finite number of ancestor lists: one list for each positive and each negative predicate symbol occurring in the considered object theory. All literals that are added to the ancestor list are then added only to the list with correspoding positive or negative predicate symbol. As the negative ancestor check can only succeed when there is at least one literal in the ancestor list, an empty list would indicate failure of the ancestor check for this specific literal.

The idea of having a finite number of ancestor lists is not new. The ancestor list has been

$solve(G, A, D) \leftarrow prove(G, A, D).$

$prove(G, A, D) \leftarrow D > 0, memberl(G, A)$
$prove(G, A, D) \leftarrow D > 1, D1 \; is \; D - 1,$
$\quad clause((G :- B)),$
$\quad neg(G, GN),$
$\quad addanc(GN, A, GNA),$
$\quad proveall(B, GNA, D1)$

$proveall([\,], \_, \_) \leftarrow$
$proveall([G|R], A, D) \leftarrow prove(G, A, D),$
$\quad proveall(R, A, D)$

$memberl(G, [[A|As]|\_]) \leftarrow bound(G, A),$
$\quad member(G, As)$
$memberl(G, [[A|\_]|B]) \leftarrow$
$\quad memberl(G, B)$

$addanc(G, [[A|As]|Bs], [[A, G|As]|Bs]) \leftarrow bound(G, A)$
$addanc(G, [[A|As]|B], [[A|As]|C]) \leftarrow$
$\quad addanc(G, B, C)$

Program 8.2: Clausal Theorem Prover Amenable to Partial Evaluation

implemented in this way in the Prolog Technology Theorem Prover [100]. However, there it was done for efficiency reasons and not with the aim of deleting ancestor checks through the use of partial evaluation. This also confirms that our amenable clausal theorem prover is a reasonable program that we might expect from a competent programmer.

Consider the theorem prover in Program 8.2 from de Waal [23] that is amenable to partial evaluation with a very simple "user controlled" iterative deepening procedure to make the program runnable. $memberl(X, Y)$ is true if X is a member of one of the lists that constitutes Y. $addanc(X, Y, Z)$ is true if X can be added to one of the lists constituting Y to give resulting list Z. $bound(X, Y)$ is true if the predicate symbol appearing in X matches one of the predicate symbols in Y. We now use the directive introduced in Chapter 3 to force the unfolding of *addanc*. Because there are only a finite number of sublists in the ancestor list, the unfolding will always terminate.

$$solve(reverse(X0, X1), [[reverse], [append], [not(reverse)], [not(append)]], X2) \leftarrow$$
$$\quad prove\_1(X0, X1, [\,], X2)$$

$$prove\_1([\,], [\,], X0, X1) \leftarrow X1 > 1, X2 \; is \; X1 - 1$$
$$prove\_1([X0|X1], X2, X3, X4) \leftarrow X4 > 1, X5 \; is \; X4 - 1,$$
$$\quad prove\_1(X1, X6, [not(reverse([X0|X1], X2))|X3], X5),$$
$$\quad prove\_2(X6, X0, X2, X1, X2, X3, [\,], X5)$$

$$prove\_2([\,], X0, [X0], X1, X2, X3, X4, X5) \leftarrow X5 > 1, X6 \; is \; X5 - 1$$
$$prove\_2([X0|X1], X2, [X0|X3], X4, X5, X6, X7, X8) \leftarrow X8 > 1, X9 \; is \; X8 - 1,$$
$$\quad prove\_2(X1, X2, X3, X4, X5, X6, [not(append([X0|X1], [X2], [X0|X3]))|X7], X9)$$

Program 8.3: Improved Partial Evaluation

Partial evaluation of this program with respect to naive reverse with an unbounded depth bound and top-level goal $solve(reverse(\_, \_), [[reverse], [append], [not(reverse)], [not(append)]], \_)$ gives the partially evaluated program in Program 8.3.

This program includes most of the optimisations discussed by Poole and Goebel [85]. All the unnecessary contrapositives have been deleted. The negative ancestor check has been eliminated. However, the redundant ancestor list is still being built although it is never used. In this example it is much more efficiently handled than was previously the case and partial evaluation has further optimised it. This program is a considerable simplification of the partially evaluated program given in Program 8.1.

The example in the next section shows that partial evaluation can selectively delete redundant negative ancestor checks where appropriate.

8.3 Example

Consider the prime number theorem, originally from Chang and Lee [17], from the TPTP Problem Library given in Theory 8.1. Partial evaluation of the prover given in Program 8.2 with respect to this theory and top-level goal $solve(divide(\_, \_), [[prime], [product], [divides], [\neg(prime)], [\neg(product)], [\neg(divides)]], \_)$ gives the partially evaluated program shown in Program 8.4.

1. $product(X, X, square(X))$
2. $\neg product(X, Y, Z) \vee product(Y, X, Z)$
3. $\neg product(X, Y, Z) \vee divides(X, Z)$
4. $\neg prime(X) \vee \neg product(Y, Z, U) \vee \neg divides(X, U) \vee divides(X, Y) \vee divides(X, Z)$
5. $prime(a)$
6. $product(a, square(c), square(b))$
7. $\neg divide(a, b)$

Theory 8.1: NUM014-1—VallsPrm—Prime Number Theorem

The only ancestor check remaining is the check for $\neg(divides(\_, \_))$. Note also that for this example the redundant building of the negative ancestor list that we experienced with the naive reverse example has nearly been completely eliminated (note the many [ ]'s appearing in the negative ancestor list arguments). However, this is very much dependent on the object theory being used and needs further investigation. The above program is efficient and finds a proof to the prime number theorem in a few milliseconds.

8.4 Discussion

Althought we rewrote a specific theorem prover in this chapter, this does not preclude generalisation of this rewriting method to other meta-program or theorem provers. Good specialisation results can also be expected by applying this method to meta-programs where:

- the meta-program collects some "history" and makes decisions based on this information and
- there is a finite number of predicate names in the object theory.

The crucial idea is the identification of some property of the object theory that may be exploited by the meta-program to achieve useful specialisation through partial evaluation. Optimisations that were previously only possible with specialised analysis techniques may now be possible with partial evaluation. However, as we pointed out in the introduction, this rewriting technique is less general than the methods developed in Chapter 5. If the specialisations depend on information other than the predicate symbol and sign, then this method may not be applicable. It can therefore only be used in special cases and we should not rely on this technique for all our specialisations.

$$solve(d(X0, X1), [[prime], [product], [divides], [\neg(prime)], [\neg(product)],$$
$$[\neg(divides)]], X2) \leftarrow$$
$$prove\_1(X0, X1, [\,], X2)$$

$$prove\_1(X0, X1, X2, X3) \leftarrow X3 > 1, X4 \; is \; X3 - 1,$$
$$prove\_2(X0, X5, X1, [\,], [\,], X0, X1, X2, X4)$$
$$prove\_1(a, X0, X1, X2) \leftarrow X2 > 1, X3 \; is \; X2 - 1, X3 > 1, X4 \; is \; X3 - 1,$$
$$prove\_2(X5, X0, X6, [\,], [\,], a, X0, X1, X3),$$
$$prove\_1(a, X6, [\neg divides(a, X0)|X1], X3),$$
$$prove\_5(a, X5, [\,], X0, X1, X3)$$
$$prove\_1(a, X0, X1, X2) \leftarrow X2 > 1, X3 \; is \; X2 - 1, X3 > 1, X4 \; is \; X3 - 1,$$
$$prove\_2(X0, X5, X6, [\,], [\,], a, X0, X1, X3),$$
$$prove\_1(a, X6, [\neg divides(a, X0)|X1], X3),$$
$$prove\_5(a, X5, [\,], X0, X1, X3)$$

$$prove\_2(a, square(c), square(b), X0, X1, X2, X3, X4, X5) \leftarrow X5 > 1, X6 \; is \; X5 - 1$$
$$prove\_2(X0, X0, square(X0), X1, X2, X3, X4, X5, X6) \leftarrow X6 > 1, X7 \; is \; X6 - 1$$
$$prove\_2(X0, X1, X2, X3, X4, X5, X6, X7, X8) \leftarrow X8 > 1, X9 \; is \; X8 - 1,$$
$$prove\_2(X1, X0, X2, X3, [\neg product(X0, X1, X2)|X4], X5, X6, X7, X9)$$

$$prove\_5(X0, X1, X2, X3, X4, X5) \leftarrow X5 > 0,$$
$$member\_1(X0, X1, [\neg divides(X0, X3)|X4])$$
$$prove\_5(a, X0, X1, X2, X3, X4) \leftarrow X4 > 1, X5 \; is \; X4 - 1, X5 > 1, X6 \; is \; X5 - 1,$$
$$prove\_2(X7, X8, X0, [divides(a, X0)|X1], [\,], a, X2, X3, X5),$$
$$prove\_5(a, X7, [divides(a, X0)|X1], X2, X3, X5),$$
$$prove\_5(a, X8, [divides(a, X0)|X1], X2, X3, X5)$$

$$member\_1(X0, X1, [\neg divides(X0, X1)|X2]) \leftarrow$$
$$member\_1(X0, X1, [X2|X3]) \leftarrow member\_1(X0, X1, X3)$$

Program 8.4: Partially Evaluated Prime Number Theorem

There are at least three possible solutions that need investigation when the specialisation of programs are considered.

1. Develop a problem specific analysis method that can be used to get the required specialisation. This was done by Poole and Goebel [85].

2. Extend current analysis and transformation techniques. This option was investigated in this thesis for proof procedures written as meta-programs.

3. Rewrite the meta-program so that it becomes amenable to program specialisation. This was the subject of this chapter.

Chapter 9

Conclusions

In this chapter we discuss the contributions of this thesis to the fields of automated theorem proving, partial evaluation, abstract interpretation and meta-programming. Second, we contrast our results with that achieved by related work. Finally, we discuss ideas for future research.

9.1 Contributions

The main contribution of this thesis is that we have shown how the transformation and analysis of a proof procedure, object theory and formula (possibly a theorem), as a complete unit, may lead to the inference of significant information that may speed up theorem provers.

More specifically, for the field of automated theorem proving we have demonstrated practically how reasoning with approximations may lead to a useful decrease in the search space of a theorem proving problem. We have shown how proof procedures currently of interest to the theorem proving community [100, 6, 70], can be transformed and analysed with state-of-the-art logic programming techniques. The results inferred by the proposed methods are interesting and useful as we have demonstrated in Chapter 6.

The meta-programming approach has been shown to be useful in the analysis of theorem provers. A wide variety of analysis and transformation techniques developed for logic programs have been made available for use in theorem proving. Although theorem provers implemented as meta-programs may not yet be able to compete directly with state-of-the-art theorem provers, because of the relative slowness of declarative languages and the restriction placed on the implementation of theorem provers as definite logic programs in this thesis,

the developed techniques might assist in closing the performance gap even further.

The current correctness requirements for partial evaluation place restrictions on the types of specialisation possible. We relaxed these requirements to fit the theorem proving domain. We showed how a simple renaming of literals in a resultant may improve the amenability of a program generated by partial evaluation to approximation.

One of the first practical uses of approximation in theorem proving was demonstrated in this thesis. We have shown that regular approximations are a very useful approximation that can be computed in reasonable time capturing information important to theorem proving.

The adaptation of a bottom-up approximation framework with query-answer transformations to capture query information to a program was successfully exploited. Variations on the basic query-answer transformation were developed that capture important information. This information can be used to improve the precision of the regular approximation procedure. A close symbiosis between partial evaluation and regular approximation was also proposed.

The approximation times given in Table 6.1 are disappointing compared with the times theorem provers take to prove theorems. However, the principle of obtaining speedups by the construction of and reasoning with approximations has been established for practical problems. If a significant reduction in the approximation times can be achieved, a preprocessing step consisting of identifying useless inference rules and redundant object theory clauses may become a worthwhile addition to current theorem provers.

It is important to point out that several critical refinements of the basic techniques used in this thesis were necessary to infer the information presented in our results. We list these refinements.

- Adding static information to the meta-programs to assist in the control of unfolding (Chapter 2).
- Renaming different instances of inference rules (Chapter 3).
- Intersecting analysis results based on different computation rules to increase the precision of the approximation (Chapter 4).
- Combining partial evaluation, abstract interpretation and a simple decision procedure into one specialisation method (Chapter 5).

The omission of any one of of these refinements may destroy the precision and usefulness of the inferred information.

9.2 Related Work

9.2.1 Automated Theorem Proving

We now discuss other research related to the research presented in this thesis. However, the goals of this research were slightly different from ours.

Giunchiglia and Walsh developed a general theory of abstraction for automated theorem proving [44]. This may be seen as a theory of abstraction analogous to the theory of abstract interpretation developed by Cousot and Cousot [20] for logic programming.

Their viewpoint is that the theory needs changing and not the logic. This differs from our approach in that we approximate a theory and proof procedure for the logic of the theory with a regular unary logic program. Although the resulting regular approximation is still a logic program, it is actually a program in a different logic and a procedure other than SLDNF-resolution should be used to decide success or failure given some goal. In previous chapters we only checked for the existence of a regular approximation and did not go into much detail about the decision procedures that exist for regular unary logic programs.

Furthermore, a different abstraction is not designed for every logic or proof procedure. One carefully chosen abstraction, namely regular unary logic programs, is the only abstraction used. This allows more time to be spent on improving the speed and precision of the chosen approximation system as it does not change with the logic or proof procedure we wish to approximate.

A class of mappings, called abstractions, was defined by Plaisted [83]. These abstractions maps a set of clauses S onto a possibly simpler set of clauses T. Resolution proofs from S map onto simpler resolution proofs from T. An "abstract proof" of a clause C is constructed using T and the abstraction mapping is then inverted to obtain a proof of C from S.

The main difference with our work is that we do not only map a set of first-order clauses into a simpler set of clauses, but also map the proof procedure into a possibly simpler proof procedure. The approximation we construct does not distinguish between the clauses of the first-order theory and the clauses of the proof procedure (partial evaluation removes this distinction). Furthermore, we also do not try to construct a "real proof" from an "abstract proof". Our aim is to prune the search space and gain decidability with the construction of an approximation. Some of the proposed abstractions, such as deleting arguments are very crude compared to the regular approximations we construct.

Poole and Goebel in their paper "Gracefully Adding Negation and Disjunction to Prolog"

1. $a \leftarrow b \wedge c$;
2. $a \vee b \leftarrow d$;
3. $c \vee e \leftarrow f$;
4. $\neg g \leftarrow e$;
5. $g \leftarrow c$;
6. g;
7. $f \leftarrow h$;
8. h;
9. d;

Theory 9.1: Set of clauses from Poole and Goebel

[85] showed how to add negation and disjunction to Prolog. Their aim was to show that there is no overhead in their method when the negation and disjunction is not used.

Their method is based on an extension of Loveland's MESON proof procedure (see Program 2.1), which requires that a negative ancestor search and the availability of contrapositive forms of formulas be added to Prolog. The notions of a literal being relevant and askable are introduced. Askable corresponds to a literal being generated as a subgoal and relevant to a literal being provable within the context of its ancestors. As this is generally undecidable, the notions of potentially askable and potentially relevant are introduced that are decidable and can be computed at runtime. This leads to an analysis on signed predicate symbols indicating when contrapositives are not needed and ancestor resolution not required.

We now argue that our analysis is at least as precise as that of Poole and Goebel. In a corollary they state that an ancestor search is only necessary if both a literal and its negation are potentially askable and potentially relevant, in which case ancestor resolution is necessary for both the positive and negative instances of a literal. It is therefore impossible to distinguish between the necessity of ancestor search of a positive and negative literal. As shown in our results, this have been done for the prime number theorem (NUM014-1 in the TPTP Problem Library), where ancestor resolution is only necessary for $\neg divides(X, Y)$. The best their analysis can do is to indicate that ancestor resolution is necessary for $divides(X, Y)$ and $\neg divides(X, Y)$. Also, their restriction to signed predicate symbols corresponds to the analysis we did when specialising the ground representation in Chapter 6. However, our analysis is potentially still more precise as pointed out in the previous paragraph, even for the ground representation.

The set of clauses in Theory 9.1 was analysed in [85]. Their analysis indicated that the nega-

tive ancestor search is only necessary for a, $\neg a$, b and $\neg b$. Our analysis using KERNEL indicates that ancestor resolution is only necessary for $\neg a, h$ and f (with query $\leftarrow solve(\_, [\,])$). h and f are the result of the merging of results for different literals and query-answer patterns during the regular approximation.

However, if we do a separate analysis for each positive and negative literal (e.g. query of the form $\leftarrow solve(a, [\,])$), and then "merge" the results, we get that only $\neg a$ need to be considered during the ancestor search. For this example, even when restricted to signed predicate symbols our analysis is more precise than that in [85]. This should always be the case as the two notions in [85] have equivalent or more precise counterparts in our method.

In [103] Sutcliffe describes syntactically identifiable situations in which reduction does not occur in chain format linear deduction systems (e.g. model elimination, SL-resolution). His aim is to detect linear-input subdeductions that can be solved without reduction steps to restrict the search space. He develops three analysis methods to detect such linear-input subdeductions. The most powerful of the tree methods is the Linear-Input Subset for Literals (LISL) analysis.

An extension tree whose nodes are literals from the input set is constructed. A literal is in the LISL if and only if it is not complementary unifiable with an ancestor and all of its descendants are in the LISL.

Compared with our analysis Sutcliffe's analysis is very precise and may infer the same information for ancestor resolution. However, our method is not restricted to ancestor resolution and we may infer results also for input resolution as the next examples show.

The object theory given in Theory 9.2 from [103] was analysed using LISL analysis. His analysis method indicates that no reductions are needed for the subset $\{r, \neg t, u, \neg s, \neg p(b)\}$ and query $\leftarrow r \wedge \neg p(a) \wedge q$. Our analysis method infers exactly the same results when KERNEL is applied to this example. However, it is easy to construct an example where our analysis will indicate failure and LISL analysis will not infer any useful information. For example, consider the object theory given in Theory 9.3 which is a subset of the clauses in the previous example. Consider also the queries $\leftarrow q$ and $\leftarrow p(a)$. In this case LISL analysis infers no useful information, but our analysis applied to the program given in Program 2.2 and the above two clauses gives an empty approximation for *solve* which indicates failure.

Wakayama [107] defines a class of Case-Free Programs that requires no factoring and ancestor resolution. A characterisation of Case-Free programs is given, but no procedure is given for detecting instances of Case-Free programs. The method in this paper can be used to detect instances of Case-Free programs. However, we do not claim that our method will detect all Case-Free programs as we work with approximations to check this. Our method

1. $\neg r \vee \neg p(a) \vee \neg q$
2. $\neg p(a) \vee q$
3. $p(a) \vee \neg q$
4. $p(a) \vee q$
5. $\neg r \vee \neg t \vee \neg s$
6. $t \vee u$
7. $\neg u$
8. $s \neg p(b)$
9. $p(b)$

Theory 9.2: Set of clauses from Sutcliffe

1. $\neg p(a) \vee q$
2. $p(a) \vee \neg q$

Theory 9.3: Subset of clauses from Sutcliffe

is comparable when considering all possible derivations from a given theory, but also allows "Case-Freeness of particular proofs" to be inferred, as in the DBA BHP example. This is a stronger notion than just CASE-Freeness and a larger class of theories fall into this class.

In [96, 56] Kautz and Selman use the idea of approximation for propositional theorem proving. A "Horn Approximation" of any propositional theory can be constructed. Since the complexity of proving theorems in Horn theories is less than the general case it may be worth testing formulas in the approximating theory before applying a general propositional theorem prover. Some non-theorems can be thrown out fast. This is an example of the same principle of reasoning by approximation.

In that work the main benefit of using an approximation is efficiency, whereas in the approximations we propose the main benefits are decidability and efficiency, since our approximating theories are regular theories for which theoremhood is decidable.

It is worth emphasising that the methods developed in this thesis are independent of the proof procedure which is not the case for the methods developed by Poole and Goebel, Sutcliffe, Wakayama and Kautz and Selman. Any theorem prover that can be written as a logic program can be analysed and optimised using this method.

Many of the classic optimisations in automated theorem proving such as MULT, PURE, TAUT, SUBS, UNIT and ISOL [11] have corresponding counterparts in our specialisation process. For instance, determinate unfolding in partial evaluation may be seen as an instance of UNIT and dead end detection [36] as an instance of PURE [80]. Although these optimisations are not explicitly expressed in the proof procedure in Program 2.2, they may be specialised into the proof procedure by the specialisation process.

One of the criticisms against model elimination is the requirement that all contrapositives of clauses must be available for deduction steps [84]. In many cases our method eliminates many unnecessary contrapositives and so also reduces the size of the search space. Examples of this can be found in Chapter 6 where all contrapositives can be deleted that have a literal that cannot contribute to a proof (information given for input resolution). A recent alternative approach to solving this problem was presented by Baumgartner and Furbach [6], where a modification to model elimination was proposed that does not necessitate the use of contrapositives. A restart operation similar to that of nH-Prolog was proposed. This model elimination proof procedure without contrapositives is another candidate for our specialisation methods and it should be interesting to compare the results inferred for their proof procedure with that presented in this thesis.

Other techniques for optimising model elimination, such as the Positive Refinement of Plaisted [84] reduces the need to do ancestor resolution and may make our inferred information less useful. However, ancestor resolution still has to be done (or some other mechanism has to be introduced that replaces ancestor resolution) and in many cases our analysis information still leads to a reduction in the number of ancestor resolutions that needs to be done. Mayr [74] also presented refinements and extensions to model elimination. In his extension he added another inference rule to the basic model elimination procedure to facilitate the multiple use of the same subproofs. A comparison of three Prolog extensions (including nH-Prolog) can be found in [89]. We now turn to the techniques used in the proposed specialisation methods. Relevant research is discussed in the appropriate sections.

9.2.2 Partial Evaluation

Two pieces of research already mentioned in previous chapters are of interest. The first is the logic program specialisation system SP [36] developed by Gallagher which was extensively used during this research. The second is the self applicable partial evaluator for Gödel [46] by Gurr. We restrict this discussion to aspects of their research we found relevant to the specialisation methods developed in this thesis.

The SP-System provides a choice of unfolding strategies: certain strategies being more

applicable to certain types of programs and applications. Currently ten choices are provided. At least one of the choices includes a renaming similar to that described in Chapter 4. Determinate unfolding and different levels of abstraction are also included in some choices. However, none of the current strategies consistently provides satisfactory partial evaluations amenable to regular approximations. Termination problems for some object theories from the TPTP Problem Library were experienced with at least one of the more specialised unfolding rules. Generalisations of this rule to enforce termination sometimes lost important information that are needed in the detection of useless clauses.

However, the idea of providing different unfolding strategies for different applications seems very promising. Taking this idea one step further, different unfolding strategies can be provided for the various proof procedures to get the best partial evaluations possible. This may sound very complicated, but as we are already working in a very specialised domain, this idea needs further investigation. Such an unfolding rule was written for the proof procedure in Program 2.1 that gives consistent good specialisations for all the problems in the TPTP Problem Library tried so far.

The attraction of the research by Gurr is that he handles a full logic programming language in his partial evaluation system, SAGE. We have concentrated on definite programs without side effect during our discussion of partial evaluation. As Gödel is a declarative logic programming language supporting a ground and non-ground representation, it seems that incorporation of our ideas regarding partial evaluation into SAGE and combining it with the deletion of useless clauses may extend the applicability of our ideas beyond the field of theorem proving into general logic programming. This may also make it easier to write a proof procedure as an efficient logic program as pruning and other features of the language may then also be used and it can be partially evaluated efficiently and effectively.

9.2.3 Abstract Interpretation

In Chapter 3 we improved partial evaluation to generate different renamed versions of predicates based on the argument information. In principle, we could have left this specialisation until after the approximation phase. To do this, we need an approximation system that can analyse different query patterns (calls) to the same predicate and generate a corresponding answer pattern for each generated query pattern. These query-answer patterns can then be used to generate different specialised versions of the predicates in our program.

We chose not to follow this route as another non-trivial specialisation phase would be needed after the approximation phase to generate the specialised predicates. Also, because the renaming of predicates is done by the partial evaluation phase, an already complex

abstraction phase is not further complicated.

Another option is to incorporate abstract interpretation into the partial evaluation algorithm. A general approach towards the incorporation of abstract interpretation into partial evaluation was developed by de Waal and Gallagher [24]. However, as we have so far concentrated on regular approximations, we restricted the discussion to the incorporation of regular approximations into partial evaluation.

In the previous chapters partial evaluation and regular approximation were separate phases in the specialisation process. A benefit of this separation is that the partial evaluation and regular approximation algorithms may be independently designed, implemented, tested and evaluated. Furthermore, partial evaluation and regular approximation only have to be done once for every program.

A drawback however, is a possible loss of specialisation information as during each iteration of the basic partial evaluation algorithm, (see Figure 3.1), variable sharing information between literals in a resultant may be lost (when the set S is formed). A further drawback is that partial evaluation can only consider a finite number of subtrees of the computation because it has to terminate.

The ideal place to call regular approximations from within the basic partial evaluation algorithm would be where the set S is formed by collecting atoms from the resultants. We stop the creation of further resultants (and therefore further clauses in the partially evaluated program) that might have followed from clauses that are now indicated useless by the regular approximation. The revised algorithm incorporating regular approximation is given in Figure 9.1. $approx(Q, p(t), Q')$ is computed during every iteration of the partial evaluation algorithm and may now fail resultants, before they are added to S. This is very inefficient and in any practical implementation of this algorithm, this has to be reconsidered.

The tight integration of partial evaluation and regular approximation or abstract interpretation still needs a lot of research to find the right balance. Some ideas of how to proceed were sketched in this section. Looking back to the methods presented in Chapter 5, we see that a good compromise between computational complexity and usefulness was achieved that is not so easy to improve on.

9.2.4 Meta-Programming

The requirement that the proof procedure be represented as a meta-program might be regarded as the "weak link" in the approach presented in this thesis. We argue that this is not the case even though we have only concentrated on representing the three specialised

$$
\begin{array}{l}
begin \\
\quad A_0 \ := \ the\ set\ of\ atoms\ in\ G \\
\quad i \ := \ 0 \\
\quad repeat \\
\qquad P' \ := \ a\ partial\ evaluation\ of\ A_i\ in\ P \\
\qquad S \ := \ A_i \ \cup \ \{p(t) \mid (B \ \leftarrow \ Q, p(t), Q' \ in\ P' \\
\qquad\qquad\qquad\qquad AND \\
\qquad\qquad\qquad\qquad approx(Q, p(t), Q')\ is\ not\ empty) \\
\qquad\qquad\qquad OR \\
\qquad\qquad\qquad B \ \leftarrow \ Q, not(p(t)), Q'\ in\ P'\} \\
\qquad A_{i+1} \ := \ abstract(S) \\
\qquad i \ := \ i+1 \\
\quad until\ A_i \ = \ A_{i-1}\ (modulo\ variable\ renaming) \\
end
\end{array}
$$

Figure 9.1: Basic Partial Evaluation Algorithm Incorporating Regular Approximation

proof procedures as definite logic programs in Chapter 2.

We first consider the case where the methods developed in this thesis are only used to infer information for other theorem provers. The case where the aim is to get an efficient logic program as a result of applying these methods is then considered.

In the Prolog Technology Theorem Prover (PTTP) [100] one refinement of the basic model elimination procedure is to perform a reduction operation and cut (disallow any other inferences on the current goal) if the current goal is exactly complementary to one of its ancestor goals. A regular approximation of the above PTTP procedure with and without the cut can only differ in the fact that the regular approximation for the program without the cut may contain a loop in place of a possibly finite approximation for the program with the cut. The same information will be recursively represented instead of only once. This does not affect our decision procedure nor a more complex decision procedure that has to "look inside" the regular approximation. Eliminating pruning from the meta-program therefore has little or no effect on the precision of the approximation.

Negative literals may be included in the meta-program, but as they are ignored in the approximation, they are of no use other than to make the program easier to read or understand. The same holds for groundness and variable checks. We can therefore get a very

precise approximation from a simple representation of a proof procedure while ignoring many of the intricacies of the "real" theorem prover.

Next we consider the case where we want an efficient runnable theorem prover from our specialisation method. The proof procedure in Program 2.1 is a very naive implementation of a theorem prover based on model elimination. It may be improved by for instance adding pruning and possibly groundness and variable checks (to bring it up to current state-of-the-art implementation standards). As has been shown by Dan Sahlin in Mixtus [93] this may severely complicate the partial evaluation phase.

We made a deliberate decision in Chapter 1 not to get involved in the specialisation of the non-declarative features of Prolog. At least three options are available.

- Improve this "naive" theorem prover by extending it, but staying within the "declarative" definite logic program paradigm.
- Use all of or only selected parts of the "rest" of Prolog, but be prepared for a less ideal partial evaluation with greater complexity because of the extra features used.
- Switch to a ground representation as in Gödel. This will allow all the features of the Gödel language available for programming in this representation to be used. However, the partial evaluation is going to be complex and it might be that the specialisation results shown for definite programs in not achievable. This can only be checked by future research.

The approximation is not much influenced by the above considerations as pointed out in the previous paragraphs, except for possibly the ground representation as argued in Chapter 6. Only more research and experimentation will show whether the proposed techniques can be successfully extended beyond that presented in this thesis.

9.3 Future Work

This thesis may be regarded as the first general practical framework for the use of approximation in automated theorem proving. Several directions of research are now possible.

- Apply the developed techniques to other proof procedure (for instance non first-order logic proof procedures).
- Refine the analysis of the proof procedures studied in this thesis so as to get them incorporated into state of the art theorem provers.

- Investigate the use of other approximations (less or more precise) in the place of regular approximations.
- Improve the precision of the regular approximation procedure to include sharing information.
- Use a more complicated analysis system that includes groundness, freeness and mode information.
- Use Gödel as the specification-, target- and implementation language.

Each of the above suggestions may be pursued independently or we might try to make small improvements in the proposed methods by making selective improvements in some of the indicated areas.

An area of research that has not been mentioned so far, but that was instrumental in the development of this thesis, is constraint logic programming. At least two possibilities are open to investigation.

- Specialise constraint solvers using the techniques developed in this thesis (a constraint solver is just another proof procedure).
- Use a partial evaluator written in a constraint logic programming language, such as PrologIII for the partial evaluation of logic programs. The constraint domains (e.g. interval arithmetic, boolean algebra) can then be used in the construction of approximations and reasoning about approximations.

The first possibility was the catalyst for this thesis. The methods in this thesis have been applied to PrologIII list constraint solving, but because of the complexity of the constraint solver (43 rewrite rules) it is not practical at the moment to include in a compiler. This research was done under the ESPRIT Project PRINCE (5246) and may be continued in the future.

The second possibility looks very attractive as we will have all the power of the constraint domains with their associated proof procedures at our disposal to do constraint partial evaluation (not partial evaluation of constraints). We therefore use a more powerful language, e.g. PrologIII, to specialise a less powerful language, e.g. Prolog. The approximation phase can then be more tightly integrated into partial evaluation, as constraints can be directly associated with resultants during partial evaluation. The idea of calling constraint solvers during partial evaluation is one of the aims of current work on the partial evaluation of constraint logic programming.

Appendix A

Theorem 5.3.1: Minimal Partial Evaluation

```
% This program was generated by a process of unfolding and renaming from the model
% elimination proof procedure in Program 2.2 and the theory in Section 5.3.

solve(++atom(+(_179)),A) :- prove_1(++atom(+(_179)),A).
solve(++not_atom(-(_179)),A) :- prove_2(++not_atom(-(_179)),A).
solve(++clause(_179),A) :- prove_3(++clause(_179),A).
solve(++body(_179),A) :- prove_4(++body(_179),A).
solve(++contr(_179,_180),A) :- prove_5(++contr(_179,_180),A).
solve(++pick_a_head(+(_179),_182,_183),A) :- prove_61(++pick_a_head(+(_179),_182,_183),A).
solve(++pick_a_head(-(_179),_182,_183),A) :- prove_62(++pick_a_head(-(_179),_182,_183),A).
solve(++not(+(_181),-(_179)),A) :- prove_71(++not(+(_181),-(_179)),A).
solve(++not(-(_181),+(_179)),A) :- prove_72(++not(-(_181),+(_179)),A).
solve(++infer(+(_179),_182),A) :- prove_81(++infer(+(_179),_182),A).
solve(++infer(-(_179),_182),A) :- prove_82(++infer(-(_179),_182),A).
solve(++anc(+(_179),_182),A) :- prove_91(++anc(+(_179),_182),A).
solve(++anc(-(_179),_182),A) :- prove_92(++anc(-(_179),_182),A).
solve(++inferall(_179,_180),A) :- prove_10(++inferall(_179,_180),A).
solve(--atom(+(_179)),A) :- prove_11(--atom(+(_179)),A).
solve(--not_atom(-(_179)),A) :- prove_12(--not_atom(-(_179)),A).
solve(--clause(_179),A) :- prove_13(--clause(_179),A).
solve(--body(_179),A) :- prove_14(--body(_179),A).
solve(--contr(_179,_180),A) :- prove_15(--contr(_179,_180),A).
solve(--pick_a_head(+(_179),_182,_183),A) :- prove_161(--pick_a_head(+(_179),_182,_183),A).
solve(--pick_a_head(-(_179),_182,_183),A) :- prove_162(--pick_a_head(-(_179),_182,_183),A).
solve(--not(+(_181),-(_179)),A) :- prove_171(--not(+(_181),-(_179)),A).
solve(--not(-(_181),+(_179)),A) :- prove_172(--not(-(_181),+(_179)),A).
solve(--infer(+(_179),_182),A) :- prove_182(--infer(+(_179),_182),A).
solve(--infer(-(_179),_182),A) :- prove_182(--infer(-(_179),_182),A).
solve(--anc(+(_179),_182),A) :- prove_191(--anc(+(_179),_182),A).
```

```
solve(--anc(-(_179),_182),A) :- prove_192(--anc(-(_179),_182),A).
solve(--inferall(_179,_180),A) :- prove_20(--inferall(_179,_180),A).

prove_1(++atom(+(_191)),A) :- member_1(++atom(+(_191)),A).
prove_2(++not_atom(-(_191)),A) :- member_2(++not_atom(-(_191)),A).
prove_3(++clause(_191),A) :- member_3(++clause(_191),A).
prove_4(++body(_191),A) :- member_4(++body(_191),A).
prove_5(++contr(_191,_192),A) :- member_5(++contr(_191,_192),A).
prove_61(++pick_a_head(+(_191),_194,_195),A) :-
     member_61(++pick_a_head(+(_191),_194,_195),A).
prove_62(++pick_a_head(-(_191),_194,_195),A) :-
     member_62(++pick_a_head(-(_191),_194,_195),A).
prove_71(++not(+(_193),-(_191)),A) :- member_71(++not(+(_193),-(_191)),A).
prove_72(++not(-(_193),+(_191)),A) :- member_72(++not(-(_193),+(_191)),A).
prove_81(++infer(+(_191),_194),A) :- member_81(++infer(+(_191),_194),A).
prove_82(++infer(-(_191),_194),A) :- member_82(++infer(-(_191),_194),A).
prove_91(++anc(+(_191),_194),A) :- member_91(++anc(+(_191),_194),A).
prove_92(++anc(-(_191),_194),A) :- member_92(++anc(-(_191),_194),A).
prove_10(++inferall(_191,_192),A) :- member_10(++inferall(_191,_192),A).
prove_11(--atom(+(_191)),A) :- member_11(--atom(+(_191)),A).
prove_12(--not_atom(-(_191)),A) :- member_12(--not_atom(-(_191)),A).
prove_13(--clause(_191),A) :- member_13(--clause(_191),A).
prove_14(--body(_191),A) :- member_14(--body(_191),A).
prove_15(--contr(_191,_192),A) :- member_15(--contr(_191,_192),A).
prove_161(--pick_a_head(+(_191),_194,_195),A) :-
     member_161(--pick_a_head(+(_191),_194,_195),A).
prove_162(--pick_a_head(-(_191),_194,_195),A) :-
     member_162(--pick_a_head(-(_191),_194,_195),A).
prove_171(--not(+(_193),-(_191)),A) :- member_171(--not(+(_193),-(_191)),A).
prove_172(--not(-(_193),+(_191)),A) :- member_172(--not(-(_193),+(_191)),A).
prove_182(--infer(+(_191),_194),A) :- member_182(--infer(+(_191),_194),A).
prove_182(--infer(-(_191),_194),A) :- member_182(--infer(-(_191),_194),A).
prove_191(--anc(+(_191),_194),A) :- member_191(--anc(+(_191),_194),A).
prove_192(--anc(-(_191),_194),A) :- member_192(--anc(-(_191),_194),A).
prove_20(--inferall(_191,_192),A) :- member_20(--inferall(_191,_192),A).

member_1(++atom(+(_231)),[++atom(+(_231))|_]).
member_1(++atom(+(_231)),[_|A]) :- member_1(++atom(+(_231)),A).
member_2(++not_atom(-(_231)),[++not_atom(-(_231))|_]).
member_2(++not_atom(-(_231)),[_|A]) :- member_2(++not_atom(-(_231)),A).
member_3(++clause(_231),[++clause(_231)|_]).
member_3(++clause(_231),[_|A]) :- member_3(++clause(_231),A).
member_4(++body(_231),[++body(_231)|_]).
member_4(++body(_231),[_|A]) :- member_4(++body(_231),A).
member_5(++contr(_231,_232),[++contr(_231,_232)|_]).
member_5(++contr(_231,_232),[_|A]) :- member_5(++contr(_231,_232),A).
member_61(++pick_a_head(+(_231),_234,_235),[++pick_a_head(+(_231),_234,_235)|_]).
member_61(++pick_a_head(+(_231),_234,_235),[_|A]) :-
     member_61(++pick_a_head(+(_231),_234,_235),A).
```

```
member_62(++pick_a_head(-(_231),_234,_235),[++pick_a_head(-(_231),_234,_235)|_]).
member_62(++pick_a_head(-(_231),_234,_235),[_|A]) :-
     member_62(++pick_a_head(-(_231),_234,_235),A).
member_71(++not(+(_233),-(_231)),[++not(+(_233),-(_231))|_]).
member_71(++not(+(_233),-(_231)),[_|A]) :- member_71(++not(+(_233),-(_231)),A).
member_72(++not(-(_233),+(_231)),[++not(-(_233),+(_231))|_]).
member_72(++not(-(_233),+(_231)),[_|A]) :- member_72(++not(-(_233),+(_231)),A).
member_81(++infer(+(_231),_234),[++infer(+(_231),_234)|_]).
member_81(++infer(+(_231),_234),[_|A]) :- member_81(++infer(+(_231),_234),A).
member_82(++infer(-(_231),_234),[++infer(-(_231),_234)|_]).
member_82(++infer(-(_231),_234),[_|A]) :- member_82(++infer(-(_231),_234),A).
member_91(++anc(+(_231),_234),[++anc(+(_231),_234)|_]).
member_91(++anc(+(_231),_234),[_|A]) :- member_91(++anc(+(_231),_234),A).
member_92(++anc(-(_231),_234),[++anc(-(_231),_234)|_]).
member_92(++anc(-(_231),_234),[_|A]) :- member_92(++anc(-(_231),_234),A).
member_10(++inferall(_231,_232),[++inferall(_231,_232)|_]).
member_10(++inferall(_231,_232),[_|A]) :- member_10(++inferall(_231,_232),A).
member_11(--atom(+(_231)),[--atom(+(_231))|_]).
member_11(--atom(+(_231)),[_|A]) :- member_11(--atom(+(_231)),A).
member_12(--not_atom(-(_231)),[--not_atom(-(_231))|_]).
member_12(--not_atom(-(_231)),[_|A]) :- member_12(--not_atom(-(_231)),A).
member_13(--clause(_231),[--clause(_231)|_]).
member_13(--clause(_231),[_|A]) :- member_13(--clause(_231),A).
member_14(--body(_231),[--body(_231)|_]).
member_14(--body(_231),[_|A]) :- member_14(--body(_231),A).
member_15(--contr(_231,_232),[--contr(_231,_232)|_]).
member_15(--contr(_231,_232),[_|A]) :- member_15(--contr(_231,_232),A).
member_161(--pick_a_head(+(_231),_234,_235),[--pick_a_head(+(_231),_234,_235)|_]).
member_161(--pick_a_head(+(_231),_234,_235),[_|A]) :-
     member_161(--pick_a_head(+(_231),_234,_235),A).
member_162(--pick_a_head(-(_231),_234,_235),[--pick_a_head(-(_231),_234,_235)|_]).
member_162(--pick_a_head(-(_231),_234,_235),[_|A]) :-
     member_162(--pick_a_head(-(_231),_234,_235),A).
member_171(--not(+(_233),-(_231)),[--not(+(_233),-(_231))|_]).
member_171(--not(+(_233),-(_231)),[_|A]) :- member_171(--not(+(_233),-(_231)),A).
member_172(--not(-(_233),+(_231)),[--not(-(_233),+(_231))|_]).
member_172(--not(-(_233),+(_231)),[_|A]) :- member_172(--not(-(_233),+(_231)),A).
member_182(--infer(+(_231),_234),[--infer(+(_231),_234)|_]).
member_182(--infer(+(_231),_234),[_|A]) :- member_182(--infer(+(_231),_234),A).
member_182(--infer(-(_231),_234),[--infer(-(_231),_234)|_]).
member_182(--infer(-(_231),_234),[_|A]) :- member_182(--infer(-(_231),_234),A).
member_191(--anc(+(_231),_234),[--anc(+(_231),_234)|_]).
member_191(--anc(+(_231),_234),[_|A]) :- member_191(--anc(+(_231),_234),A).
member_192(--anc(-(_231),_234),[--anc(-(_231),_234)|_]).
member_192(--anc(-(_231),_234),[_|A]) :- member_192(--anc(-(_231),_234),A).
member_20(--inferall(_231,_232),[--inferall(_231,_232)|_]).
member_20(--inferall(_231,_232),[_|A]) :- member_20(--inferall(_231,_232),A).

prove_1(++atom(+(_178)),A) :-
```

```
     true.
prove_2(++not_atom(-(_178)),A) :-
     true.
prove_3(++clause(+(_300)),A) :-
     prove_1(++atom(+(_300)),[--clause(+(_300))|A]).
prove_3(++clause(or(+(_370),_191)),A) :-
     prove_1(++atom(+(_370)),[--clause(or(+(_370),_191))|A]),
     prove_4(++body(_191),[--clause(or(+(_370),_191))|A]).
prove_4(++body(-(_300)),A) :-
     prove_2(++not_atom(-(_300)),[--body(-(_300))|A]).
prove_4(++body(or(-(_370),_191)),A) :-
     prove_2(++not_atom(-(_370)),[--body(or(-(_370),_191))|A]),
     prove_4(++body(_191),[--body(or(-(_370),_191))|A]).
prove_5(++contr(+(_302),+(_302)),A) :-
     prove_1(++atom(+(_302)),[--contr(+(_302),+(_302))|A]).
prove_5(++contr(or(_201,_202),or(+(_401),_199)),A) :-
     prove_3(++clause(or(_201,_202)),[--contr(or(_201,_202),or(+(_401),_199))|A]),
     prove_61(++pick_a_head(+(_401),or(_201,_202),_199),
             [--contr(or(_201,_202),or(+(_401),_199))|A]).
prove_5(++contr(or(_201,_202),or(-(_401),_199)),A) :-
     prove_3(++clause(or(_201,_202)),[--contr(or(_201,_202),or(-(_401),_199))|A]),
     prove_62(++pick_a_head(-(_401),or(_201,_202),_199),
             [--contr(or(_201,_202),or(-(_401),_199))|A]).
prove_61(++pick_a_head(+(_201),or(+(_201),-(_309)),-(_309)),A) :-
     prove_2(++not_atom(-(_309)),[--pick_a_head(+(_201),or(+(_201),-(_309)),-(_309))|A]).
prove_62(++pick_a_head(-(_201),or(-(_201),-(_309)),-(_309)),A) :-
     prove_2(++not_atom(-(_309)),[--pick_a_head(-(_201),or(-(_201),-(_309)),-(_309))|A]).
prove_62(++pick_a_head(-(_201),or(_189,-(_201)),_189),A) :-
     prove_2(++not_atom(-(_201)),[--pick_a_head(-(_201),or(_189,-(_201)),_189)|A]).
prove_61(++pick_a_head(+(_201),or(+(_201),or(_181,_182)),or(_181,_182)),A) :-
     true.
prove_62(++pick_a_head(-(_201),or(-(_201),or(_181,_182)),or(_181,_182)),A) :-
     true.
prove_61(++pick_a_head(+(_212),or(_195,or(_192,_193)),or(_195,_190)),A) :-
     prove_61(++pick_a_head(+(_212),or(_192,_193),_190),
             [--pick_a_head(+(_212),or(_195,or(_192,_193)),or(_195,_190))|A]).
prove_62(++pick_a_head(-(_212),or(_195,or(_192,_193)),or(_195,_190)),A) :-
     prove_62(++pick_a_head(-(_212),or(_192,_193),_190),
             [--pick_a_head(-(_212),or(_195,or(_192,_193)),or(_195,_190))|A]).
prove_71(++not(+(_180),-(_180)),A) :-
     true.
prove_72(++not(-(_180),+(_180)),A) :-
     true.
prove_81(++infer(+(_198),_186),A) :-
     prove_91(++anc(+(_198),_186),[--infer(+(_198),_186)|A]).
prove_82(++infer(-(_198),_186),A) :-
     prove_92(++anc(-(_198),_186),[--infer(-(_198),_186)|A]).
prove_81(++infer(+(_210),_198),A) :-
     prove_3(++clause(_191),[--infer(+(_210),_198)|A]),
```

```
     prove_5(++contr(_191,+(_210)),[--infer(+(_210),_198)|A]),
     prove_1(++atom(+(_210)),[--infer(+(_210),_198)|A]).
prove_82(++infer(-(_210),_198),A) :-
     prove_3(++clause(_191),[--infer(-(_210),_198)|A]),
     prove_5(++contr(_191,-(_210)),[--infer(-(_210),_198)|A]),
     prove_2(++not_atom(-(_210)),[--infer(-(_210),_198)|A]).
prove_81(++infer(+(_224),_212),A) :-
     prove_3(++clause(_205),[--infer(+(_224),_212)|A]),
     prove_5(++contr(_205,or(+(_224),_196)),[--infer(+(_224),_212)|A]),
     prove_71(++not(+(_224),-(_533)),[--infer(+(_224),_212)|A]),
     prove_10(++inferall(_196,and(-(_533),_212)),[--infer(+(_224),_212)|A]).
prove_82(++infer(-(_224),_212),A) :-
     prove_3(++clause(_205),[--infer(-(_224),_212)|A]),
     prove_5(++contr(_205,or(-(_224),_196)),[--infer(-(_224),_212)|A]),
     prove_72(++not(-(_224),+(_533)),[--infer(-(_224),_212)|A]),
     prove_10(++inferall(_196,and(+(_533),_212)),[--infer(-(_224),_212)|A]).
prove_10(++inferall(-(_429),_199),A) :-
     prove_2(++not_atom(-(_429)),[--inferall(-(_429),_199)|A]),
     prove_72(++not(-(_429),+(_449)),[--inferall(-(_429),_199)|A]),
     prove_81(++infer(+(_449),_199),[--inferall(-(_429),_199)|A]).
prove_10(++inferall(+(_429),_199),A) :-
     prove_1(++atom(+(_429)),[--inferall(+(_429),_199)|A]),
     prove_71(++not(+(_429),-(_449)),[--inferall(+(_429),_199)|A]),
     prove_82(++infer(-(_449),_199),[--inferall(+(_429),_199)|A]).
prove_10(++inferall(or(+(_435),_200),_203),A) :-
     prove_71(++not(+(_435),-(_433)),[--inferall(or(+(_435),_200),_203)|A]),
     prove_82(++infer(-(_433),_203),[--inferall(or(+(_435),_200),_203)|A]),
     prove_10(++inferall(_200,_203),[--inferall(or(+(_435),_200),_203)|A]).
prove_10(++inferall(or(-(_435),_200),_203),A) :-
     prove_72(++not(-(_435),+(_433)),[--inferall(or(-(_435),_200),_203)|A]),
     prove_81(++infer(+(_433),_203),[--inferall(or(-(_435),_200),_203)|A]),
     prove_10(++inferall(_200,_203),[--inferall(or(-(_435),_200),_203)|A]).
prove_91(++anc(+(_194),and(+(_194),_179)),A) :-
     true.
prove_92(++anc(-(_194),and(-(_194),_179)),A) :-
     true.
prove_91(++anc(+(_207),and(+(_375),_192)),A) :-
     prove_1(++atom(+(_375)),[--anc(+(_207),and(+(_375),_192))|A]),
     prove_91(++anc(+(_207),_192),[--anc(+(_207),and(+(_375),_192))|A]).
prove_92(++anc(-(_207),and(+(_375),_192)),A) :-
     prove_1(++atom(+(_375)),[--anc(-(_207),and(+(_375),_192))|A]),
     prove_92(++anc(-(_207),_192),[--anc(-(_207),and(+(_375),_192))|A]).
prove_91(++anc(+(_207),and(-(_375),_192)),A) :-
     prove_2(++not_atom(-(_375)),[--anc(+(_207),and(-(_375),_192))|A]),
     prove_91(++anc(+(_207),_192),[--anc(+(_207),and(-(_375),_192))|A]).
prove_92(++anc(-(_207),and(-(_375),_192)),A) :-
     prove_2(++not_atom(-(_375)),[--anc(-(_207),and(-(_375),_192))|A]),
     prove_92(++anc(-(_207),_192),[--anc(-(_207),and(-(_375),_192))|A]).
prove_11(--atom(+(_213)),A) :-
```

```
     prove_13(--clause(+(_213)),[++atom(+(_213))|A]).
prove_11(--atom(+(_222)),A) :-
     prove_13(--clause(or(+(_222),_200)),[++atom(+(_222))|A]),
     prove_4(++body(_200),[++atom(+(_222))|A]).
prove_14(--body(_200),A) :-
     prove_13(--clause(or(+(_419),_200)),[++body(_200)|A]),
     prove_1(++atom(+(_419)),[++body(_200)|A]).
prove_12(--not_atom(-(_213)),A) :-
     prove_14(--body(-(_213)),[++not_atom(-(_213))|A]).
prove_12(--not_atom(-(_222)),A) :-
     prove_14(--body(or(-(_222),_200)),[++not_atom(-(_222))|A]),
     prove_4(++body(_200),[++not_atom(-(_222))|A]).
prove_14(--body(_200),A) :-
     prove_14(--body(or(-(_419),_200)),[++body(_200)|A]),
     prove_2(++not_atom(-(_419)),[++body(_200)|A]).
prove_11(--atom(+(_214)),A) :-
     prove_15(--contr(+(_214),+(_214)),[++atom(+(_214))|A]).
prove_13(--clause(or(_210,_211)),A) :-
     prove_15(--contr(or(_210,_211),or(+(_418),_208)),[++clause(or(_210,_211))|A]),
     prove_61(++pick_a_head(+(_418),or(_210,_211),_208),[++clause(or(_210,_211))|A]).
prove_13(--clause(or(_210,_211)),A) :-
     prove_15(--contr(or(_210,_211),or(-(_418),_208)),[++clause(or(_210,_211))|A]),
     prove_62(++pick_a_head(-(_418),or(_210,_211),_208),[++clause(or(_210,_211))|A]).
prove_161(--pick_a_head(+(_248),or(_210,_211),_208),A) :-
     prove_15(--contr(or(_210,_211),or(+(_248),_208)),
             [++pick_a_head(+(_248),or(_210,_211),_208)|A]),
     prove_3(++clause(or(_210,_211)),[++pick_a_head(+(_248),or(_210,_211),_208)|A]).
prove_162(--pick_a_head(-(_248),or(_210,_211),_208),A) :-
     prove_15(--contr(or(_210,_211),or(-(_248),_208)),
             [++pick_a_head(-(_248),or(_210,_211),_208)|A]),
     prove_3(++clause(or(_210,_211)),[++pick_a_head(-(_248),or(_210,_211),_208)|A]).
prove_12(--not_atom(-(_218)),A) :-
     prove_161(--pick_a_head(+(_324),or(+(_324),-(_218)),-(_218)),[++not_atom(-(_218))|A]).
prove_12(--not_atom(-(_218)),A) :-
     prove_162(--pick_a_head(-(_324),or(-(_324),-(_218)),-(_218)),[++not_atom(-(_218))|A]).
prove_12(--not_atom(-(_218)),A) :-
     prove_162(--pick_a_head(-(_218),or(_198,-(_218)),_198),[++not_atom(-(_218))|A]).
prove_161(--pick_a_head(+(_229),or(_201,_202),_199),A) :-
     prove_161(--pick_a_head(+(_229),or(_204,or(_201,_202)),or(_204,_199)),
             [++pick_a_head(+(_229),or(_201,_202),_199)|A]).
prove_162(--pick_a_head(-(_229),or(_201,_202),_199),A) :-
     prove_162(--pick_a_head(-(_229),or(_204,or(_201,_202)),or(_204,_199)),
             [++pick_a_head(-(_229),or(_201,_202),_199)|A]).
prove_191(--anc(+(_215),_195),A) :-
     prove_182(--infer(+(_215),_195),[++anc(+(_215),_195)|A]).
prove_192(--anc(-(_215),_195),A) :-
     prove_182(--infer(-(_215),_195),[++anc(-(_215),_195)|A]).
prove_13(--clause(_200),A) :-
     prove_182(--infer(+(_444),_207),[++clause(_200)|A]),
```

```
     prove_5(++contr(_200,+(_444)),[++clause(_200)|A]),
     prove_1(++atom(+(_444)),[++clause(_200)|A]).
prove_15(--contr(_200,+(_459)),A) :-
     prove_182(--infer(+(_459),_207),[++contr(_200,+(_459))|A]),
     prove_3(++clause(_200),[++contr(_200,+(_459))|A]),
     prove_1(++atom(+(_459)),[++contr(_200,+(_459))|A]).
prove_11(--atom(+(_255)),A) :-
     prove_182(--infer(+(_255),_207),[++atom(+(_255))|A]),
     prove_3(++clause(_200),[++atom(+(_255))|A]),
     prove_5(++contr(_200,+(_255)),[++atom(+(_255))|A]).
prove_13(--clause(_200),A) :-
     prove_182(--infer(-(_444),_207),[++clause(_200)|A]),
     prove_5(++contr(_200,-(_444)),[++clause(_200)|A]),
     prove_2(++not_atom(-(_444)),[++clause(_200)|A]).
prove_15(--contr(_200,-(_459)),A) :-
     prove_182(--infer(-(_459),_207),[++contr(_200,-(_459))|A]),
     prove_3(++clause(_200),[++contr(_200,-(_459))|A]),
     prove_2(++not_atom(-(_459)),[++contr(_200,-(_459))|A]).
prove_12(--not_atom(-(_255)),A) :-
     prove_182(--infer(-(_255),_207),[++not_atom(-(_255))|A]),
     prove_3(++clause(_200),[++not_atom(-(_255))|A]),
     prove_5(++contr(_200,-(_255)),[++not_atom(-(_255))|A]).
prove_13(--clause(_214),A) :-
     prove_182(--infer(+(_510),_221),[++clause(_214)|A]),
     prove_5(++contr(_214,or(+(_510),_205)),[++clause(_214)|A]),
     prove_71(++not(+(_510),-(_550)),[++clause(_214)|A]),
     prove_10(++inferall(_205,and(-(_550),_221)),[++clause(_214)|A]).
prove_13(--clause(_214),A) :-
     prove_182(--infer(-(_510),_221),[++clause(_214)|A]),
     prove_5(++contr(_214,or(-(_510),_205)),[++clause(_214)|A]),
     prove_72(++not(-(_510),+(_550)),[++clause(_214)|A]),
     prove_10(++inferall(_205,and(+(_550),_221)),[++clause(_214)|A]).
prove_15(--contr(_214,or(+(_525),_205)),A) :-
     prove_182(--infer(+(_525),_221),[++contr(_214,or(+(_525),_205))|A]),
     prove_3(++clause(_214),[++contr(_214,or(+(_525),_205))|A]),
     prove_71(++not(+(_525),-(_564)),[++contr(_214,or(+(_525),_205))|A]),
     prove_10(++inferall(_205,and(-(_564),_221)),[++contr(_214,or(+(_525),_205))|A]).
prove_15(--contr(_214,or(-(_525),_205)),A) :-
     prove_182(--infer(-(_525),_221),[++contr(_214,or(-(_525),_205))|A]),
     prove_3(++clause(_214),[++contr(_214,or(-(_525),_205))|A]),
     prove_72(++not(-(_525),+(_564)),[++contr(_214,or(-(_525),_205))|A]),
     prove_10(++inferall(_205,and(+(_564),_221)),[++contr(_214,or(-(_525),_205))|A]).
prove_171(--not(+(_271),-(_269)),A) :-
     prove_182(--infer(+(_271),_221),[++not(+(_271),-(_269))|A]),
     prove_3(++clause(_214),[++not(+(_271),-(_269))|A]),
     prove_5(++contr(_214,or(+(_271),_205)),[++not(+(_271),-(_269))|A]),
     prove_10(++inferall(_205,and(-(_269),_221)),[++not(+(_271),-(_269))|A]).
prove_172(--not(-(_271),+(_269)),A) :-
     prove_182(--infer(-(_271),_221),[++not(-(_271),+(_269))|A]),
```

```
     prove_3(++clause(_214),[++not(-(_271),+(_269))|A]),
     prove_5(++contr(_214,or(-(_271),_205)),[++not(-(_271),+(_269))|A]),
     prove_10(++inferall(_205,and(+(_269),_221)),[++not(-(_271),+(_269))|A]).
prove_20(--inferall(_205,and(-(_611),_221)),A) :-
     prove_182(--infer(+(_553),_221),[++inferall(_205,and(-(_611),_221))|A]),
     prove_3(++clause(_214),[++inferall(_205,and(-(_611),_221))|A]),
     prove_5(++contr(_214,or(+(_553),_205)),[++inferall(_205,and(-(_611),_221))|A]),
     prove_71(++not(+(_553),-(_611)),[++inferall(_205,and(-(_611),_221))|A]).
prove_20(--inferall(_205,and(+(_611),_221)),A) :-
     prove_182(--infer(-(_553),_221),[++inferall(_205,and(+(_611),_221))|A]),
     prove_3(++clause(_214),[++inferall(_205,and(+(_611),_221))|A]),
     prove_5(++contr(_214,or(-(_553),_205)),[++inferall(_205,and(+(_611),_221))|A]),
     prove_72(++not(-(_553),+(_611)),[++inferall(_205,and(+(_611),_221))|A]).
prove_12(--not_atom(-(_228)),A) :-
     prove_20(--inferall(-(_228),_208),[++not_atom(-(_228))|A]),
     prove_72(++not(-(_228),+(_466)),[++not_atom(-(_228))|A]),
     prove_81(++infer(+(_466),_208),[++not_atom(-(_228))|A]).
prove_172(--not(-(_244),+(_242)),A) :-
     prove_20(--inferall(-(_244),_208),[++not(-(_244),+(_242))|A]),
     prove_2(++not_atom(-(_244)),[++not(-(_244),+(_242))|A]),
     prove_81(++infer(+(_242),_208),[++not(-(_244),+(_242))|A]).
prove_182(--infer(+(_256),_208),A) :-
     prove_20(--inferall(-(_495),_208),[++infer(+(_256),_208)|A]),
     prove_2(++not_atom(-(_495)),[++infer(+(_256),_208)|A]),
     prove_72(++not(-(_495),+(_256)),[++infer(+(_256),_208)|A]).
prove_11(--atom(+(_228)),A) :-
     prove_20(--inferall(+(_228),_208),[++atom(+(_228))|A]),
     prove_71(++not(+(_228),-(_466)),[++atom(+(_228))|A]),
     prove_82(++infer(-(_466),_208),[++atom(+(_228))|A]).
prove_171(--not(+(_244),-(_242)),A) :-
     prove_20(--inferall(+(_244),_208),[++not(+(_244),-(_242))|A]),
     prove_1(++atom(+(_244)),[++not(+(_244),-(_242))|A]),
     prove_82(++infer(-(_242),_208),[++not(+(_244),-(_242))|A]).
prove_182(--infer(-(_256),_208),A) :-
     prove_20(--inferall(+(_495),_208),[++infer(-(_256),_208)|A]),
     prove_1(++atom(+(_495)),[++infer(-(_256),_208)|A]),
     prove_71(++not(+(_495),-(_256)),[++infer(-(_256),_208)|A]).
prove_171(--not(+(_234),-(_232)),A) :-
     prove_20(--inferall(or(+(_234),_209),_212),[++not(+(_234),-(_232))|A]),
     prove_82(++infer(-(_232),_212),[++not(+(_234),-(_232))|A]),
     prove_10(++inferall(_209,_212),[++not(+(_234),-(_232))|A]).
prove_172(--not(-(_234),+(_232)),A) :-
     prove_20(--inferall(or(-(_234),_209),_212),[++not(-(_234),+(_232))|A]),
     prove_81(++infer(+(_232),_212),[++not(-(_234),+(_232))|A]),
     prove_10(++inferall(_209,_212),[++not(-(_234),+(_232))|A]).
prove_182(--infer(+(_246),_212),A) :-
     prove_20(--inferall(or(-(_487),_209),_212),[++infer(+(_246),_212)|A]),
     prove_72(++not(-(_487),+(_246)),[++infer(+(_246),_212)|A]),
     prove_10(++inferall(_209,_212),[++infer(+(_246),_212)|A]).
```

```
prove_182(--infer(-(_246),_212),A) :-
     prove_20(--inferall(or(+(_487),_209),_212),[++infer(-(_246),_212)|A]),
     prove_71(++not(+(_487),-(_246)),[++infer(-(_246),_212)|A]),
     prove_10(++inferall(_209,_212),[++infer(-(_246),_212)|A]).
prove_20(--inferall(_209,_212),A) :-
     prove_20(--inferall(or(+(_499),_209),_212),[++inferall(_209,_212)|A]),
     prove_71(++not(+(_499),-(_497)),[++inferall(_209,_212)|A]),
     prove_82(++infer(-(_497),_212),[++inferall(_209,_212)|A]).
prove_20(--inferall(_209,_212),A) :-
     prove_20(--inferall(or(-(_499),_209),_212),[++inferall(_209,_212)|A]),
     prove_72(++not(-(_499),+(_497)),[++inferall(_209,_212)|A]),
     prove_81(++infer(+(_497),_212),[++inferall(_209,_212)|A]).
prove_11(--atom(+(_224)),A) :-
     prove_191(--anc(+(_391),and(+(_224),_201)),[++atom(+(_224))|A]),
     prove_91(++anc(+(_391),_201),[++atom(+(_224))|A]).
prove_11(--atom(+(_224)),A) :-
     prove_192(--anc(-(_391),and(+(_224),_201)),[++atom(+(_224))|A]),
     prove_92(++anc(-(_391),_201),[++atom(+(_224))|A]).
prove_191(--anc(+(_238),_201),A) :-
     prove_191(--anc(+(_238),and(+(_427),_201)),[++anc(+(_238),_201)|A]),
     prove_1(++atom(+(_427)),[++anc(+(_238),_201)|A]).
prove_192(--anc(-(_238),_201),A) :-
     prove_192(--anc(-(_238),and(+(_427),_201)),[++anc(-(_238),_201)|A]),
     prove_1(++atom(+(_427)),[++anc(-(_238),_201)|A]).
prove_12(--not_atom(-(_224)),A) :-
     prove_191(--anc(+(_391),and(-(_224),_201)),[++not_atom(-(_224))|A]),
     prove_91(++anc(+(_391),_201),[++not_atom(-(_224))|A]).
prove_12(--not_atom(-(_224)),A) :-
     prove_192(--anc(-(_391),and(-(_224),_201)),[++not_atom(-(_224))|A]),
     prove_92(++anc(-(_391),_201),[++not_atom(-(_224))|A]).
prove_191(--anc(+(_238),_201),A) :-
     prove_191(--anc(+(_238),and(-(_427),_201)),[++anc(+(_238),_201)|A]),
     prove_2(++not_atom(-(_427)),[++anc(+(_238),_201)|A]).
prove_192(--anc(-(_238),_201),A) :-
     prove_192(--anc(-(_238),and(-(_427),_201)),[++anc(-(_238),_201)|A]),
     prove_2(++not_atom(-(_427)),[++anc(-(_238),_201)|A]).
```

Appendix B

Theorem 5.3.1: Regular Approximation

```
% This regular approximation was generated by GdW from the program in Appendix A
% using a query-answer transformation based on a left-to-right and right-to-left
% computaion rule.
% We approximated with respect to the query <- solve(++infer(+(_),empty),[]).
% It took 141.3 seconds on a Sun Sparc-10 running SICStus Prolog to compute
% this regular approximation.

solve_ans(X1,X2) :-t10402(X1),t15(X2).
t10402(++X1) :-t10403(X1).
t15([]) :-true.
t10403(infer(X1,X2)) :-t10404(X1),t14(X2).
t10404(+(X1)) :-any(X1).
t14(empty) :-true.
prove_1_ans(X1,X2) :-t9334(X1),t19693(X2).
t9334(++X1) :-t9335(X1).
t19693([X1|X2]) :-t19694(X1),t19695(X2).
t9335(atom(X1)) :-t9336(X1).
t19694(--X1) :-t19697(X1).
t19695([X1|X2]) :-t19699(X1),t19695(X2).
t19695([]) :-true.
t9336(+(X1)) :-any(X1).
t19697(contr(X1,X2)) :-t15552(X1),t15553(X2).
t19697(infer(X1,X2)) :-t12549(X1),t6880(X2).
t19697(clause(X1)) :-t19470(X1).
t19699(--X1) :-t19702(X1).
t15552(+(X1)) :-any(X1).
t15553(+(X1)) :-any(X1).
t12549(+(X1)) :-any(X1).
t6880(empty) :-true.
```

```
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t19470(+(X1)) :-any(X1).
t19470(or(X1,X2)) :-t19462(X1),t19406(X2).
t19702(infer(X1,X2)) :-t8423(X1),t6880(X2).
t19702(contr(X1,X2)) :-t981(X1),t19704(X2).
t19702(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t2320(-(X1)) :-any(X1).
t19462(+(X1)) :-any(X1).
t19406(-(X1)) :-any(X1).
t19406(or(X1,X2)) :-t18979(X1),t19406(X2).
t8423(+(X1)) :-any(X1).
t981(or(X1,X2)) :-t747(X1),t685(X2).
t19704(or(X1,X2)) :-t983(X1),t15214(X2).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t18979(-(X1)) :-any(X1).
t747(+(X1)) :-any(X1).
t685(-(X1)) :-any(X1).
t685(or(X1,X2)) :-t549(X1),t685(X2).
t983(+(X1)) :-any(X1).
t15214(or(X1,X2)) :-t14989(X1),t15214(X2).
t15214(+(X1)) :-any(X1).
t15214(-(X1)) :-any(X1).
t549(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
t14989(+(X1)) :-any(X1).
t14989(-(X1)) :-any(X1).
prove_2_ans(X1,X2) :-t9308(X1),t19707(X2).
t9308(++X1) :-t9309(X1).
t19707([X1|X2]) :-t19708(X1),t19709(X2).
t9309(not_atom(X1)) :-t9310(X1).
t19708(--X1) :-t19711(X1).
t19709([X1|X2]) :-t19713(X1),t19709(X2).
t19709([]) :-true.
t9310(-(X1)) :-any(X1).
t19711(infer(X1,X2)) :-t45(X1),t14(X2).
t19711(inferall(X1,X2)) :-t8572(X1),t8418(X2).
t19711(anc(X1,X2)) :-t7184(X1),t7185(X2).
t19711(pick_a_head(X1,X2,X3)) :-t2368(X1),t2369(X2),t2372(X3).
t19711(body(X1)) :-t19117(X1).
t19713(--X1) :-t19716(X1).
t45(+(X1)) :-any(X1).
t14(empty) :-true.
t8572(-(X1)) :-any(X1).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t7184(+(X1)) :-any(X1).
t7185(and(X1,X2)) :-t7186(X1),t6880(X2).
t2368(+(X1)) :-any(X1).
```

```
t2369(or(X1,X2)) :-t2370(X1),t2371(X2).
t2372(-(X1)) :-any(X1).
t19117(-(X1)) :-any(X1).
t19117(or(X1,X2)) :-t18979(X1),t19117(X2).
t19716(infer(X1,X2)) :-t8423(X1),t6880(X2).
t19716(contr(X1,X2)) :-t1619(X1),t19718(X2).
t19716(clause(X1)) :-t18001(X1).
t19716(body(X1)) :-t18276(X1).
t19716(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t19716(anc(X1,X2)) :-t3673(X1),t7390(X2).
t19716(pick_a_head(X1,X2,X3)) :-t1630(X1),t1891(X2),t15806(X3).
t8419(-(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t7186(-(X1)) :-any(X1).
t2370(+(X1)) :-any(X1).
t2371(-(X1)) :-any(X1).
t18979(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t1619(or(X1,X2)) :-t747(X1),t685(X2).
t19718(or(X1,X2)) :-t1621(X1),t18477(X2).
t18001(or(X1,X2)) :-t18002(X1),t18157(X2).
t18276(or(X1,X2)) :-t18277(X1),t18157(X2).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t3673(+(X1)) :-any(X1).
t7390(and(X1,X2)) :-t7391(X1),t6880(X2).
t1630(+(X1)) :-any(X1).
t1891(or(X1,X2)) :-t1646(X1),t1893(X2).
t15806(or(X1,X2)) :-t14989(X1),t18477(X2).
t2320(-(X1)) :-any(X1).
t747(+(X1)) :-any(X1).
t685(-(X1)) :-any(X1).
t685(or(X1,X2)) :-t549(X1),t685(X2).
t1621(+(X1)) :-any(X1).
t18477(or(X1,X2)) :-t18477(X1),t18477(X2).
t18477(+(X1)) :-any(X1).
t18477(-(X1)) :-any(X1).
t18002(+(X1)) :-any(X1).
t18157(or(X1,X2)) :-t18157(X1),t18157(X2).
t18157(+(X1)) :-any(X1).
t18157(-(X1)) :-any(X1).
t18277(-(X1)) :-any(X1).
t549(-(X1)) :-any(X1).
t7391(-(X1)) :-any(X1).
t1646(-(X1)) :-any(X1).
t1646(+(X1)) :-any(X1).
t1893(or(X1,X2)) :-t549(X1),t685(X2).
t1893(-(X1)) :-any(X1).
```

```
t14989(+(X1)) :-any(X1).
t14989(-(X1)) :-any(X1).
prove_3_ans(X1,X2) :-t9474(X1),t19721(X2).
t9474(++X1) :-t9476(X1).
t19721([X1|X2]) :-t19722(X1),t8965(X2).
t9476(clause(X1)) :-t9478(X1).
t19722(--X1) :-t19724(X1).
t8965([]) :-true.
t8965([X1|X2]) :-t8966(X1),t8965(X2).
t9478(+(X1)) :-any(X1).
t9478(or(X1,X2)) :-t9472(X1),t7786(X2).
t19724(infer(X1,X2)) :-t6975(X1),t6880(X2).
t19724(contr(X1,X2)) :-t6966(X1),t19726(X2).
t8966(--X1) :-t8968(X1).
t9472(+(X1)) :-any(X1).
t7786(-(X1)) :-any(X1).
t7786(or(X1,X2)) :-t549(X1),t7786(X2).
t6975(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t6966(or(X1,X2)) :-t5229(X1),t4957(X2).
t19726(or(X1,X2)) :-t6968(X1),t15214(X2).
t8968(infer(X1,X2)) :-t8423(X1),t6880(X2).
t8968(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t549(-(X1)) :-any(X1).
t2320(-(X1)) :-any(X1).
t5229(+(X1)) :-any(X1).
t4957(-(X1)) :-any(X1).
t4957(or(X1,X2)) :-t549(X1),t4957(X2).
t6968(+(X1)) :-any(X1).
t15214(or(X1,X2)) :-t14989(X1),t15214(X2).
t15214(+(X1)) :-any(X1).
t15214(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t14989(+(X1)) :-any(X1).
t14989(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
prove_4_ans(X1,X2) :-t9653(X1),t19729(X2).
t9653(++X1) :-t9655(X1).
t19729([X1|X2]) :-t19730(X1),t19731(X2).
t9655(body(X1)) :-t9789(X1).
t19730(--X1) :-t19733(X1).
t19731([X1|X2]) :-t19735(X1),t19731(X2).
t19731([]) :-true.
t9789(-(X1)) :-any(X1).
t9789(or(X1,X2)) :-t549(X1),t9789(X2).
```

```
t19733(infer(X1,X2)) :-t14167(X1),t14(X2).
t19733(clause(X1)) :-t18001(X1).
t19733(body(X1)) :-t18276(X1).
t19735(--X1) :-t19738(X1).
t549(-(X1)) :-any(X1).
t14167(+(X1)) :-any(X1).
t14(empty) :-true.
t18001(or(X1,X2)) :-t18002(X1),t18157(X2).
t18276(or(X1,X2)) :-t18277(X1),t18157(X2).
t19738(infer(X1,X2)) :-t8423(X1),t6880(X2).
t19738(contr(X1,X2)) :-t6966(X1),t19740(X2).
t19738(clause(X1)) :-t18001(X1).
t19738(body(X1)) :-t18276(X1).
t19738(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t18002(+(X1)) :-any(X1).
t18157(or(X1,X2)) :-t18157(X1),t18157(X2).
t18157(+(X1)) :-any(X1).
t18157(-(X1)) :-any(X1).
t18277(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t6966(or(X1,X2)) :-t5229(X1),t4957(X2).
t19740(or(X1,X2)) :-t6968(X1),t15214(X2).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t2320(-(X1)) :-any(X1).
t5229(+(X1)) :-any(X1).
t4957(-(X1)) :-any(X1).
t4957(or(X1,X2)) :-t549(X1),t4957(X2).
t6968(+(X1)) :-any(X1).
t15214(or(X1,X2)) :-t14989(X1),t15214(X2).
t15214(+(X1)) :-any(X1).
t15214(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
t14989(+(X1)) :-any(X1).
t14989(-(X1)) :-any(X1).
prove_5_ans(X1,X2) :-t9861(X1),t9566(X2).
t9861(++X1) :-t9863(X1).
t9566([X1|X2]) :-t9567(X1),t8965(X2).
t9863(contr(X1,X2)) :-t9865(X1),t9866(X2).
t9567(--X1) :-t9568(X1).
t8965([]) :-true.
t8965([X1|X2]) :-t8966(X1),t8965(X2).
t9865(+(X1)) :-any(X1).
t9865(or(X1,X2)) :-t9472(X1),t7786(X2).
t9866(+(X1)) :-any(X1).
t9866(or(X1,X2)) :-t9859(X1),t8487(X2).
```

```
t9568(infer(X1,X2)) :-t9569(X1),t6880(X2).
t8966(--X1) :-t8968(X1).
t9472(+(X1)) :-any(X1).
t7786(-(X1)) :-any(X1).
t7786(or(X1,X2)) :-t549(X1),t7786(X2).
t9859(+(X1)) :-any(X1).
t8487(-(X1)) :-any(X1).
t8487(or(X1,X2)) :-t549(X1),t8487(X2).
t9569(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t8968(infer(X1,X2)) :-t8423(X1),t6880(X2).
t8968(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t549(-(X1)) :-any(X1).
t2320(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t8419(-(X1)) :-any(X1).
prove_61_ans(X1,X2) :-t9929(X1),t19744(X2).
t9929(++X1) :-t9931(X1).
t19744([X1|X2]) :-t19745(X1),t19746(X2).
t9931(pick_a_head(X1,X2,X3)) :-t9923(X1),t9933(X2),t9989(X3).
t19745(--X1) :-t19748(X1).
t19746([X1|X2]) :-t19750(X1),t19746(X2).
t19746([]) :-true.
t9923(+(X1)) :-any(X1).
t9933(or(X1,X2)) :-t9925(X1),t9990(X2).
t9989(-(X1)) :-any(X1).
t9989(or(X1,X2)) :-t549(X1),t9989(X2).
t19748(infer(X1,X2)) :-t703(X1),t14(X2).
t19748(contr(X1,X2)) :-t1619(X1),t15268(X2).
t19748(pick_a_head(X1,X2,X3)) :-t15692(X1),t1891(X2),t15695(X3).
t19750(--X1) :-t19753(X1).
t9925(+(X1)) :-any(X1).
t9990(-(X1)) :-any(X1).
t9990(or(X1,X2)) :-t549(X1),t9989(X2).
t549(-(X1)) :-any(X1).
t703(+(X1)) :-any(X1).
t14(empty) :-true.
t1619(or(X1,X2)) :-t747(X1),t685(X2).
t15268(or(X1,X2)) :-t15269(X1),t15214(X2).
t15692(+(X1)) :-any(X1).
t1891(or(X1,X2)) :-t1646(X1),t1893(X2).
t15695(or(X1,X2)) :-t14989(X1),t15214(X2).
t19753(infer(X1,X2)) :-t8423(X1),t6880(X2).
t19753(contr(X1,X2)) :-t1619(X1),t15268(X2).
t19753(pick_a_head(X1,X2,X3)) :-t15278(X1),t1891(X2),t15695(X3).
```

```
t19753(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t747(+(X1)) :-any(X1).
t685(-(X1)) :-any(X1).
t685(or(X1,X2)) :-t549(X1),t9989(X2).
t15269(+(X1)) :-any(X1).
t15214(or(X1,X2)) :-t14989(X1),t15214(X2).
t15214(+(X1)) :-any(X1).
t15214(-(X1)) :-any(X1).
t1646(-(X1)) :-any(X1).
t1646(+(X1)) :-any(X1).
t1893(or(X1,X2)) :-t549(X1),t685(X2).
t1893(-(X1)) :-any(X1).
t14989(+(X1)) :-any(X1).
t14989(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t15278(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t2320(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
prove_71_ans(X1,X2) :-t10025(X1),t9974(X2).
t10025(++X1) :-t10026(X1).
t9974([X1|X2]) :-t9975(X1),t8965(X2).
t10026(not(X1,X2)) :-t10027(X1),t10028(X2).
t9975(--X1) :-t9976(X1).
t8965([]) :-true.
t8965([X1|X2]) :-t8966(X1),t8965(X2).
t10027(+(X1)) :-any(X1).
t10028(-(X1)) :-any(X1).
t9976(infer(X1,X2)) :-t9977(X1),t6880(X2).
t8966(--X1) :-t8968(X1).
t9977(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t8968(infer(X1,X2)) :-t8423(X1),t6880(X2).
t8968(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t2320(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t549(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
prove_72_ans(X1,X2) :-t10304(X1),t10167(X2).
t10304(++X1) :-t10305(X1).
t10167([X1|X2]) :-t8611(X1),t10206(X2).
```

```
t10305(not(X1,X2)) :-t10306(X1),t10307(X2).
t8611(--X1) :-t8614(X1).
t10206([X1|X2]) :-t10171(X1),t10206(X2).
t10206([]) :-true.
t10306(-(X1)) :-any(X1).
t10307(+(X1)) :-any(X1).
t8614(infer(X1,X2)) :-t2124(X1),t14(X2).
t8614(inferall(X1,X2)) :-t8658(X1),t8418(X2).
t10171(--X1) :-t10173(X1).
t2124(+(X1)) :-any(X1).
t14(empty) :-true.
t8658(-(X1)) :-any(X1).
t8658(or(X1,X2)) :-t549(X1),t8658(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t10173(inferall(X1,X2)) :-t5905(X1),t8418(X2).
t10173(infer(X1,X2)) :-t8423(X1),t6880(X2).
t549(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t5905(-(X1)) :-any(X1).
t5905(or(X1,X2)) :-t549(X1),t5905(X2).
t8423(+(X1)) :-any(X1).
t2320(-(X1)) :-any(X1).
prove_81_ans(X1,X2) :-t9980(X1),t8965(X2).
t9980(++X1) :-t9981(X1).
t8965([]) :-true.
t8965([X1|X2]) :-t8966(X1),t8965(X2).
t9981(infer(X1,X2)) :-t9982(X1),t6880(X2).
t8966(--X1) :-t8968(X1).
t9982(+(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t8968(infer(X1,X2)) :-t8423(X1),t6880(X2).
t8968(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t2320(-(X1)) :-any(X1).
t8423(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t549(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
prove_10_ans(X1,X2) :-t10384(X1),t10118(X2).
t10384(++X1) :-t10386(X1).
t10118([X1|X2]) :-t10120(X1),t8965(X2).
t10386(inferall(X1,X2)) :-t10391(X1),t8418(X2).
t10120(--X1) :-t10122(X1).
t8965([]) :-true.
t8965([X1|X2]) :-t8966(X1),t8965(X2).
```

```
t10391(-(X1)) :-any(X1).
t10391(or(X1,X2)) :-t549(X1),t10391(X2).
t8418(and(X1,X2)) :-t8419(X1),t6880(X2).
t10122(infer(X1,X2)) :-t10103(X1),t6880(X2).
t10122(inferall(X1,X2)) :-t10115(X1),t8418(X2).
t8966(--X1) :-t8968(X1).
t549(-(X1)) :-any(X1).
t8419(-(X1)) :-any(X1).
t6880(empty) :-true.
t6880(and(X1,X2)) :-t2320(X1),t6880(X2).
t10103(+(X1)) :-any(X1).
t10115(or(X1,X2)) :-t10116(X1),t6159(X2).
t8968(infer(X1,X2)) :-t8423(X1),t6880(X2).
t8968(inferall(X1,X2)) :-t6881(X1),t8418(X2).
t2320(-(X1)) :-any(X1).
t10116(-(X1)) :-any(X1).
t6159(-(X1)) :-any(X1).
t6159(or(X1,X2)) :-t549(X1),t6159(X2).
t8423(+(X1)) :-any(X1).
t6881(-(X1)) :-any(X1).
t6881(or(X1,X2)) :-t549(X1),t6881(X2).
```

Bibliography

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Anonymous. The QED Manifesto. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 238–240. Springer-Verlag, 1994. The authors of this document chose to remain anonymous.

[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[5] P. Baumgartner and D.A. de Waal. Experiments with problems from the TPTP problem library. Experiments were carried out to determine the effect of deleting clauses from first-order theories that are not needed in proofs, 1994.

[6] P. Baumgartner and U. Furbach. Model elimination without contrapositives. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 87–101. Springer-Verlag, 1994.

[7] P. Baumgartner and U. Furbach. PROTEIN: A PROver with a Theory Extension INterface. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 769–773. Springer-Verlag, 1994.

[8] B. Beckert and R. Hähnle. An improved method for adding equality to free variable semantic tableaux. In *Proceedings—CADE 11*. Springer-Verlag, 1992.

[9] B. Beckert and J. Possega. leanTAP: Lean tableau-based theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 791–797. Springer-Verlag, 1994.

[10] K. Benkerimi and J.W. Lloyd. A partial evaluation procedure for logic programs. In S.K. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming,* Austin, pages 343–358. MIT Press, 1990.

[11] W. Bibel. *Automated Theorem Proving.* Vieweg, second edition, 1987.

[12] W. Bibel, S Brüning, U. Egly, and T. Rath. KoMeT. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 783–787. Springer-Verlag, 1994.

[13] W. Bibel, S. Hölldobler, and J. Würtz. Cycle unification. In *Proceedings of the Conference on Automated Deduction*, pages 94—108. Springer-Verlag, 1992.

[14] S. Brüning. Detecting non-provable goals. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 222—236. Springer-Verlag, 1994.

[15] S. Brüning and D.A. de Waal. Comparing regular approximations and monadic clause sets. The results of approximating interesting examples using regular approximations and monadic clause sets were compared, 1994.

[16] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1990.

[17] C. Chang and R.C.-T Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[18] A. Church. An unsolvable problem of number theory. *Amer. J. Math.*, 58:345–363, 1936.

[19] M. Codish. *Abstract Interpretation of Sequential and Concurrent Logic Programs.* PhD thesis, The Weizmann Institute of Science, 1991.

[20] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[21] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, pages 201–215, 1960.

[22] D. De Schreye and M. Bruynooghe. The compilation of forward checking regimes through meta-interpretations and transformation. In *Meta-Programming in Logic Programming*, pages 217–231. The MIT Press, 1989.

[23] D.A. de Waal. The power of partial evaluation. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 113–123. Springer-Verlag, 1994.

[24] D.A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 205–221. Springer-Verlag, 1992.

[25] D.A. de Waal and J. Gallagher. Logic program specialisation with deletion of useless clauses (poster abstract). In D. Miller, editor, *Proceedings of the 1993 Logic programming Symposium, Vancouver*, page 632. MIT Press, (full version appears as CSTR-92-33, Dept. Computer Science, University of Bristol), 1993.

[26] D.A. de Waal and J.P. Gallagher. Specialising a theorem prover. Technical Report CSTR-92-33, University of Bristol, November 1992.

[27] D.A. de Waal and J.P. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

[28] S. Debray and R. Ramakrishnan. Canonical computations of logic programs. Technical report, University of Arizona-Tucson, July 1990.

[29] S.K. Debray. Static analysis of logic programs, (advanced tutorial). In *Proceedings of ILPS'93*, 1993.

[30] F. Defoort. De grondrepresentatie in Prolog en in Gödel. Master's thesis, Katholieke Universiteit Leuven, 1991.

[31] A.P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Descriptions of Programming Languages*, pages 390–421. North-Holland, 1978.

[32] C. Fermuller, A. Leitsch, T. Tammet, and N. Zamov. Resolution methods for the decision problem. In *LNAI 679*. Springer-Verlag, 1993.

[33] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[34] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *6th IEEE Symposium on Logic in Computer Science, Amsterdam*, 1991.

[35] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[36] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[37] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages

732–746, Jerusalem, June 1990. MIT Press. Revised version in *New Generation Computing*, 9(3&4):305–334.

[38] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic, Leuven, Belgium*, 1990.

[39] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[40] J. Gallagher and D.A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, University of Bristol, March 1992.

[41] J. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. Technical Report CSTR-93-19, University of Bristol, November 1993.

[42] J. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[43] P.C. Gillmore. A program for the production of proofs for theorems derivable within the first order predicate calculus from axioms. In *Proc. Intern. Conf. on Infor. Processing*, 1959.

[44] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57, 1992.

[45] C. Goller, R Letz, K. Mayr, and J Schumann. SETHEO V3.2: Recent developments. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 778—782. Springer-Verlag, 1994.

[46] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, University of Bristol, 1993.

[47] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco*, pages 197–209. ACM Press, 1990.

[48] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. Technical report, Brown University, Department of Computer Science, December 1993.

[49] J. Herbrand. Recherches sur la theorie de la demonstration. In *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III sciences mathematiques et physiques, No 33*, 1930.

[50] M. Hermenegildo. Abstract interpretation and its applications (advanced tutorial). In *Proceedings of JICSLP'92*, 1992.

[51] P. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[52] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In *Meta-Programming in Logic Programming*, pages 23–52. The MIT Press, 1989.

[53] N. Jones. Flow analysis of lazy higher order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.

[54] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program generation*. Prentice Hall, 1993.

[55] T. Kanamori. Abstract interpretation based on Alexander templates. Technical Report TR-549, ICOT, March 1990.

[56] H. Kautz and B. Selman. Forming concepts for fast inference. In *Proceedings of the Tenth National Conference on Artificial Intelligence, San Jose, California*. AAAI/MIT Press, 1992.

[57] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[58] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In Leech, editor, *Combinatorial Problems In Abstract Algebras*, pages 263–270, 1970.

[59] J. Komorowski. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1981.

[60] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(1), 1971.

[61] P. Kursawe. Pure partial evaluation and instantiation. In D. Bjørner, A.P Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[62] M. Leuschel. Partial evaluation of the real thing. In *Pre-Proceedings of LOPSTR94*. University of Piza, 1994.

[63] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3(1):225–240, 1984.

[64] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[65] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[66] L.A. Lombardi. Incremental computation. *Advances in Computers*, 8:247–333, 1967.

[67] D.W. Loveland. *Automated theorem proving: a logical basis*. North-Holland, 1978.

[68] D.W. Loveland. Automated-theorem proving: A quarter-century review. *Contemporary Mathematics*, 29:1–46, 1983.

[69] D.W. Loveland. Near-Horn Prolog and beyond. *Journal of Automated Reasoning*, 7(1):1–26, 1991.

[70] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. Lusk and R Overbeek, editors, *9th International Conference on Automated Deduction*, pages 415–434. Springer-Verlag, 1988.

[71] K. Marriott, L. Naish, and J-L. Lassez. Most specific logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington*, August 1988.

[72] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of logic programs. *Journal of Logic Programming*, 13:181–204, 1992.

[73] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Evaluation in Logic Programming*. PhD thesis, Katholieke Universiteit Leuven, 1994.

[74] K. Mayr. Refinements and extensions of model elimination. In A. Voronkov, editor, *Logic Programming and Automated Reasoning—LPAR'93*, pages 217–228. Springer-Verlag, 1993.

[75] C.S. Mellish. Using specialisation to reconstruct two mode inference systems. Technical report, University of Edinburgh, March 1990.

[76] J. Minker and A. Rajasekar. A fixpoint semantics for disjunctive logic programs. *Journal of Logic Programming*, 9:45–74, 1990.

[77] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, 1984.

[78] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 214–227. Springer-Verlag, 1993.

[79] L. Naish. Types and the intended meaning of logic programs. Technical report, University of Melbourne, 1990.

[80] G. Neugebauer and D.A. de Waal. An anatomy of partial evaluation. The similarities and differences between operations used in partial evaluation and automated theorem proving were studied, 1994.

[81] A. Newell, J.C. Shaw, and H.A. Simon. Emperical explorations of the logic theory machine: a case study in heuristics. In *Proc. Western Joint Computer Conference*, pages 218–239. McGraw-Hill, 1956.

[82] F.J. Pelletier. Seventy-five problems for testing automated theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[83] D. Plaisted. Abstraction mappings in mechanical theorem proving. In W. Bibel and R. Kowalski, editors, *Automated Deduction—CADE-5*, pages 264–280. Springer-Verlag, 1980.

[84] D. Plaisted. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4:287–325, 1988.

[85] D.L. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 635–641. Lecture Notes in Computer Science, Springer-Verlag, 1986.

[86] J. Possega. Compiling proof search in semantic tableaux. In J. Komorowski and Z.W. Raś, editors, *Methodologies for Intelligent Systems*, pages 39–48. Springer-verlag, 1993.

[87] D. Prawitz. An improved proof procedure. *Theoria*, pages 102–139, 1960.

[88] S. Prestwich. Online partial deduction of large programs. In *Proceedings PEPM'93*, pages 111–118, Copenhagen, Denmark, June 1993. ACM.

[89] D.W. Reed and D.W. Loveland. A comparison of three Prolog extensions. *Journal of Logic Programming*, 12(1):25–50, 1992.

[90] D.W. Reed, D.W. Loveland, and T. Smith. An alternative characterization of disjunctive logic programs. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 54–68. MIT Press, 1991.

[91] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Jour. Assoc. for Comput. Mach.*, pages 23–41, 1965.

[92] J. Rohmer, R. Lescœr, and J.-M. Kerisit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4, 1986.

[93] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology, 1991.

[94] C. Samaka and H. Seki. Partial deduction of disjunctive logic programs: A declarative approach. In *Pre-Proceedings of LOPSTR94.* University of Piza, 1994.

[95] H. Seki. On the power of Alexander templates. In *Proceedings of the 8^{th} ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, Pennsylvania*, 1989.

[96] B. Selman and H. Kautz. Knowledge compilation using Horn approximation. In *Proceedings of the Ninth National Conference on Artificial Intelligence.* AAAI/MIT Press, 1991.

[97] J. Slaney, E. Lusk, and W. McCune. SCOTT: Semantically constrained Otter system description. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 764–768. Springer-Verlag, 1994.

[98] R. Smullyan. *First-Order Logic.* Springer-Verlag, 1968.

[99] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[100] M.E. Stickel. A Prolog Technology Theorem Prover. In *International Symposium on Logic Programming,* Atlantic City, NJ, pages 211–217, Feb. 6-9 1984.

[101] M.E. Stickel. Schubert's Steamroller Problem. *Journal of Automated Reasoning*, 2(1):89–101, 1986.

[102] F. Stolzenburg and D.A. de Waal. Eliminating ancestor resolution from model elimination. The influence of the removal of ancestor resolution from model elimination proofs were investigated, 1994.

[103] G. Sutcliffe. Linear-input subset analysis. In D. Kapur, editor, *11th International Conference on Automated Deduction*, pages 268–280. Lecture Notes in Computer Science, Springer-Verlag, 1992.

[104] C. Sutter, G. Sutcliffe, and T. Yemenis. The TPTP Problem Library. Technical Report TPTP v1.0.0 - TR 12.11.93, James Cook University, Australia, November 1993.

[105] A.M. Turing. On computable numbers, with an application to the entscheindungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936.

[106] F. van Harmelen. The limitations of partial evaluation. In J. Dassow and J. Kelemen, editors, *5th International Meeting of Young Computer Scientists*, pages 170–187. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[107] T. Wakayama. CASE-Free programs: An abstraction of definite Horn programs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 87–101. Lecture Notes in Computer Science, Springer-Verlag, 1990.

[108] H. Wang. Toward mechanical mathematics. *IBM Jour. Research and Devel.*, pages 2–22, 1960.

[109] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[110] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.

[111] J. Zobel. Derivation of polymorphic types for Prolog programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*. MIT Press, 1988.