The Applicability of Logic Program Analysis and Transformation to Theorem Proving [1]

D.A. de Waal   J. Gallagher

September 1993

Department of Computer Science
University of Bristol
Queen's Building
University Walk
Bristol BS8 1TR
U.K.

e-mail: andre@compsci.bristol.ac.uk, john@compsci.bristol.ac.uk

[1]Work supported by ESPRIT Project PRINCE (5246)

Abstract

Analysis and transformation techniques developed for logic programming can be successfully applied to automatic theorem proving. In this paper we demonstrate how these techniques can be used to infer useful information that can speed up theorem provers, assist in the identification of necessary inference rules for solving specific problems, how failure branches can be eliminated from the proof tree and how a non-terminating deduction in a proof system can be turned into failure. In addition, this method also provides sufficient conditions for identifying Case-free Theories [26]. The specialisation techniques developed in this paper are independent of the proof system and can therefore be applied to theorem provers for any logic written as logic programs.

1 Introduction

Analysis and transformation techniques developed for logic programming can be successfully applied to automatic theorem proving. In this paper we demonstrate how these techniques can be used to infer nontrivial results. We show how failure branches (possibly non-terminating) can be eliminated from the proof tree and how a non-terminating deduction in a proof system can be turned into failure.

Our motivation for this work is the following: we want to speed up theorem provers by restricting inference rules to necessary ones needed for proving a given theorem and avoiding sentences in the theory that do not contribute to its proof. The first Futamura projection [5], defining compilation, is a well-known transformation technique based on partial evaluation [20]. It provides an adequate framework in which this problem can be expressed elegantly and can be briefly explained as follows.

An interpreter for some language L is a meta-program taking a program in L as data. The result of restricting the interpreter to a particular object program P_0 is a specialised interpreter that can only interpret one program P_0. The specialised interpreter may be regarded as a compiled version of P_0. More specifically, when the interpreter is a theorem prover for some language (logic) L and the object program P_0 is a object theory in L, the result of restricting the theorem prover to a particular object theory is a specialised theorem prover that can only prove theorems in the given object theory. The specialised theorem prover is the result of compiling the theory P_0 into the language in which the theorem prover is written.

If the meta-language is expressive enough such that any theorem prover (or more conservatively, a large number of theorem provers) can be written using this language, we may regard it as a specification language and state different theorem provers as sets of inference rules in it. Combining partial evaluation with state of the art analysis techniques may now give us a way of detecting inference rules and object clauses that are unnecessary for proving a given theorem given some object theory and theorem prover. This information may be used to prune *useless* derivations or attempted derivation and enable the specialised prover to prove theorems considerably faster that was possible with the original theorem prover and just partial evaluation. The following diagrammatically illustrates the process:

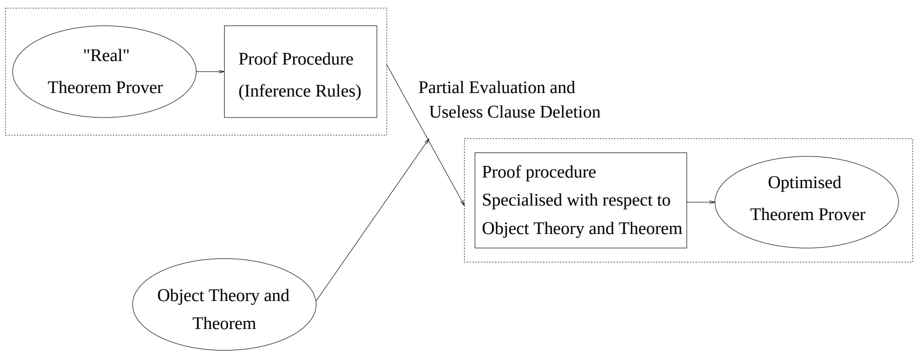
"Real" Theorem Prover
Proof Procedure (Inference Rules)
Partial Evaluation and Useless Clause Deletion
Proof procedure Specialised with respect to Object Theory and Theorem
Optimised Theorem Prover
Object Theory and Theorem

Figure 1: Specialisation of Theorem Provers

In the above figure, the "real" theorem prover and the proof procedure stated as inference rules together with the procedure for generating proofs may be the same (e.g. a first order predicate logic theorem prover written as a logic program). Alternatively, the "real" theorem prover might be implemented in some fast procedural language and the proof procedure may be regarded as a specification of the theorem prover. In both cases, the results of applying the method described in this paper may then be used to optimise the "specifications" and/or the original theorem prover. For instance, if a certain inference rule is detected as being unused in the proof of a certain theorem, this information could be applied in the implementation of the "real" theorem prover. As we will show in this paper, this method may lead to a significant reduction in the search space of the theorem prover and the time taken to prove a given theorem.

Two first order clausal theorem provers are analysed. The first is a clausal theorem prover by Poole and Goebel [22]. This prover is based on the model elimination proof procedure by Loveland [15]. The second theorem prover is the nH-Prolog proof system by Loveland [16] developed to solve efficiently near-Horn or almost-Horn problems [16] (although it handles any clausal theory).

Our analysis method depends heavily on techniques developed to analyse and transform logic programs. The theories of abstract interpretation and partial evaluation provide us with the necessary techniques needed for our analysis. A two step analysis method is developed and applied to the two proof procedures and nontrivial results inferred.

As this is a bridging paper that tries to show the relevance of logic program analysis and transformation techniques to the field of automatic theorem proving, we do not assume any specialist knowledge, but it will be helpful if the reader is familiar with the basic ideas and terminology of logic programming and automatic theorem proving as presented in [13] and [15].

In the next section the logic programming analysis techniques needed are described and regular unary logic programs introduced by means of an example. In Section 3 a two stage analysis method is developed that consists of a partial evaluation and an approximation phase. In Section 4 the Poole-Goebel clausal theorem prover is described and we show how this prover can be specialised with respect to some object theory. The Naive nH-Prolog proof system is then described we show how nontermination of this proof system can be turned into failure. In Section 6 we give some performance results. The paper ends with a short discussion and conclusions.

2 Approximation of Logic Programs

One of the main ideas in our method is the use of *approximations* to analyse a program. Problem-solving by approximation is a very general principle and we discuss some related work in Section 3.3. We now define the concept of a safe approximation of a logic program.

Definition 2.1 *safe approximation*

Let P and P' be normal programs. Then P' is a safe approximation of P if for all definite goals $\leftarrow G$,

- *if $P' \cup \{\leftarrow G\}$ has a finitely failed SLDNF-tree then $P \cup \{\leftarrow G\}$ 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'. So a stronger and more intuitive condition, which is usually used when constructing approximations, is as follows:

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

Approximations satisfying this condition can be used to establish properties of the set of successful goals in P. However for our purposes we are interested in detecting non-successful goals in P (see Proposition 3.4), and for this the condition of Definition 2.1 is adequate.

3 Useless Clauses

Given a logic program it is desirable to detect clauses that yield no solutions. This need has arisen mainly in the context of program development, where the detection of such a clause could indicate a bug, since that clause contributes nothing to the program's computation. Such a clause might be considered "badly-typed". Related topics are discussed in [17], [29], [19] and [30]. Normally a programmer would try not to write useless clauses and so their detection is regarded as debugging. However useless clauses arise more frequently in programs that are generated by transforming some other program. In such cases removal of useless clauses is an important part of the program construction process.

The following definitions are quoted from [3].

Definition 3.1 *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 3.2 *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 associated with T.

Obviously a clause that is useless by Definition 3.1 is also useless with respect to any goal by Definition 3.2.

3.1 Using Approximation to Find Useless Clauses

It is undecidable whether an arbitrary clause yields no solutions with respect to the computation of a goal. However, use of an approximation provides sufficient conditions to detect such useless clauses. The following definition and proposition shows how this can be done.

Definition 3.3 *Let G be a normal goal. Then G^+ denotes the definite goal obtained by deleting all negative literals from G.*

Proposition 3.4 *safe approximation for detecting useless clause*

Let P and P' be normal programs, where P' is 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 2.1 $P \cup \{\leftarrow B^+\}$ 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 it is useless in P. □

The use of approximations can thus provide us with ways of detecting useless clauses, if we can construct approximations in which finite failure is decidable. In order to delete clauses that are useless with respect to a computation, any abstract interpretation framework for logic programming could in principle be used, provided that it gives some approximate description, for each clause in the program, of the set of instances of the clauses with which it is called, and successful instances of the clause. If the analysis shows that a clause never returns any results (in a computation), then it is useless (with respect to that computation). We quote the frameworks described in [27], [1] and [3] as examples of appropriate methods. Note that a fixed computation rule (usually left-to-right) is usually assumed for abstract interpretation.

3.2 Regular Approximations

The safe approximations used in this paper are Regular Unary Logic (RUL) programs, defined in [29]. For any program P, we construct an RUL program that is a safe approximation of P [8]. This can then be used to detect useless clauses, since finite failure of definite goals is decidable in an RUL program.

A *regular unary clause* is of the form $p(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$ where $x_1, \ldots, x_n$ are distinct variables. An *RUL program* is a set of regular unary clauses in which no two clause heads have a common instance. It is decidable whether a given goal succeeds in an RUL program. A goal does not succeed if and only if it fails finitely.

A detailed description of a method for computing a regular approximation of a given normal program can be found in [7], and the method is also summarised in [8]. Briefly, the method is based on abstract interpretation of the standard fixpoint semantics of a definite program P, given by its T_P operator. An abstract version of the T_P operator is defined, more specifically, a monotonic operator that maps one RUL program to another. Its least fixed point is computed as the limit of an ascending sequence, and it can easily be shown that this represents a safe approximation of the least fixed point of T_P.

There are other ways of generating regular programs that are more expressive (and hence could give more precise approximations) but we chose this form because it is easy to manipulate and it is straightforward to check success or failure of derivations. Another rather direct method for finding a regular approximation of a logic program is given in [4]. The authors emphasise that there is a direct connection between their method and methods based on computing an abstraction of T_P. Given a program P, an approximation is first constructed by defining a unary "type" predicate for each program variable. The clauses defining these predicates can be written down directly from the program text, and a unary program results. This can then be converted to a regular unary program, but not restricted to RUL programs since variables may be repeated in clause bodies, and two clause heads may contain common instances.

In fact, that form could also be used as the approximation, and the only difference would be that a somewhat more complex decision procedure for checking useless clauses would be needed.

We have as yet not performed sufficient analysis on the complexity of our approximation algorithm to indicate upper bounds on complexity, or, more importantly, average complexity. Our approximations are in general somewhat less precise than those defined in [4], but this trades off with faster approximation times. (We confirmed this by implementing a version of the approximation procedure in [4]). Timings are given in a later section. An important point is that a program can be split into smaller sets of clauses each of which is separately approximated. Each set contains a group of mutually recursive predicates (a "strongly-connected component" of the program's dependency graph). Average programs rarely contain more than groups of two or three mutually recursive predicates.

To conclude this section, we give an example for the reader unfamiliar with approximation using regular programs. Consider the following naive reverse program.

$reverse([\,],[\,]) \leftarrow true$
$reverse([x|xs], ys) \leftarrow reverse(xs, zs), append(zs, [x], ys)$

$append([\,], ys, ys) \leftarrow true$
$append([x|xs], ys, [x|zs]) \leftarrow append(xs, ys, zs)$

Example 1: Naive reverse

A regular approximation of Example 1 is

$approx(reverse(x1, x2)) \leftarrow t1(x1), t2(x2)$

$t1([\,]) \leftarrow true$
$t1([x1|x2]) \leftarrow any(x1), t1(x2)$
$t2([x1|x2]) \leftarrow any(x1), t2(x2)$
$t2([\,]) \leftarrow true$

$approx(append(x1, x2, x3)) \leftarrow t3(x1), any(x2), any(x3)$

$t3([\,]) \leftarrow true$
$t3([x1|x2]) \leftarrow any(x1), t3(x2)$

Example 2: A regular approximation of Example 1

where $any(t)$ is true for any term t in the Herbrand Universe of the program. The predicates $t1$ to $t4$ are ordinary logic program predicates and $x1$, $x2$ and $x3$ logic program variables. The program in Example 2 was produced automatically by the logic program specialisation system SP [6] at the University of Bristol.

3.3 Approximation in Problem Solving

The use of approximation to solve problems is common. In its simplest form it means that when trying to show that some property $p(x)$ holds for every element x of a set S, it is sufficient to show that $p(x)$ holds for every element of any larger set T (called an *approximation* of S). In particular, one can show that some element a is not in S (that is, $p(x) \equiv x \neq a$) by showing that it is not in T. The usefulness of this principle depends on the construction of T together with the proof for T being simpler or faster than the proof for S.

An approximation of a logical theory S is some theory T such that T is a logical consequence of S. That is, the set of consequences of S is a subset of the consequences of T. Therefore if a formula does not follow from T then it does not follow from S.

In [23] and [12] Kautz and Selman use this idea 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 for theoremhood in the approximating theory before applying a general propositional theorem prover. Some non-theorems can be thrown out fast.

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

Obviously, when assessing the potential advantage of using approximations in solving a problem, the cost of constructing the approximation has to be considered. Worst cases for any interesting classes of approximation seem to be exponential in the size of the approximating theory and/or the time taken to compute the approximation. Three points should be made here:

- In our experiments average cases are quite tractable. Factors influencing exponential complexity are certainly not directly related to the size of the program, since large programs can be approximated in reasonable times.
- There are possible tradeoffs between precision and complexity, and a somewhat less precise approximation can often be obtained much faster.
- The cost of producing the approximation may be divided among all the theorems that can be checked using it, much as the time taken to compile a program is saved by running the program many times.

The approximation principle also underlies the field of abstract interpretation in program analysis (see for example [11], [27] and [1]), and has close links with type checking and other branches of program analysis.

4 Analysis Method

The analysis method consists of two stages. Let the theorem prover be a normal program P, and the theory and theorem be represented by a normal goal G.

1. Partially evaluate [5], [14], [6], [20] P wrt G (or some G' such that G is an instance of G'). Call the partially evaluated program P_1.

2. Compute a regular approximation of clauses in P_1 with respect to $P_1 \cup \{G\}$. This yields P_2 which is used to delete useless clauses from P_1, giving P_3.

By the correctness of partial evaluation [14], and by Proposition 3.4, 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
2. if $P \cup \{G\}$ has a finitely failed SLDNF-tree then $P_3 \cup \{G\}$ has a finitely failed SLDNF-tree.

The combined specialisation thus preserves all the results of terminating computations. Note that nothing is said about nonterminating computations. However, it is important to realise that the above method may turn a nonterminating computation into a terminating computation (an infinitely failed computation may be turned into finite failure and the specialised program P_3 may yield more answers than the original program P).

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 during partial evaluation, since clearly some calls to a procedure may fail or loop while other calls succeed, e.g. the negative ancestor check in the model elimination prover may succeed for some literals, but fail for others. In particular, our partial evaluation phase produces a separate version for each signed predicate and inference rule. The usefulness of each inference rule with respect to each (positive or negative occurrence of a) predicate can thus be distinguished. If renaming is not done then useless clauses are much less likely to be generated. The partial evaluation stage is not compulsory, but dramatically increases the effectiveness of the second stage.

5 Analysis of Proof Procedures

In this section we review two first order predicate calculus proof procedures. The first procedure is a clausal theorem prover by Poole and Goebel [22] and the second a Naive nH-Prolog proof system by Loveland [16].

The procedures are presented as meta-programs for three reasons. First, it is not the aim of this paper to develop efficient theorem provers based on the above proof procedures, but to demonstrate the applicability of analysis and transformation techniques, and the procedures as presented here suffice for this aim. Second, as the theorem prover you wish to analyse may not be implemented in a logic programming language, this approach demonstrates how an analysable version may be created without reimplementing all the intricacies present in the original prover. Third, the theory of meta-programming has received a lot of attention during the last few years and notable improvements in meta-programming style and specialisation techniques have cleared the way for this approach to be taken seriously as it can now deliver comparative performance results to conventional systems [9], [6], [2].

5.1 Poole-Goebel Theorem Prover

Consider the following implementation of a clausal theorem prover described by Poole and Goebel in [22]. This procedure is based on the model elimination proof procedure by Loveland [15].

```
solve(Goal, Anc) ←
        literal(Goal),
        depth_bound(Depth),
        prove(Goal, Anc, Depth).

prove(G, A, D) ← D > 0,
        infer(G, A, A1, G1),
        D1 is D − 1,
        proveall(G1, A1, D1).

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

infer(G, A, _ , [ ]) ←                    % Ancestor Resolution
        member(G, A).
infer(G, A, [G_neg|A], Body) ←            % Input Resolution
        clause((G : −Body)),
        neg(G, G_neg).
```

Program 1: Proof procedure with inference rules

Program 1 is similar to the procedure in [22]. A depth first iterative deepening mechanism has been added and the program has been rewritten in a style that makes the two inference rules explicit. Different sets of inference rules can be substituted for the above two to get a proof procedure for a different logic.

neg(X,Y) is true if X is the negation of Y with X and Y both in their simplest form. *clause* is not the usual Prolog *clause*. *clause*(X,Y) is true if $X \leftarrow Y$ is a contrapositive of an arbitrary clause of the form $a_1 \vee \ldots \vee a_n \leftarrow b_1 \wedge \ldots \wedge b_m$ and X is a literal. *literal*(G) is true if G is an atom or the negation of an atom in the object language. *member*(X,Y) is true if X is a member of the list Y.

This procedure is sound when executed with a Prolog system that includes the occur check. As the occur check relates to the underlying system on which Program 1 will be executed, the analysis method is independent of whether the occur check is implemented or not. Program 1 can be regarded as a sound and complete specification of the model elimination theorem prover. Obviously, other optimisations could be made [24], but we are interested only in the success set of Program 1, not the search space. Any results inferred for Program 1 will therefore also be valid for the "real" theorem prover (see figure 1).

5.2 Obtaining a Specialised Prover

The proof procedure may be specialised with respect to a specific class of theorems (queries to Program 1 of the form $solve(T, [\,])$) or with respect to a given theorem (e.g. $solve(c(\_), [\,])$. The more limited the queries, the more precise the results are likely to be for a given approximation method. As the theorems that we want to prove are known in the examples that follow, we generally approximate with respect to a given theorem. In general, this does not preserve the completeness of the proof system, but this can easily be restored by defining a set of queries which have to be tried [21] and doing the above analysis for each query.

As the technique described in this paper is not dependent at all on the proof system, this technique can be applied to any proof procedure and an analysis may indicate the following:

1. areas of the search space that may be avoided as it will never contribute to a proof,
2. failure for infinitely failed derivations or
3. failure to find a proof before a proof is even attempted.

We illustrate the first analysis by considering the following object theory from [18], which is a version of Popplestone's "Blind Hand Problem", together with the proof procedure in Section 5.1. This problem is also used in [24] as a benchmark.

1. $\neg A(x,z,y) \vee \neg H(z,y) \vee I(x,P(y))$
2. $\neg H(w,y) \vee H(z,G(z,y))$
3. $A(x,z,G(z,y)) \vee \neg H(w,y) \vee \neg I(x,y)$
4. $\neg H(z,y) \vee H(z,L(y))$
5. $A(S,E,N)$
6. $\neg I(x,L(y))$
7. $\neg A(x,E,y) \vee R(x)$
8. $\neg A(x,z,y) \vee A(x,z,L(y))$
9. $\neg A(x,z,y) \vee A(x,z,P(y))$
10. $\neg A(x,z,y) \vee B(x,P(G(z,L(y))))$
11. $C(y) \vee \neg Q(x,T,y) \vee \neg R(x)$
12. $\neg A(x,w,y) \vee \neg B(x,y) \vee Q(x,z,G(z,y))$
13. $\neg A(x,w,y) \vee A(x,w,G(z,y)) \vee I(x,y)$
14. $\neg C(y)$

Object Theory 1: DBA BHP

Our aim is to detect necessary inference rules for proving the above 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 are needed.

We now apply our analysis method to the Program 1 and Object Theory 1 as follows:

1. Partially evaluate Program 1 with respect to the object theory obtained from clauses 1 to 13 of Object Theory 1 (clause 14 is the negation of the goal we want to prove) and the general goal $solve(\_,\_)$. This yields P_1.

2. Compute a regular approximation of P_1 with respect to the initial query to solve $solve(c(\_),[\,])$, where $c(\_)$ is the goal we want to prove and the empty negative ancestor list $[\,]$ indicates the start of the proof.

An empty approximation for all the different *member* procedures results which indicates that no ancestor resolutions are needed when trying to prove $c(\_)$. Furthermore, empty approximations for the following *prove* procedures results, namely $h, i, \neg a, \neg h, \neg c, \neg q, \neg b$ and $\neg r$. This indicates that all branches in the object level search tree, that correspond to one of these literals, are failure branches and can therefore be deleted. Removal of all ancestor resolution and the useless *prove* procedures from P_1 will yield P_3.

The deletion of all ancestor resolutions is a surprising result as it is not at all obvious from Object Theory 2 that ancestor resolution is never needed to prove $c(\_)$. Furthermore, only clauses 5 to 13 need to be considered in the proof of this theorem. Deletion of the corresponding useless clauses results in a specialised proof procedure with a greatly reduced search space which will allow a much faster proof as we will show in Section 6.

Partial evaluation and approximation made it possible to detect that Object Theory 2 is an instance of a class of theories where input resolution (deduction) is the only inference rule needed to prove the given theorem. It also shows that the limiting factor in proving the above theorem in current logic programming systems is not its inference rule, but the representation chosen for the object theory (its inability to represent negative literals in the head of a clause).

The analysis performed in this section therefore provides us with information about the inference rules needed as well as the representation needed. This information may now be used in different ways:

1. to construct a specialised prover that takes advantage of this information,
2. to switch to a different and possibly much simpler proof system that may dramatically improve the proof process or
3. to change the representation.

5.3 Naive nH-Prolog

As a second proof procedure consider the following set of inference rules specifying naive nH-Prolog as described in [16]. The intuitive idea of this prover is to try and 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 corresponds to one application of the splitting rule.

The inference rules for nH-Prolog are given in Program 2.

```
infer([start], [ ], [ ], [ ]).                                      % End
infer([start], [A|As], [A|As], [[restart, Goal]]) ←                 % Restart
        goal(Goal).
infer([start, G|Gs], A, A_new, [[start|G_new]]) ←                   % Reduction
        clause((Head : −B)),
        pick_a_head(Head, G, Rest),
        append(Rest, A, A_new),
        append(B, Gs, G_new).

infer([restart, _], [ ], [ ], [ ]).                                 % End restart block
infer([restart], [_|As], As, [[restart, Goal]]) ←                   % Restart
        goal(Goal).
infer([restart, G|Gs], [G|As], [G|As], [[restart|Gs]]).             % Cancellation
infer([restart, G|Gs], [A|As], [A|A_new], [[restart|G_new]]) ←      % Reduction
        clause((Head : −B)),
        pick_a_head(Head, G, Rest),
        append(Rest, As, A_new),
        append(B, Gs, G_new).
```

Program 2: Naive nH-Prolog

The procedure for *prove* is taken from Program 1. We assume the usual definition for *append*. *pick_a_head*(*Head*, *G*, *Rest*) is true if G is a literal from the multihead *Head* and *Rest* is the remaining literals in the multihead (deferred heads). *clause* is again not the usual Prolog *clause*, but a clause containing atoms only. Program 2 is a correct 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 affect soundness. The same remarks about the occur check stated in the previous section also holds for this procedure.

Program 2 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 (the number of atoms in the body of an object clause is finite [2]).

5.4 Failure Detection

We illustrate the second and third analyses described in Section 5.2 by analysing the following object theory from [16] in conjunction with Program 2. This object theory was used to demonstrate the incompleteness of the naive nH-Prolog proof system.

$q \leftarrow a, b$
$a \leftarrow c$
$a \leftarrow d$
$c \vee d$
$b \leftarrow e$
$b \leftarrow f$
$e \vee f$

Object Theory 2: Program from [16]

The following deduction was also given that showed that naive nH-Prolog would never terminate when trying to prove the query q.

(0)	$?-q.$	
(1)	$:-a, b$	
(2)	$:-c, b$	
	$\vdots$	
(5)	$:-\ \#[f, d]$	
(6)	$:-q\ \#f[d]$	$\{restart\}$
(7)	$:-a, b\ \#f[d]$	
	$\vdots$	
(17)	$:-\ \#d\ [f, d]$	
(18)	$:-q\ \#f[d]$	$\{restart\}$
	$\vdots$	

Deduction 1: Nonterminating deduction from [16]

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 reoccurring pattern of deductions. In this case, it is impossible to detect 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 4.

We again apply our analysis method to the Program 2 and Object Theory 2 as follows:

1. Partially evaluate Program 2 with respect to Object Theory 2 and the general goal $solve(\_, \_)$. This yields P_1.
2. Compute a regular approximation of P_1 with respect to the initial query to solve $solve([start, q], [\,])$, where q is the goal we want to prove and the empty deferred head list $[\,]$ indicates the start of the proof.

An empty approximation for *solve* results and indicates that the above proof 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. The 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) [3]. 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 Object Theory 2. In general, the existence of more powerful theorem provers that are able to terminate all non-terminating deductions can not be assumed. The only way to identify failure for some non-terminating deductions may be to use a method such as the one described in this paper.

6 Performance Results

In this section we give some performance results for the "Blind Hand Problem" and indicate how the speedup given by this method over partial alone increases with increasing depth bound. The timings for proving the "Blind Hand Problem" one hundred times on a SPARC station IPC using a depth bound of eight are given in Table 1. All timings are in seconds.

Blind Hand Problem	
Program	Time
Program 1 with Object Theory 1	5.279
Partially evaluated program	2.691
Useless clauses w.r.t. $c(\_)$ deleted	0.501

Table 1: Timings for running $\leftarrow solve(c(Y), [])$ one hundred times

After specialisation, the "Blind Hand Problem" is solved very efficiently. All the redundant ancestor resolutions have been eliminated and a large number of contrapositives have been deleted. Furthermore, the only literals that can contribute to a proof are a, c, q, b, r and $\neg i$.

The approximation time for this example was 5.8 seconds. This might seem a long time compared to the speedup achieved, but as we have only used a prototype implementation to construct the approximation, we feel that this analysis time can be greatly reduced. The five times speedup achieved when a depth bound of only eight was needed to prove this theorem suggests that when larger depth bounds are required to prove difficult theorems, the possible speedup achievable will be much greater. We also feel that as the depth bounds needed to prove large theorems increase, there will come a point where the analysis time will become insignificant compared to the time taken to prove the theorem and the time saved by the deletion of useles clauses.

The following results show how the speedup over partial evaluation alone increases with increasing depth bound. The times are for finding all possible proofs to the "Blind Hand Problem" with a given depth bound. In the table PE stands for the partially evaluated program and PE+D the partially evaluated program with useless clauses deleted.

Blind Hand Problem				
			Time	
Depth Bound	Program 1	PE	PE+D	Speedup (PE/PE+D)
8	4.55	2.69	0.53	5.1
9	25.73	14.81	2.63	5.6
10	157.00	99.41	14.12	7.0
11	1030.00	617.79	83.60	7.4
12	> 5000	3925.00	411.09	9.6

Table 2: Timings for finding all possible proofs one hundred times

As a last indication of the possible improvements that the method described in this paper may give over partial evaluation, consider reversing a list of integers. Although the naive reversing program used in this experiment is a definite theory, this example shows clearly the effect of the increasing depth bound on the speedup that can be expected. However, the branching factor is quite small for this example and better speedups may be expected with a bigger branching factor. Program 1 was specialised with respect to naive reverse and query $solve(reverse(\_, \_), [\,])$. The approximation time for this example was 1.6 seconds.

Naive Reverse				
			Time	
Depth Bound	Program 1	PE	PE+D	Speedup (PE/PE+D)
10	0.829	0.471	0.089	5.3
20	5.300	3.349	0.379	8.4
30	14.730	10.200	0.869	11.7
40	30.290	24.448	1.510	16.2
50	57.220	46.439	2.360	19.7
60	95.700	81.261	3.430	23.7

Table 3: Timings for reversing a list one hundred times

Most interesting first-order theorems lie somewhere between being definite and extensively requiring the full power of the first-order theorem prover. For Program 1, the speedups obtainable therefore depend on the extent to which ancestor resolution is required, the depth bound required for the proof, the branching factor of the search tree and the number of contrapositives in the object theory that may be deleted. Further experiments will indicate the influence of each factor on the speedup achieved.

7 Discussion

In Section 5.2 the "Blind Hand problem" was analysed in conjunction with the Poole-Goebel prover. What does the same analysis infer when done on the same problem but in conjunction with the naive nH-Prolog prover?

In this case the information inferred shows that we may need a restart block as there may be deferred heads ($A(\_,\_,\_)$ and $I(\_,\_)$). We are therefore not able to delete any inference rules. This example also shows the close interaction or coupling between inference rules and representation. We have a more compact representation for the Naive nH-Prolog approach (only positive literals in the object theory), but pay for this in our inability to delete any inference rules. In the Poole-Goebel approach, we have a more elaborate representation with contrapositives, but gain in the detection of a redundant inference rule for this problem.

In [22] Program 1 was analysed with respect to the following set of propositional clauses.

$a \leftarrow b \wedge c$
$a \vee b \leftarrow d$
$c \vee e \leftarrow f$
$\neg g \leftarrow e$
$g \leftarrow c$
g
$f \leftarrow h$
h
d

Object Theory 3: Set of clauses from [22]

Their analysis indicated that the negative ancestor check may be necessary for a, $\neg a$, b and $\neg b$. When applying the method in this paper to the above object theory and Program 1, empty approximations are derived for all the specialised *member* procedures in the above theory, except for the *member* procedures for literals $\neg a$, f and h. This shows the our method is at least as precise as the method developed in [22], for this example.

In [25] Sutcliffe describes syntactically identifiable situations in which reduction does not occur in chain format linear deduction systems. 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. The following object theory was analysed using this method.

$r \vee \neg p(a) \vee \neg q$
$\neg p(a) \vee q$
$p(a) \vee \neg q$
$p(a) \vee q$
$\neg r \vee \neg t \vee \neg s$
$t \vee u$
$\neg u$
$s \neg p(b)$
$p(b)$

Object Theory 4: Set of clauses from [25]

His analysis method indicates that no reductions are needed for the subset $\{r, \neg t, u, \neg s, \neg p(b)\}$ and query $r \wedge \neg p(a) \wedge q$. Our analysis method infers exactly the same results when applied to Program 1 and Object Theory 4. 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 following Object Theory.

$$\neg p(a) \vee q$$
$$p(a) \vee \neg q$$

Object Theory 5: Subset of clauses from Object Theory 4

In this case LISL analysis infers no useful information, but our analysis applied to Program 1 and Object Theory 5 gives an empty approximation for *solve* which indicates failure.

Wakayama [26] 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. Nevertheless, as was illustrated with the "Blind Hand Problem", this method is precise enough to detect that only input deduction is necessary to solve it.

It is worth emphasising that this method is independent of the proof procedure which is not the case for the methods developed by Poole and Goebel, Sutcliffe and Wakayama. Any theorem prover that can be written as a logic program can be analysed and optimised using this method. Furthermore, this method can be significantly improved by making use of other decidable theories in the approximation with a lesser information loss (better approximation) than for regular unary logic programs. We also would like to develop a general approach using meta-programming to specify theorem provers. Such an approach might be adapted from work such as [10].

8 Conclusions

We presented a novel way of analysing theorem provers with general analysis and transformation techniques developed for logic programming. Two first order clausal theorem provers were analysed with these techniques and nontrivial results inferred. We also indicated how the analysis information may be used to guide choices regarding representation and inference rules. In future work we intend to work on the suggested test problems in [28] and refine the analysis presented in this paper and to develop better approximations based on other decidable theories.

Acknowledgements

We would like to thank Gerd Neugebauer, Technische Hochschule Darmstadt, for reading a draft version of this paper and for his constructive remarks and many valuable suggestions that contributed greatly to the overall appearance of this paper.

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1990.

[2] D.A. de Waal. The power of partial evaluation. To be published in the Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation, Louvain-la-Neuve, 1993.

[3] D.A. de Waal and J. Gallagher. Logic program specialisation with deletion of useless clauses. To appear as a poster in ILPS93, 1993.

[4] 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.

[5] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[6] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[7] 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.

[8] J. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[9] C.A. Gurr. Specialising the ground representation in the logic programming language Gödel. To be published in the Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation, Louvain-la-Neuve, 1993.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[11] N. Jones and H. Søndergaard. Abstract interpretation of logic programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[12] 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.

[13] J.W. Lloyd. *Foundations of Logic Programming: 2nd Edition.* Springer-Verlag, 1987.

[14] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[15] D.W. Loveland. *Automated theorem proving: a logical basis.* North-Holland, 1978.

[16] D.W. Loveland. Near-Horn Prolog and beyond. *Journal of Automated Reasoning*, 7(1):1–26, 1991.

[17] 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.

[18] D. Michie, R. Ross, and G.J. Shannan. G-deduction. *Machine Intelligence*, 7:141–165, 1972.

[19] L. Naish. Types and the intended meaning of logic programs. Technical report, University of Melbourne, 1990.

[20] C.K. Gomard N.D. Jones and P. Sestoft. *Partial Evaluation and Automatic Program generation.* Prentice Hall, 1993.

[21] G. Neugebauer. Reachability analysis. To be published in the Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation, Louvain-la-Neuve, 1993.

[22] 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.

[23] B. Selman and H. Kautz. Knowledge compilation using Horn approximation. In *Proceedings of the Ninth National Conference on Artificial Intelligence.* AAAI/MIT Press, 1991.

[24] M.E. Stickel. A Prolog Technology Theorem Prover. In *International Synposium on Logic Programming,* Atlantic City, NJ, pages 211–217, Feb. 6-9 1984.

[25] 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.

[26] 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.

[27] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In *Proceedings of the North American Conference on Logic Programming, Cleveland.* MIT Press, October 1989.

[28] L. Wos. Basic research problems: The problem of choosing the representation, inference rule and strategy. *J. Automated Reasoning*, 7(4):631–634, 1991.

[29] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.

[30] J. Zobel. Derivation of polymorphic types for Prolog programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming.* MIT Press, 1988.