Logic Program Specialisation With Deletion of Useless Clauses [1]

D.A. de Waal **J.P. Gallagher**
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

Abstract

In this paper we describe a method of program specialisation and give an extended example of its application to specialisation of a refutation proof procedure for first order logic. In the specialisation method, a partial evaluation of the proof procedure with respect to a given theory is first obtained. Secondly an abstract interpretation of the partially evaluated program is computed, and this is used to detect and remove clauses that yield no solutions (useless clauses). A proof is given that such clauses can be deleted from a normal program while preserving the results of all finite computations. The model elimination proof procedure described in [20] is specialised with respect to given theories, and the negative ancestor check inference rule can be eliminated in cases where it is not relevant. Our results for the model elimination prover, obtained by general-purpose transformations, are comparable to those obtained in [20] by a special-purpose analysis. It is shown that specialisations of the proof procedure can be achieved that cannot be obtained by partial evaluation. The method is applicable to any normal program, and thus provides an extension of the power of partial evaluation.

Keywords: partial evaluation, program specialisation, abstract interpretation.

1 Introduction

An algorithm for specialisation of logic programs will be described that can give considerably better results than partial evaluation. In this paper we show how to perform effective specialisation of a refutation proof procedure for first order logic, using general-purpose program transformation and analysis methods. We take the implementation of the model elimination

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

proof procedure given by Poole and Goebel in [20], and show how to specialise it with respect to a given object theory. The aim is to optimise proofs within the given theory. This could be used either to speed up theorem proving applications, or to compile "programs" written using full first order logic.

The method combines partial evaluation with abstract interpretation. It is first shown that partial evaluation alone does not give interesting results when specialising the proof procedure. Abstract interpretation is then used to delete *useless clauses* from the partially evaluated program, which results in effective specialisation.

The main aim of Poole and Goebel in [20] was to provide the full expressiveness of clausal theories, without necessarily incurring the overhead of using a full theorem prover. They developed a special purpose analysis of an object theory in order to optimise the prover for a given theory. More specifically, they aim to avoid the negative ancestor check of the model elimination procedure as much as possible, and in the case of definite object theories, to avoid it altogether.

The fact that, as will be shown, general transformation methods can yield comparable results to the special-purpose method in [20] is worth emphasising. Since our techniques are not dependent at all on this prover our approach could be applied to optimising other proof procedures as well. In general the purpose is to avoid the use of inference rules that are not relevant to a given theory or formula within a theory.

In the next section the model elimination prover is described. In Section 3 we discuss the use of partial evaluation to specialise the prover, and the limitations of this technique. Then the notion of a useless clause is defined, and in Section 5 a method of detecting useless clauses by abstract interpretation is described. This method is combined with partial evaluation to give the required specialisation. The correctness of the method is shown for normal programs, although in this paper only definite programs are specialised.

Several examples are then developed and the performance improvements are shown.

2 Clausal Theorem Prover

Consider the following implementation of a clausal theorem prover described by Poole and Goebel in [20]. This procedure is based on the model elimination proof procedure by Loveland [15].

$$
\begin{array}{ll}
solve(G, A) \leftarrow literal(G), & \\
\quad prove(G, A). & \\
 & \\
prove(G, A) \leftarrow member(G, A). & \%\ Negative\ ancestor\ check \\
prove(G, A) \leftarrow clause(G, B), & \%\ Resolution \\
\quad neg(G, GN), & \\
\quad proveall(B, [GN|A]). & \\
 & \\
proveall([\,], \_). & \\
proveall([G|R], A) \leftarrow prove(G, A), & \\
\quad proveall(R, A). &
\end{array}
$$

Program 1: Clausal theorem prover

This refutation proof procedure is identical to the procedure in [20], except for the addition of the first clause. $neg(X, Y)$ 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.

If this procedure is to be executed using current Prolog technology, the depth first search strategy of most Prolog systems makes the procedure incomplete. This shortcoming of Program 1 can easily be rectified, by augmenting the proof procedure with a depth bound [24]. This procedure can be run with successively larger depth bounds until a proof is found. Optimisation of this strategy is considered in [24] but it does not concern us here. The augmented proof procedure is given in Program 2.

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

$prove(G, A, D) \leftarrow D > 0,$ % *Negative ancestor check*
$\quad member(G, A).$
$prove(G, A, D) \leftarrow D > 1, D1\ \ is\ \ D - 1,$ % *Resolution*
$\quad clause(G, B),$
$\quad neg(G, GN),$
$\quad proveall(B, [GN|A], D1).$

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

Program 2: Proof procedure with depth bound

The occur check is also missing from most Prolog implementations. However, its lack in the proof procedure is not a problem that influences the specialisation method. Sound unification (provided as an option by systems such as Prolog by BIM), can be used after specialisation to ensure correct proofs.

We also do not go into the problems caused by indefinite answers, but it is worth mentioning that methods like the one proposed by Green in [11] can be used to extract these types of answers.

3 Partial Evaluation of the Theorem Prover

The theorem prover can be partially evaluated with respect to a given object theory. This is the same idea as partially evaluating a meta interpreter for logic programs with respect to a fixed object program [5], [7], [23].

The partial evaluator (called SP) used for our experiments is fully described in [6]. In principle we could use any partial evaluator based on the theory in [14], (augmented with a method of renaming predicates in the final partially evaluated program). In the next sections, the inherent limitations of partial evaluation for specialisation of the proof procedure are shown.

3.1 Partial Evaluation and Infinite Failures

Let P be a normal program and G a normal goal. Let P' be a partial evaluation of P with respect to G. We recall that the main result in [14] is that (when P' fulfils certain conditions) $P \cup \{G\}$ and $P' \cup \{G\}$ have exactly the same computed answers and finite failures. In particular, if $P \cup \{G\}$ has an infinite unsuccessful derivation, then so does $P' \cup \{G\}$.

Computations with the model elimination prover contain many infinite but unsuccessful branches. For example take the theory $\{p(f(X)) \leftarrow p(X), p(a) \leftarrow true\}$. Given the goal clause $\leftarrow \neg p(Z)$, it is clear that there is no refutation but that the computation tree for the goal $\leftarrow solve(not(p(Z)), [\,])$ with Program 1, is infinite. Therefore in any partial evaluation of the prover with respect to the given theory, the same goal will continue to have an infinite unsuccessful derivation.

We would like the specialised theorem prover to fail (finitely) for as many non-theorems as possible. This may mean that an infinite failure in the original prover may be turned into a finite failure in the specialised prover. Obviously, partial evaluation cannot give such specialisations.

3.2 Infinite Sets of Finitely Failed Computations

Now let us consider another aspect of the problem. Take the same theory and let the goal clause be $\leftarrow p(Z)$. The goal for Program 1 is thus $\leftarrow solve(p(Z), [\,])$. There is an infinite number of possible computed answers for Z, of the form $Z = f^n(a), n \geq 0$. Examining the derivations in more detail, we see that the derivation includes the following goals.

$\leftarrow prove(p(Z), [\,])$
$\leftarrow prove(p(Z_1), [not(p(f(Z_1)))])$
$\leftarrow prove(p(Z_2), [not(p(f(Z_2))), not(p(f(f(Z_2))))])$
$\dots$

For each such call to *prove* in the derivation, there are two matching clause heads in the prover. The first is the 'ancestor check' clause, namely $prove(X, Y) \leftarrow member(X, Y)$.

The resulting call to $member(X, Y)$ is always such that Y is a finite list consisting entirely of negative literals, while X is an atom. Hence the call always fails (finitely). However, an infinite number of such finitely failing calls to $member(X, Y)$ arise in the computation. The problem is how to remove an infinite number of similar finitely failing branches from a computation tree.

Clearly, a partial evaluation can consider only a finite number of atoms, and therefore only a finite number of calls to $member(X, Y)$. For the correctness of partial evaluation, at least one atom $member(X, Y)$ that has an infinite number of calls as instances has to be partially evaluated, and clearly any such atom will not fail.

Therefore, although all calls to $member(X, Y)$ in the computation of $\leftarrow prove(p(Z), [\,])$ fail finitely, partial evaluation cannot eliminate the useless 'ancestor check' from the specialised program.

In the next section, we show how to handle specialisations that eliminate both infinitely failed computations and infinite sets of finitely failed branches of a computation. Since we are dealing with infinite computation trees, global analysis techniques are required.

4 Useless Clauses

One of the applications of program analysis by abstract interpretation is to detect clauses that yield no solutions. This application has occurred 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 [16], [26], [19], [27].

Definition 4.1 *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 refutation of $P \cup \{G\}$.*

A stronger notion of uselessness is obtained by considering particular computations.

Definition 4.2 *useless clause with respect to a computation*
Let P be a normal program, G a normal goal, and let $C \in P$ be a clause. Then C is useless with respect to the SLDNF-computation $P \cup \{G\}$ if C is not used in any refutation in the computation of $P \cup \{G\}$, including all the subsidiary computations.

Obviously a clause that is useless by Definition 4.1 is also useless with respect to any goal by Definition 4.2.

4.1 Deletion of Useless Clauses

Useless clauses can be deleted from a program, but the SLDNF procedural semantics is not in general preserved, since loops may be eliminated. However, a slightly more limited but useful correctness result can be shown, namely, that the results of finite computations are preserved.

Proposition 4.3 *Deletion of useless clauses*
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.*

Proof. First note that if C is useless with respect to $P \cup \{G\}$ and the goal H occurs at a node in some SLDNF-tree in the computation of $P \cup \{G\}$, or in subsidiary computation trees, then C is useless with respect to $P \cup \{H\}$.

Let D be an SLDNF-refutation (resp. finitely failed SLDNF-tree) for $P \cup \{G\}$. The proof is by induction on the rank of D, as defined in [13].

Basis:

1. If D if of rank 0 and does not use C then removal of C does not affect D.
2. There is no refutation of rank 0 that uses C since otherwise C would not be useless with respect to $P \cup \{G\}$.
3. Suppose D is a finitely failed SLDNF-tree of rank 0 for $P \cup \{G\}$. Construct a finitely failed SLDNF-tree of $P' \cup \{G\}$ as follows. Let $\leftarrow L_1, \ldots, A, \ldots L_m$ be a non-leaf node in D, A be the selected atom, and C one of the clauses whose head unifies with A. For each such node in D, remove the branch connected to the node via C. The result is clearly a finitely failed SLDNF-tree for $P' \cup \{G\}$.

Induction: Suppose D is of rank $k+1$, and suppose the proposition holds for refutations and finitely failed trees of rank at most k.

1. If D is a refutation then C is not used in D, since otherwise C would not be useless with respect to $P \cup \{G\}$.
2. Suppose D is a refutation, and a ground negative literal $\neg H$ is selected at some step of D, where $P \cup \{\leftarrow H\}$ has a finitely failed SLDNF-tree (of rank at most k). If C is useless with respect to $P \cup \{G\}$ then as noted above it is also useless with respect to the sub-computation $P \cup \{\leftarrow H\}$. Then by the induction hypothesis, $P' \cup \{\leftarrow H\}$ has a finitely failed SLDNF-tree.
   The computed answer substitution for D is not affected by this step of the derivation. This is the case for all such negative literals selected in D, hence D is also an SLNDF-refutation of $P' \cup \{G\}$ (with the same answer substitution).
3. Suppose D is a finitely failed SLDNF-tree and C is used one or more branches of D. Then remove branches below the point where C is used, as for the rank 0 case. Suppose $\leftarrow L_1, \ldots, \neg H, \ldots L_m$ is a non-leaf node in D, where $\neg H$ is selected, and $P \cup \{\leftarrow H\}$ has a finitely failed SLDNF-tree (of rank at most k). Then $P' \cup \{\leftarrow H\}$ has a finitely failed SLDNF-tree, by the induction hypothesis (following the argument for step 2).
   Suppose $\leftarrow L_1, \ldots, \neg H, \ldots L_m$ is a leaf node in D, where $\neg H$ is selected, and $P \cup \{\leftarrow H\}$ has an SLNDF-refutation (of rank at most k). Similarly by the induction hypothesis $P' \cup \{\leftarrow H\}$ also has an SLNDF-refutation.
   Hence, there is a finitely failed SLDNF-tree for $P' \cup \{G\}$.

Hence the proposition holds for D of rank $k+1$.

Hence, by induction the result holds for SLDNF-refutations (resp. finitely failed SLDNF-trees) of any rank. □

But computations using P and P' are not identical, since removal of a useless clause might remove infinite failing derivations from the SLDNF tree for $P \cup \{G\}$. Hence an infinite failure in P can be turned into a finite failure in P'. This in turn can lead to new successful computations in P'. For example let $P = \{p \leftarrow p, r \leftarrow \neg p\}$. Then the clause $p \leftarrow p$ can be eliminated since $\leftarrow p$ has no solution. Hence $P' = \{r \leftarrow \neg p\}$. $P \cup \{\leftarrow r\}$ loops, but $P' \cup \{\leftarrow r\}$ succeeds.

However, preservation of finite computations seems to be what is required for most purposes, since usually a looping program would be regarded as erroneous. Note also that removal of infinite failures preserves the perfect model semantics of a stratified normal program [22].

5 Detection of Useless Clauses

It is undecidable whether an arbitrary clause is useless (with respect to the computation of a goal), but global analysis methods based on abstract interpretation can establish sufficient conditions for uselessness.

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 activations and results of the clause. If the analysis shows that a clause yields no results (in a computation), then it is useless with respect to that computation. We quote the frameworks described in [25] and [2] as examples of appropriate methods.

Space does not permit a detailed description of the analysis method that we used for the results reported in this paper, but we outline the main ideas. Detailed descriptions can be found in [9] and [10]. The central notion is that of a *safe approximation* of a program.

Definition 5.1 *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.*

Note that the approximation is intended to give a sufficient condition for uselessness of clauses in P, and therefore the stronger condition, namely,

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

is not required. However in practice the approximations that we use fulfil the latter condition.

Safe approximation preserves the result of definite (i.e. positive) goals. The following definition and proposition shows how this can be used to detect useless clauses.

Definition 5.2 *Let G be a normal goal. Then G^+ denotes the definite goal obtained by deleting all negative literals from G.*

Proposition 5.3 *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 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 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.

5.1 Regular Safe Approximations

In the work reported here we used Regular Unary Logic (RUL) programs, defined in [26]. For any program P, we construct, using an abstraction of the T_P operator, an RUL program that is a safe approximation of P [9]. This can then be used to detect useless clauses, since finite failure of definite goals is decidable in an RUL program.

An RUL approximation algorithm has been implemented in Prolog and has been successfully run on large programs. In the worst case such algorithms, like other computations associated with regular sets [26], require exponential time, but in practice worst cases are rarely encountered. Also, a dependency analysis of the program being approximated allows the computation to be broken up into a large number of small pieces, (only mutually recursive predicates need be simultaneously approximated, and so sizable programs can effectively be approximated (e.g. the PRESS program with 59 predicates and 165 clauses needs approximately 7 seconds including dependency analysis).

To detect uselessness with respect to the computation of a goal, we use a variation of a *query-answer* transformation, as described in [9]. Such transformations are well-known in abstract interpretation of logic programs. They are based on "magic sets" and "Alexander" transformations, which have been adapted to implement program analysis methods [3], [4], [12] and [17]. A safe RUL approximation of the query-answer program is computed, and a clause that is useless with respect to the computation of a goal is detected by finding useless clauses in the corresponding query-answer program. A query-answer transformation can be computed in time linear to the size of the program and goal being transformed.

6 A Specialisation Procedure

When a program is generated automatically, for example by partial evaluation, it is quite likely that it could contain useless clauses. By contrast, the deletion of useless clauses is probably not very widely applicable in the normal course of events, since obviously most people try to write programs in which every clause is useful.

We therefore use the detection and deletion of useless clauses as a post-processing step following partial evaluation. Given a program P and a goal G, a specialised program is obtained in two stages.

1. Obtain a partial evaluation P_1 of P with respect to G, (or some G' such that G is an instance of G').

2. Delete clauses in P_1 that are useless with respect to $P_1 \cup \{G\}$, yielding P_2.

By the correctness of partial evaluation, and by Proposition 4.3, we have the following properties.

1. if $P \cup \{G\}$ has an SLDNF-refutation with computed answer θ then $P_2 \cup \{G\}$ has an SLDNF-refutation with computed answer θ; and
2. if $P \cup \{G\}$ has a finitely failed SLDNF-tree then $P_2 \cup \{G\}$ has a finitely failed SLDNF-tree.

The specialisation thus preserves all the results of terminating computations, which is all one is likely to be interested in. Note that 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 clearly some calls to a procedure may fail or loop while other calls succeed. Renaming allows the failing branches to be removed while retaining the successful ones. If renaming is not done then useless clauses are much less likely to be generated.

In the partial evaluation of the theorem prover discussed below, a separate renamed version of the *prove* and *member* predicates is produced for each literal in the object theory. Therefore, if say, the negative ancestor check is useless for some literals but useful for others, the useless ones can be thrown out.

7 Specialisation of the Theorem Prover

In this section we apply our specialisation method to two object theories and compare our results with that obtainable by partial evaluation. The first example is treated fairly comprehensively, but because of space limitations, it is not possible to include all programs generated during the specialisation process.

7.1 Definite Theories

First we consider a definite object theory with the aim of specialising the proof procedures given in section 2 with respect to this theory.

Consider the following naive reverse program :

$$
\begin{array}{l}
reverse([\,],[\,]) \leftarrow reverse([X|Xs],Z) \\
\qquad reverse(Xs,Y), \\
\qquad append(Y,[X],Z). \\
append([\,],X,X). \\
append([X|Xs],Y,[X|Zs]) \leftarrow \\
\qquad append(Xs,Y,Zs).
\end{array}
$$

Object Theory 1: Naive reverse

Partial evaluation of the proof procedure given in Program 1 with respect to the above theory gives the following specialised program:

1. $solve(reverse(X0,X1),X2) \leftarrow prove\_1(X0,X1,X2).$
2. $solve(append(X0,X1,X2),X3) \leftarrow prove\_2(X0,X1,X2,X3).$

3. $solve(not\ reverse(X0, X1), X2) \leftarrow prove\_3(X0, X1, X2).$
4. $solve(not\ append(X0, X1, X2), X3) \leftarrow prove\_4(X0, X1, X2, X3).$
5. $prove\_1(X0, X1, X2) \leftarrow member\_1(X0, X1, X2).$
6. $prove\_1([\,], [\,], X0).$
7. $prove\_1([X0|X1], X2, X3) \leftarrow prove\_1(X1, X4, [not\ reverse([X0|X1], X2)|X3]),$
   $prove\_2(X4, [X0], X2, [not\ reverse([X0|X1], X2)|X3]).$
8. $prove\_2(X0, X1, X2, X3) \leftarrow member\_2(X0, X1, X2, X3).$
9. $prove\_2([\,], X0, X0, X1).$
10. $prove\_2([X0|X1], X2, [X0|X3], X4) \leftarrow$
    $prove\_2(X1, X2, X3, [not\ append([X0|X1], X2, [X0|X3])|X4]).$
11. $prove\_3(X0, X1, X2) \leftarrow member\_3(X0, X1, X2).$
12. $prove\_3(X0, X1, X2) \leftarrow prove\_2(X1, [X3], X4, [reverse(X0, X1)|X2]),$
    $prove\_3([X3|X0], X4, [reverse(X0, X1)|X2]).$
13. $prove\_4(X0, X1, X2, X3) \leftarrow member\_4(X0, X1, X2, X3).$
14. $prove\_4(X0, [X1], X2, X3) \leftarrow prove\_1(X4, X0, [append(X0, [X1], X2)|X3]),$
    $prove\_3([X1|X4], X2, [append(X0, [X1], X2)|X3]).$
15. $prove\_4(X0, X1, X2, X3) \leftarrow$
    $prove\_4([X4|X0], X1, [X4|X2], [append(X0, X1, X2)|X3]).$
16. $member\_1(X0, X1, [reverse(X0, X1)|X2]).$
17. $member\_1(X0, X1, [X2|X3]) \leftarrow member\_1(X0, X1, X3).$
18. $member\_2(X0, X1, X2, [append(X0, X1, X2)|X3]).$
19. $member\_2(X0, X1, X2, [X3|X4]) \leftarrow member\_2(X0, X1, X2, X4).$
20. $member\_3(X0, X1, [not\ reverse(X0, X1)|X2]).$
21. $member\_3(X0, X1, [X2|X3]) \leftarrow member\_3(X0, X1, X3).$
22. $member\_4(X0, X1, X2, [not\ append(X0, X1, X2)|X3]).$
23. $member\_4(X0, X1, X2, [X3|X4]) \leftarrow member\_4(X0, X1, X2, X4).$

Program 3: Program 1 partially evaluated with respect to Object Theory 1

Note that partial evaluation was unable to remove any of the negative ancestor checks (calls to *member* predicates) as discussed in section 3. Without additional specialisation, the overhead of the negative ancestor check is not avoided even though the check is not needed.

We now apply step two of our specialisation procedure to the above program with the aim of eliminating as many useless clauses as possible. A regular approximation of a query-answer transformed version of the above program (as discussed in Section 5) with top level query $solve(reverse(\_, \_), [\,])$ shows that all clauses except clauses 1,6,7,9 and 10 are useless with respect to the query.

Examining Program 3, we can interpret these results. For example, all of the clauses for *prove_3* and *prove_4* are useless, showing that there are no proofs of $\neg reverse(\_, \_)$ or $\neg append(\_, \_, \_)$. All the clauses for the different versions of *member* are useless, which means that the negative ancestor check never succeeds in any proof.

The result of deleting the useless clauses is the following specialised program:

Program 4 is a considerable simplification of the partially evaluated program (Program 3) derived in step 1 of the specialisation procedure. Nearly all the overhead of the meta-program has been eliminated. *prove_1* and *prove_2* correspond very closely to *reverse* and *append* in the original object program (Object Theory 1). However, the above program has

$solve(reverse(X0, X1), X2) \leftarrow prove\_1(X0, X1, X2).$

$prove\_1([\,], [\,], X0).$
$prove\_1([X0|X1], X2, X3) \leftarrow prove\_1(X1, X4, [not(reverse([X0|X1], X2))|X3]),$
$\quad prove\_2(X4, [X0], X2, [not(reverse([X0|X1], X2))|X3]).$

$prove\_2([\,], X0, X0, X1).$
$prove\_2([X0|X1], X2, [X0|X3], X4) \leftarrow$
$\quad prove\_2(X1, X2, X3, [not(append([X0|X1], X2, [X0|X3]))|X4]).$

Program 4: Final specialised program for Object Theory 1

a redundant argument, the ancestor list, which is never used. Methods such as the one described by Proietti and Pettorossi [21] might be used to delete unnecessary variables. The above program should also be nearly as efficient as the original object program when run directly by Prolog.

7.2 Non-Definite Theories

Consider the following object theory from [18], which is a version of Popplestone's 'Blind Hand Problem'. 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. $\neg A(x, w, y) \vee A(x, w, G(z, y)) \vee I(x, y)$

Object Theory 2: DBA BHP

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 checks are needed. Partial evaluation of the augmented proof procedure (Program 2) with respect to the object theory obtained from clauses 1 to 13 of the above theorem gives uninteresting results, as all the negative ancestor checks stay intact. However, by applying step two of our specialisation method to the partially evaluated program, and if we restrict our initial query to *solve* to queries of the form $solve(c(\_), [\,], \_)$, it is possible to detect that no negative ancestor checks are needed.

$solve(c(X0), X1, X2) \leftarrow prove\_4(X0, X1, X2).$
$prove\_1(s, e, n, X0, X1) \leftarrow X2\ is\ X1 - 1, X2 > 1.$
$prove\_1(X0, X1, l(X2), X3, X4) \leftarrow X5\ is\ X4 - 1, X5 > 1,$
$\quad prove\_1(X0, X1, X2, [not(a(X0, X1, l(X2)))|X3], X5).$
$prove\_1(X0, X1, p(X2), X3, X4) \leftarrow X5\ is\ X4 - 1, X5 > 1,$
$\quad prove\_1(X0, X1, X2, [not(a(X0, X1, p(X2)))|X3], X5).$
$prove\_1(X0, X1, g(X2, X3), X4, X5) \leftarrow X6\ is\ X5 - 1, X6 > 1,$
$\quad prove\_10(X0, X3, [not(a(X0, X1, g(X2, X3)))|X4], X6),$
$\quad prove\_1(X0, X1, X3, [not(a(X0, X1, g(X2, X3)))|X4], X6).$
$prove\_4(X0, X1, X2) \leftarrow X3\ is\ X2 - 1, X3 > 1,$
$\quad prove\_5(X4, t, X0, [not(c(X0))|X1], X3),$
$\quad prove\_7(X4, [not(c(X0))|X1], X3).$
$prove\_5(X0, X1, g(X1, X2), X3, X4) \leftarrow X5\ is\ X4 - 1, X5 > 1,$
$\quad prove\_6(X0, X2, [not(q(X0, X1, g(X1, X2)))|X3], X5),$
$\quad prove\_1(X0, X6, X2, [not(q(X0, X1, g(X1, X2)))|X3], X5).$
$prove\_6(X0, p(g(X1, l(X2))), X3, X4) \leftarrow X5\ is\ X4 - 1, X5 > 1,$
$\quad prove\_1(X0, X1, X2, [not(b(X0, p(g(X1, l(X2)))))|X3], X5).$
$prove\_7(X0, X1, X2) \leftarrow X3\ is\ X2 - 1, X3 > 1,$
$\quad prove\_1(X0, e, X4, [not(r(X0))|X1], X3).$
$prove\_10(X0, l(X1), X2, X3) \leftarrow X4\ is\ X3 - 1, X4 > 1.$

Program 5: Final specialised program for Object Theory 2

The final specialised program for Object Theory 2 after deletion of useless clauses is given in Program 5.

This is a surprising result as it is not at all obvious from Object Theory 2 that the negative ancestor check is never needed. In [20] it is stated that the class of theories, for which the negative ancestor check is not needed, is larger than that of definite clauses. Object Theory 2 is an instance of this class.

From the above examples it may look like our method can only delete all or none of the negative ancestor checks. This is not the case. Although this is not shown in the above examples, our method can selectively delete useless clauses from complex programs (delete only some of the negative ancestor checks that are not needed). This is possible since the partial evaluation constructs a separate version of the ancestor check for each object predicate.

In all the above examples, the depth bound has always been left unspecified and is only instantiated when a concrete query is actually given to the proof procedure to be executed.

8 Performance Results

We now give an indication of the improvement given by our method for each of the two examples. We compare the times it took to prove each theorem with three different programs:

1. the original meta-program;

2. the meta-program partially evaluated with respect to the object theory and an arbitrary theorem;

3. partial evaluation as in 2, but with the deletion of all useless clauses with respect to $\leftarrow solve(P, [\,])$, where P is a general atom containing the predicate of the theorem to be proved.

The difference between 2 and 3 show the advantage of deleting useless clauses from the partially evaluated program. Useless clauses can also be deleted with respect to an arbitrary theorem, $\leftarrow solve(\_, [\ \ ])$, but in general this gives more general results than 3.

In order to get reliable timings, we repeated each experiment one hundred times on a SPARCstation IPC using SICStus Prolog version 2.1 and give the total time it took to prove the each theorem one hundred times. All timings are in seconds.

The following table gives the results for reversing a list of fifty integers one hundred times.

Naive Reverse	
Program	Time
Program 1 with object program	38.901
Partially evaluated program	33.431
Useless clauses w.r.t. $reverse(\_, \_)$ deleted	0.949

Table 1: Timings for reversing a list of fifty integers one hundred times

As can be seen from the above table, the improvement over partial evaluation alone that deletion of useless clauses gives, is considerable. The reason for this improvement is that naive reverse needed a depth bound of fifty in the proof tree to reverse a list of fifty elements. At level n in the proof tree, ancestor checks of length n are avoided.

The results for the 'Blind Hand Problem' using a depth bound of eight are given in Table 2.

Blind Hand Problem	
Program	Time
Program 2 with object program	3.541
Partially evaluated program	2.041
Useless clauses w.r.t. $c(\_)$ deleted	0.391

Table 2: Timings for running $\leftarrow solve(c(Y), [], 8)$ one hundred times

After specialisation, the 'Blind Hand Problem' is solved very efficiently. Not only has all the redundant negative ancestor checks been eliminated, but most clauses generated by the contrapositives have also been deleted. The improvement is not as great as for naive reverse. Nevertheless, the improvement over partial evalaution is still considerable.

A general point worth emphasising is that the advantages of removing the ancestor check are much greater when a deeper proof is required. This was illustrated by the much greater efficiency gain for the naive reverse example, which used a tree of depth fifty, than for the second example in which the tree was only of depth eight.

9 Related Work and Discussion

In [20] a property of literals was identified that can be statically determined, in order to avoid the full generality of the full clausal proof procedure when not required. They defined conditions under which all the contrapositives are not needed as well as when it is not necessary to search up the tree (negative ancestor check). In [20] the following set of clauses was analysed

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;

Object Theory 3: Set of clauses from [20]

Their analysis indicates that the negative ancestor check may be necessary for a, $\neg a$, b and $\neg b$. Our specialisation method, after deleting clauses useless with respect to $\leftarrow solve(\_, [\,])$, shows that the negative ancestor check may be necessary for $\neg a$, f and h. The two analyses yield different results (both are approximations); the only common literal given by both methods is $\neg a$, which shows that only proofs of this literal may need the ancestor check.

In [20] it is shown that for a definite clause theory, the ancestor check is always completely eliminated. All our experiments on definite theories have also eliminated all ancestor checks and we suspect that our analysis gives this result for all definite theories.

As our method uses general program analysis and transformation techniques, it can be used to specialise any proof procedure, which is not the case for the static analysis method developed by Poole and Goebel. Our method also incorporates low level optimisations that are part of partial evaluation, such as removal of redundant structure [8].

Consider the negative ancestor check for $\neg a$ that both our methods indicated may be needed to prove the theorem in Object theory 3. The Poole-Goebel method will have a general *member* procedure in their final program whereas our method specialises the *member* procedure to a one argument procedure (as it is known that the first argument is fixed). This aspect is potentially important since if proof procedures with many complex inference rules, such as proof procedures for constraint logic programming, are to be efficiently specialised, the specialisation method must be able to do more that detecting when inference rules are not needed: it must also be able to specialise the remaining inference rules using partial evaluation or other program transformation techniques.

Furthermore, the Poole-Goebel static analysis restricts itself to signed predicate symbols, whereas the precision of ours depends on the abstract interpretation used to detect useless clauses.

10 Conclusion

Although the great potential of application of program specialisation has been noted, current techniques of partial evaluation are likely to be too weak to achieve good results. We have defined an algorithm for program specialisation that uses abstract interpretation to improve the results of partial evaluation. We showed that it gives specialisations comparable to a more specialised method. The importance of global analysis in program transformation is increasingly recognised, and in future work we intend to develop algorithms which integrate partial evaluation and abstract interpretation to a greater degree.

References

[1] BIM. Prolog by BIM Reference Manual, Release 3.1. 1991.

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

[3] M. Codish. *Abstract Interpretation of Sequential and Concurrent Logic Programs.* PhD thesis, The Weizmann Institute of Science, 1991.

[4] S. Debray and R. Ramakrishnan. *Canonical computations of logic programs.* Technical Report, University of Arizona-Tucson, July 1990.

[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. Transforming logic programs by specialising interpreters. In *ECAI-86, Proc. of the 7th European Conference on Artificial Intelligence, Brighton, England*, pages 109–122, 1986.

[8] 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.

[9] 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*, pages 151–167, Springer-Verlag, 1993.

[10] 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.

[11] C. Green. Theorem-proving by resolution as a basis for question-answering systems. *Machine Intelligence*, 4:183–205, 1969.

[12] T. Kanamori. *Abstract interpretation based on Alexander templates.* Technical Report TR-549, ICOT, March 1990.

[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] 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.

[17] C.S. Mellish. *Using specialisation to reconstruct two mode inference systems.* Technical Report, University of Edinburgh, March 1990.

[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] 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.

[21] M. Proietti and A. Pettorossi. An automatic transformation strategy for avoiding unnecessary variables in logic programs. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation,* Manchester 1991, pages 216–218, Workshops in Computing, Springer-Verlag, 1992.

[22] T. Przymusinski. On the declarative and procedural semantics of logic programs. *J. Automated Reasoning*, 5(2):167–206, 1989.

[23] S. Safra and E.Y. Shapiro. Meta interpreters for real. In H-J. Kugler, editor, *Proc. IFIP'86, Dublin*, North-Holland, 1986.

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

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

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