Tutorial on Specialisation of Logic Programs

J.P. Gallagher
Department of Computer Science
University of Bristol
University Walk
Bristol BS8 1TR, U.K.
e-mail: john@compsci.bristol.ac.uk

Abstract

In this tutorial the specialisation of declarative logic programs is presented. The main correctness results are given, and the outline of a basic algorithm for partial evaluation of a logic program with respect to a goal. The practical considerations of gaining efficiency (and not losing any) are discussed. A renaming scheme for performing structure specialisation is then described, and illustrated on a well-known string matching example. The basic algorithm is enhanced by incorporating abstract interpretation. A two-phase specialisation is then described and illustrated, in which partial evaluation is followed by the detection and removal of useless clauses. This is shown for the specialisation of a proof procedure for first order logic. The specialisation of meta programs is very important in logic programming, and the ground representation for object programs has to be handled. Some techniques for doing this are described. Comparisons are made in the tutorial to similar work in other programming languages, and the similarities and differences between them and logic program specialisation is discussed.

1 Aims of Logic Program Specialisation

1.1 Correctness of Specialisation

The main aim of program specialisation in logic programming is to take a program P and a goal G, and derive a program P' specialised with respect to G, with the following property:

- Computations of $P \cup \{G\}$ and $P' \cup \{G\}$ give identical computed answer substitutions.

In addition, the specialisation in most cases should satisfy the following:

- $P \cup \{G\}$ fails finitely if and only if $P' \cup \{G\}$ does.

The preservation of finite failures is important for the soundness of the transformation because of the negation-by-(finite)-failure inference rule, But as will be seen, there are many occasions when the desired specialisation changes a loop into a finite failure, and in this case a more limited soundness result is proved.

In [LS91] the main results for partial evaluation in logic programming are developed, and these results form the core of current methods in logic program specialisation.

1.2 Practical Considerations

Apart from correctness the important practical requirement on the specialisation is that:

- Computation of $P' \cup \{G\}$ is more efficient than $P \cup \{G\}$.

This is by no means assured by straightforward partial evaluation algorithms, and there are several aspects to the problem.

- Control: it must be ensured that the unfolding does not "compile in" poor control choices, inadvertently increasing the run-time search space.

- Space usage: unfolding tends to propagate data structures by performing substitutions at compile time. This can have a deleterious effect on space efficiency, and hence time efficiency, in the specialised program. It is also desirable to remove redundant structure from the original program.

- Indexing: specialisation should not remove indexing information from the program, which is needed to get fast unification. Some techniques proposed for specialising logic programs, such as introducing equalities and disjunctions, may have a harmful effect.

1.3 Comparisons

The basic aims and methods of logic program specialisation are comparable to the techniques developed for other languages, but there are several distinctive aspects.

- The inherent non-determinism of logic programs allows great flexibility in unfolding a program.

- Secondly, the so-called static-dynamic distinction is hardly present in logic programs, in contrast to functional or imperative languages, or rather it appears

in a rather different form. Consideration of determinacy and choice points is far more important for control, while partially instantiated structures are more the rule than the exception. during partial evaluation.

- Thirdly, information is propagated both backwards and forwards by substitution application; one is often able to instantiate the "input" further during partial evaluation, as well as compute part of the "output".

- Fourthly, the combination of meta-programming with partial evaluation is even more significant in logic programming than in other formalisms, since there are many non-standard procedural readings of logic programs (top-down, bottom-up, coroutining, parallel, ...), which can be expressed as meta-programs and compiled by means of partial evaluation. Such applications have great potential in program development and artificial intelligence.

In this tutorial an attempt is made to present the central core of theoretical results and basic algorithms that are used in current work on logic program specialisation. Examples are demonstrated using the SP logic program specialisation tool [Gal91] which incorporates many techniques developed both inside and outside the logic programming world. This approach seems preferable to a survey of the state of the art, which would be more objective but less systematic.

1.4 Logic Programming versus Prolog

A good deal of research in logic program specialisation has dealt with ways to handle the various idiosyncracies of Prolog, such as the cut, non-logical primitives and side-effects. On the other hand a body of results has been obtained for declarative logic programs, that is, programs which are first order logic formulas (normal clauses).

While research on Prolog is of obvious value (because there are already so many Prolog programs written) this tutorial does not consider the non-logical aspects of Prolog. As a general remark it may be said that whatever results are obtained for declarative logic programs are weakened when applied to Prolog, and therefore specialisation of Prolog programs is less effective. Since many of the builtins (such as arithmetic) are included in the declarative part of Prolog, a restriction to declarative programs does not mean a serious lack of expressiveness. Similar remarks as for Prolog hold for other logic program dialects (such as concurrent logic programs) which diverge in some or other aspect from first order logic.

1.5 Specialisation: Beyond Partial Evaluation

Partial evaluation has limitations, some of which are mentioned below. The most important ideas for extending the scope of program specialisation, in logic programming as in other languages, is program analysis using *abstract interpretation* and related methods. Elsewhere [Gal92] we have argued that logic program specialisation should be regarded as an application of global program analysis, along with mode and type inference and such like.

In this sense the terms "partial evaluation/deduction" which are sometimes used synonymously with "specialisation" are somewhat outmoded. Specialisation is a less operational term, and allows transformations not achievable by partial evaluation alone. For instance the notion of a "more

specific logic program" [MNL88] is a kind of specialisation, in which a logic program clause may be replaced by a more instantiated version, or even eliminated from the program, without altering the program's success or finite failure set. Similarly the notion of a "useless clause" defined below, as detected by an abstract interpretation, is beyond the scope of partial evaluation. Transformations such as folding and renaming may also be incorporated in the specialisation process.

1.6 Historical Notes: Partial Evaluation in Logic Programming

Partial evaluation was introduced into logic programming by Komorowski [Kom82], who set out a number of basic transformations and strategies for applying them. The work of Kahn [KC84] showed that a Prolog to Lisp compiler could be produced by partial evaluation, by partial evaluation of an interpreter for Prolog written in Lisp, (but here of course the partial evaluation is applied to Lisp, not Prolog). Venken [Ven84] wrote a partial evaluator for Prolog and applied it to optimisation of deductive database queries.

The combination of meta-programming with partial evaluation started with [Gal86], [Tak86] and [SS86], followed by [LS88], [Ste86] and others. The two main streams of research concerned "compiling" of non-standard interpreters such as coroutining interpreters, and language "enhancements" and "extensions" such as expert system shells, and debugging systems, achieved by enriching the basic structure of a logic program interpreter. The related work of "compiling control" [MBK89] was developed independently. Program development methodology and the role of partial evaluation was studied in [Lak89] and [Kom89]. Partial evaluation of other languages in the logic programming family have also been studied [SS86], [Smi91] and more recently [Gur92].

A major clarification of the field was provided by Lloyd and Shepherdson [LS91], who identified the closedness and independence conditions necessary for correctness of partial evaluation. A provably correct algorithm based on these results was published later [BL90].

The growing importance of abstract interpretation in logic program specialisation was identified in [GCS88], and later in [GB91].

A partial evaluator for full Prolog was written by Sahlin [Sah91] and has been widely distributed. The goal of writing a self-applicable partial evaluator for Prolog has been attempted [FF88], [MB93], and recently a purely declarative self-applicable partial evaluator has been constructed in the language Gödel [Gur92].

2 Basic Definitions and Algorithm

A basic algorithm for partial evaluation is now developed; refinements are introduced later. The next five definitions concerning partial evaluation are quoted from [LS91]. Other logic programming terminology is defined in [Llo87].

Definition 2.1 *resultant*
   A resultant is a first order formula $Q_1 \leftarrow Q_2$ where Q_i is either absent or a conjunction of literals, and any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Definition 2.2 *resultant of a derivation*

Let P be a normal program and $G_0 = \ \leftarrow G$ be a normal goal. Let $G_0, \ldots, G_n$ be an SLDNF derivation of $P \cup \{G_0\}$, where $G_n = \ \leftarrow Q$ and $\theta_1, \ldots, \theta_n$ is the sequence of substitutions associated with the derivation. Let $\theta = \theta_1 \theta_2 \ldots \theta_n$. Then the derivation has length n, computed answer $\theta|_G$ and resultant $G\theta \leftarrow Q$. Note that if G is an atom then the resultant is a normal clause.

Definition 2.3 *partial evaluation*

Let P be a normal program and A an atom. Let T be an SLDNF tree for $P \cup \{\leftarrow A\}$, and let $\leftarrow G_1, \ldots, \leftarrow G_n$ be goals chosen from the non-root nodes of T such that there is exactly one goal from each non-failing branch of T. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, \leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$ is called a partial evaluation of A in P.

If $\mathbf{A}$ is a finite set of atoms, then a partial evaluation of $\mathbf{A}$ in P is the union of the partial evaluations of the elements of $\mathbf{A}$.

The concepts of closedness and independence given next are essential *global* properties needed to establish correctness of a partial evaluation.

Definition 2.4 *closed*

Let S be a set of first order formulas and $\mathbf{A}$ a finite set of atoms. Then S is $\mathbf{A}$-closed if each atom in S containing a predicate symbol occurring in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}$.

Definition 2.5 *independence*

Let $\mathbf{A}$ be a set of atoms. Then $\mathbf{A}$ is independent if no two atoms in $\mathbf{A}$ have a common instance.

Given an atom A and a program P, there exist in general infinitely many different partial evaluations of A in P. Some fixed rule for generating resultants is used, called an *unfolding rule*.

Definition 2.6 *unfolding rule*

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

2.1 Partial Evaluation Theorem (Lloyd-Shepherdson)

The central result proved in [LS91] is the following theorem about correctness of partial evaluation.

Theorem 2.1 *Let P be a normal program, and $\mathbf{A}$ an independent set of atoms. Let P' be a partial evaluation of $\mathbf{A}$ in P. Then for all goals G such that $P' \cup G$ is $\mathbf{A}$-closed:*

- *$P \cup G$ has an SLDNF-refutation with computed answer θ iff $P' \cup G$ has an SLDNF-refutation with computed answer θ.*

- *$P \cup G$ has a finitely-failed SLDNF-tree iff $P' \cup G$ has a finitely-failed SLDNF-tree.*

This theorem has clear implications for correct algorithms for partial evaluation of a program P with respect to a goal G. The focus of the algorithm must be to compute an independent set of atoms, such that, if P' is some partial evaluation of $\mathbf{A}$ in P and $P' \cup G$ is $\mathbf{A}$-closed. Most algorithm designers had previously concentrated on the local aspects of the algorithm, such as unfolding rules and loop prevention.

```
INPUT:   a program P and goal G,
         and unfolding rule U.
OUTPUT:  a set of atoms

begin
   A[0] := the set of atoms in G
   i := 0
   repeat
      P' := a partial evaluation
               of A[i] in P using U
      S := A[i] U
           { p(t) | B <- Q,p(t),Q' in P'
                  OR
                  B <- Q,not(p(t)),Q' in P' }
      A[i+1] := abstract(S)
      i := i+1
   until A[i] = A[i-1] (modulo variable
                          renaming)
end
```

Figure 1: Basic Algorithm

2.2 Basic Algorithm

In Figure 1 the outline of a naive algorithm for computing a set of atoms is given.

Note that in the algorithm there is an operation called $abstract(S)$ whose role will be discussed shortly. Its purpose is to ensure termination of the algorithm, and also to satisfy the independence requirement.

Definition 2.7 *An operation $abstract(S)$ is any operation satisfying the following condition. Let S be a set of atoms; then $abstract(S)$ is a finite independent 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.*

If the algorithm terminates (see below) then it returns an independent set of atoms $A[i]$. By construction of $A[i]$, it holds that if P' is a partial evaluation of $A[i]$ in P using U, then $P' \cup G$ is $A[i]$-closed. Hence by the partial evaluation theorem, P' is a correct partial evaluation of P with respect to G.

2.3 Two Termination Problems

In partial evaluation of logic programs, two distinct questions about termination arise.

1. Termination of the unfolding rule U.

2. Termination of the iterative algorithm, that is, the `repeat` loop in the above algorithm (global termination).

(Note: in partial evaluation of functional languages a similar division of the termination problem arises [Dan93]). Usually, the first of these problems is emphasised – but in our view it is the easier of the two to solve. The question of appropriate unfolding rules is discussed in Section 4. The global termination problem is concerned with the *abstract* operation, and will be discussed in connection with abstract interpretation in Section 5.

```
match(P,T) :-
        match1(P,T,P,T).

match1([],_,_,_).
match1([A|Ps],[A|Ts],P,T) :-
        match1(Ps,Ts,P,T).
match1([A|_Ps],[B|_Ts],P,[_|T]) :-
        \+ (A = B),
        match1(P,T,P,T).
```

Figure 2: Naive Matching Algorithm

3   Partial Evaluation With Structure Specialisation

Let P be a program, and $\leftarrow G$ a goal. For convenience and without loss of generality, we assume that a clause $answer(\bar{x}) \leftarrow G$ is included in P, where $\bar{x}$ are the distinct variables occurring in G and $answer$ is a predicate symbol not occurring in P except in the head of that clause. Thus we can restrict attention to partial evaluation of the goal $\leftarrow answer(\bar{x})$.

3.1   Example: String Matching

We consider a well-known example in the literature of partial evaluation, namely, the derivation of an efficient string-matching algorithm from a naive algorithm, shown in Figure 2. The derived algorithm is similar in structure to the Knuth-Morris-Pratt string-matching algorithm. The example is discussed by [Fuj91], [Lam89], [Smi91] and others.

Let the goal to be partially evaluated be $\leftarrow match([a, a, b],)$ that is, searching for the pattern $[a, a, b]$ in an unknown string. Thus we assume a clause

```
answer(X) :-  match([a,a,b],X).
```

The basic algorithm (with a suitable unfolding rule, see Section 4) produces the following set of atoms.

```
answer(A1)
match1([a,a,b],A1,[a,a,b],A1)
match1([a,b],A1,[a,a,b],[a | A1])
match1([b],A1,[a,a,b],[a,a | A1])
```

The corresponding partial evaluation of these atoms is as follows:

```
answer(A1) :- match1([a,a,b],A1,[a,a,b],A1).
match1([a,a,b],[a | A1],[a,a,b],[a | A1]) :-
     match1([a,b],A1,[a,a,b],[a | A1]).
match1([a,a,b],[A0 | A1],[a,a,b],[A0 | A1]) :-
     \+ (a = A0),
     match1([a,a,b],A1,[a,a,b],A1).
match1([a,b],[a | A1] ,[a,a,b],[a,a | A1] ) :-
     match1([b],A1,[a,a,b],[a,a | A1]).
match1([a,b],[A0 | A1] ,[a,a,b],[a,A0 | A1] ) :-
     \+ (a = A0),
     \+ (a = A0),
     match1([a,a,b],A1,[a,a,b],A1).
```

```
match1([b],[b | A1] ,[a,a,b],[a,a,b | A1] ) :-
     true.
match1([b],[A0 | A1] ,[a,a,b],[a,a,b | A1] ) :-
     match1([a,b],A1,[a,a,b],[a | A1]).
```

Next we show how this program can be simplified by specialising the predicates.

3.2   Renaming for Structure Specialisation

Although traditionally the complexity of logic program computation has been measured by the number of logical inferences in it, the efficient use of storage is also of great importance. Often a program can be specialised merely by specialising the structures that occur in the predicates [GB90]. By combining partial evaluation with structure specialisation, much greater performance improvements are possible than by partial evaluation alone.

Several partial evaluators for Prolog have included some kind of structure specialisation (e.g. [Fuj91], [SS86], [Sah91], [Gal91]). However, given the basic algorithm above, it can be seen that structure specialisation can easily be incorporated and shown to be correct. In order to do so, predicate renaming is defined as follows.

Definition 3.1 *renaming definition*
   Let A be an atom, and let $x_1, \ldots, x_n$ be the distinct variables in A, in order of their first occurrence. Let p be a predicate symbol different from that in A. Then the clause $p(x_1, \ldots, x_n) \leftarrow A$ is called a renaming definition for A.
   Let $\mathbf{A}$ be a set of atoms; then a set of renaming definitions for $\mathbf{A}$ is the set of renaming definitions for the atoms in $\mathbf{A}$, with a distinct predicate name (different from any predicate in $\mathbf{A}$) chosen for each definition.

Let $\mathbf{A}$ be the set of atoms produced by the basic algorithm with program P and goal $\leftarrow answer(\bar{x})$. Construct a set of renaming definitions R for $\mathbf{A} - \{answer(\bar{x})\}$. Let $\mathbf{A}'$ be the heads of the clauses in R. For each $B \in \mathbf{A} - \{answer(\bar{x})\}$, let $B' \leftarrow B$ be its renaming definition in R. Then it can be seen that if there is a derivation of length n from $\leftarrow B$ to $\leftarrow Q$ in P, then there is a derivation of length $n+1$ from $\leftarrow B'$ to $\leftarrow Q$ in $P \cup R$. Clearly also the derivations of $\leftarrow answer(\bar{x})$ in P are also derivations in $P \cup R$.

Thus, if P' is a partial evaluation of $\mathbf{A}$ in P then there is a partial evaluation, R' say, of $\mathbf{A}'$ in $P \cup R$, in which the bodies of the clauses are the same as in P'.

As a consequence of the closedness property, every atom in the bodies of clauses in R' is an instance of some $B \in \mathbf{A}$. We can thus perform folding on every atom [Sek89], using the corresponding renaming clause in R. The predicates occurring in the resulting program, say $\bar{R}$, are either renaming predicates, or $answer$.

3.3   Correctness of Renaming Transformation

By the correctness of partial evaluation and the unfold-fold transformations [Sek89], the programs $P \cup R$ and $P \cup \bar{R}$ are equivalent with respect to computations of $\leftarrow answer(\bar{x})$. But all clauses except those in $\bar{R}$ can be dropped since they are not reachable. Hence $\bar{R}$ can be regarded as the result of the partial evaluation.

3.4 Independence and Polyvariant Specialisation

It can be seen that we can now drop the condition that the basic algorithm returns an independent set of atoms, since the set $\mathbf{A}'$ defined above is independent and thus satisfies the condition of the partial evaluation theorem.

Thus the problem of polyvariant specialisation is solved since clearly any number of different atoms with the same predicate can be produced by specialisation, and the renaming procedure ensures that they are correctly distinguished.

(Note that since we use the *answer* predicate, we can also ignore any problems that arise due to non-independence of the atoms in the goal $\leftarrow G$).

Returning to the `match` example, renaming clauses for the atoms produced by the basic algorithm are:

```
m1(A1) :- match1([a,a,b],A1,[a,a,b],A1).
m2(A1) :- match1([a,b],A1,[a,a,b],[a | A1]).
m3(A1) :- match1([b],A1,[a,a,b],[a,a | A1]).
```

Thus the final program obtained after partial evaluation and folding is:

```
answer(X0) :-
      m1(X0).
m1([a | X0]) :-
      m2(X0).
m1([X0 | X1]) :-
      \+ (a = X0),
      m1(X1).
m2([a | X0]) :-
      m3(X0).
m2([X0 | X1]) :-
      \+ (a = X0),
      \+ (a = X0),
      m1(X1).
m3([b | X0]) :-
      true.
m3([X0 | X1]) :-
      \+ (b = X0),
      m2([X0 | X1]).
```

The repeated call in the second clause for `m2` can easily be removed by a post-processing step. It can be seen that the partial evaluation contains no redundant structure such as the pattern $[a, a, b]$ or subpatterns, as were present in the original set of atoms generated by the basic algorithm.

4 Unfolding Rules

In the basic algorithm an unfolding rule U is assumed, which, given an atomic goal and a program, returns a finite set of resultants. There are many possible unfolding rules, and it is an unanswered question whether there is a single useful general-purpose unfolding rule. Experience suggests that a handful of different rules is needed. The SP program specialisation system [Gal91] gives the user the choice of several unfolding rules.

Some general principles can be stated.

1. The unfolding rule effectively makes some control decisions at compile time, so care must be taken in unfolding choice points. Assuming that the specialised program will use a left-to-right computation rule (as in Prolog), this means that *only the leftmost choicepoint in a goal should be unfolded*. For example, consider the program

```
q :- s.
r :- a.
r :- b.
```

with the goal

```
<- q,r.
```

If the atom `r` is unfolded, then result is the two clauses

```
<- q,a.
<- q,b.
```

With a left-to-right computation rule this means that if `a` fails then on backtracking `q` must be re-solved, unlike the original program. So the partial evaluation may be less efficient than the original if `s` is an expensive computation.

2. Unfolding determinate goals is almost always a good idea. Following the discussion on the previous point, it can be seen that unfolding determinate atoms cannot introduce extra computation into a program. The importance of determinacy in logic programming seems to correspond to the importance of static arguments in functional programming, possibly because in a function, a known argument normally determines the result.

3. An atom which has a finite number of solutions may be completely unfolded (assuming it occurs at the leftmost choicepoint). Finiteness analysis can thus play an important role in determinate computations.

4. Loop prevention, the basis of several unfolding strategies, is not on its own a good basis for unfolding rules, since it is based on the assumption that "as much as possible" should be unfolded without looping. As discussed above, this is not the case. In out opinion, finiteness analysis (see previous point) is a good place to use loop prevention techniques.

5. The handling of negative goals has to be safe; if no non-ground negative literal is selected then the unfolding is safe. Unfolding rules based on constructive negation have also been proposed [CW89].

In the SP system, the default unfolding rule is quite simple but surprisingly effective. It uses no loop-checking techniques.

Definition 4.1 *Let* $\leftarrow A$ *be an atomic goal.*

1. *Let* $A_1 \leftarrow B_1, \ldots, A_n \leftarrow B_n$ *be the clauses whose heads unify with* A.

2. *Unfold each of the ← B_i so long as determinate choices can be made. At the end of this process the result is either a failed goal or a goal ← R_i in which each atom in R_i either matches at least two clause heads, or is a non-ground negative literal.*

3. *Return the set of resultants of form $A\theta_i ← R_i$, where θ_i is the substitution associated with the derivation from ← A to ← R_i.*

The definition of "determinate" can be varied.

- The standard definition is that an atom is determinate if it matches exactly one clause head in the program.

- Ground negative literals can also be regarded as determinate, provided that their evaluation does not loop.

- A "look-ahead" of a finite number of computation steps could be used to detect further cases of determinacy. For example, given the program

```
p(X) :- X > 0, q(X).
p(X) :- X =< 0, r(X)
```

the goal `<- p(3)` is determinate (after look-ahead of 1 step).

- An analysis could determine cases where an atom has at most one solution [Sah90], in which case it could be treated as determinate. (But loops may be ignored by such an analysis; see Section 5.1).

5 Abstract Interpretation in Program Specialisation

As mentioned in Section 2.3, the second termination problem in the basic algorithm is global termination. This problem is equivalent to ensuring that the sets of atoms generated in the sequence $A[1], A[2], \ldots$ do not grow too large, and is the reason for the presence of the operation $abstract(S)$ in the algorithm.

Example: Let P be the program for reversing a list with an accumulating argument.

```
reverse(nil, L, L) .
reverse([X | Xs], Ys, Zs) :-
    reverse(Xs, Ys, [X | Zs] ).
```

Let G be the goal `<- reverse(_,_,nil)`. Then a partial evaluation generates atoms of the form

```
reverse(_, _, nil ),
reverse(_, _, [X1] ) ,
reverse(_, _, [X1,X2] ),
 ......
```

and it is clear that, without abstraction, the algorithm does not terminate.

Roughly speaking, abstraction generalises the atoms that appear in the bodies of resultants, which are the atoms (instances of which) are potentially selected in the partial evaluation. By generalising the atoms, it is certain that the

```
INPUT:   a program P,
         abstract atomc goal (A, Theta),
         and abstract unfolding rule U.
OUTPUT:  a set of abstract atoms

begin
   A[0]  := {(A,Theta)}
   i := 0
   repeat
      P' := a partial evaluation
            of A[i] in P using U.
      A[i+1] := A[i] U
         { (p(t),Beta) | (B,Theta') in A[i],
            ((B.rho <- Q,p(t),Q',Phi)) in P'
            OR
            (B.rho <- Q,not(p(t)),Q',Phi) in P',
            Phi \= bottom,
            Beta = restrict(Phi,vars(p(t)))}
      i := i+1
   until A[i] = A[i-1] (modulo variable
                       renaming)
end
```

Figure 3: Enhanced Algorithm

resulting partial evaluation will handle *at least* all of the calls that arise. This can be reformulated in the language of abstract interpretation. Let P be a program and G a goal. For each predicate p, we wish to construct a (finite) description of the set of activations of p in the computation of $P \cup \{G\}$. This description is a safe approximation if any actual activation is described, along with possibly some additional activations. There are many frameworks for abstract interpretation of logic programs that can compute such a description. In [GB91] and [GCS88] we proposed program specialisation algorithms based on abstract interpretation.

We assume a domain **Sub** of *abstract substitutions*, with a complete partial order $\sqsubseteq$ and bottom element $\bot$. There is a monotonic *abstraction* function α that maps a set of substitutions to an element of **Sub**. We also assume an operation $restrict(\Theta, V)$, where Θ is an abstract substitution, V is a set of variables and the result is a restriction of Θ to V.

5.1 Enhancement of the Basic Algorithm

Here we only indicate how the basic algorithm may be enhanced to include more sophisticated descriptions of sets of activations. In Figure 3 the enhanced algorithm is shown. Instead of computing a sequence of sets of atoms, the enhanced algorithm computes a sequence of sets of pairs (A, ϕ), where A is an atom and $\phi \in$ **Sub** is an abstract substitution. We call a pair (A, Θ) and *abstract atom*.

An unfolding rule U can then be abstracted to take an abstract atom and return a set of pairs $(A ← R, \Phi)$ where Φ is an abstract substitution applied to the resultant $A ← R$. For each atom B in R, the restricted substitution $restrict(\Phi, vars(B))$ can be computed. If $\Phi = \bot$, then the resultant can be eliminated since this is a sufficient condition for showing that no successful computations are possible using that resultant.

The *abstract* operation is no longer needed since the abstractions are incorporated in the abstract substitutions. The enhanced algorithm potentially allows descriptions of activations of any desired precision, making use of the many abstract domains that have been developed for analysing logic program computations.

5.2 Abstract Interpretation and Useless Clauses

Rather than develop this idea here, an alternative approach with which we have experimented is described. A two-stage process is employed, in which we first perform a partial evaluation with the basic algorithm, but then use abstract interpretation to get a better result. This is unlike some other approaches to using abstract interpretation in program specialisation, in which abstract interpretation is used before partial evaluation to give control information.

The notion of *useless clause* is quoted from [dWG92b].

Definition 5.1 *useless clause*
Let P be a normal program and let $A \leftarrow B$ be a clause. Then $A \leftarrow B$ is **useless** with respect to P if for all instances B' of B, $P \cup \{\leftarrow B'\}$ has no refutation.

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

Definition 5.2 *useless clause with respect to a computation*

Let P be a normal program, G a normal goal, and let $A \leftarrow B$ be a clause. Then $A \leftarrow B$ is useless with respect to the computation $P \cup \{G\}$ if there is no atom A' such that:

1. A' is selected in some SLDNF derivation (or failure derivation of a negated literal) in the computation of $P \cup \{G\}$,

2. A' unifies with A, with mgu θ, and

3. $P \cup \{\leftarrow B\theta\}$ has an SLDNF-refutation.

Obviously a clause that is useless by Definition 5.1 is also useless with respect to any goal by Definition 5.2.

Useless clauses can be deleted from a program without affecting the results of finite computations (including finite failures). The following theorem is proved in [dWG92b].

Proposition 5.3 *Deletion of useless clauses*
Let P be a normal program containing a clause C. Let P' be the program obtained by removing C from P. Then for all goals G, 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.

However, deletion of useless clauses does not preserve the full procedural equivalence of a program, since an infinite loop may be turned into a finite failure.

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

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

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

Figure 4: A Model Elimination Prover

5.3 Detecting Useless Clauses By Abstract Interpretation

Abstract interpretation computes, for a given goal and a program, a description of a superset of the computed answers. Therefore if the superset is empty, the goal fails with respect to the program. Thus abstract interpretation can be used to find sufficient conditions for useless clause detection.

In our experiments we have used an abstraction based on regular unary logic programs (RUL programs) [YS90]. For a given program and goal, we can compute an RUL program that describes a superset of the success set of the program [GdW92]. It is decidable whether any goal fails with respect to an RUL program, so this gives us a sufficient criterion to find useless clauses. namely, if there is some clause whose body fails in the RUL program, then that clause is useless. Goal-dependent useless clauses can be detected using similar techniques.

5.4 Example: Specialising a Proof Procedure

We take the model elimination prover [Lov78] given by Poole and Goebel in [PG86], and show how to specialise it with respect to a given object theory. The aim is to optimise proofs within the given theory. In [PG86] a version of a model elimination proof procedure is written; this is a refutation theorem prover for full first-order logic in clausal form. A version similar to theirs, but incorporating a depth-bound argument for use with an iterative deepening search strategy [Sti84], is shown in Figure 4. Poole and Goebel showed that given an object theory, a simplified version of the prover could be produced, using a a special-purpose analysis of the program.

Our aim was to achieve the same (or better) specialisation using partial evaluation and related general-purpose program transformation methods.

The object theory is represented as a set of clauses of form

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$$

where $A_1, \ldots, A_m, B_1, \ldots, B_n$ are atoms. `clause(G,R)` in the prover is true if $G \leftarrow R$ is a *contrapositive* of the above clause, where G is a positive or negative literal and R is a conjunction.

% If a is prime and $a = b^2/c^2$ then a divides b.
% $d(X,Y) \equiv X$ $divides$ Y, $\quad m(X,Y,Z) \equiv X*Y = Z$, $\quad p(X) \equiv X$ $prime$.
1. $p(a)$
2. $m(a, s(c), s(b))$
3. $m(X, X, s(X))$
4. $\neg m(X,Y,Z) \ \lor \ m(Y,X,Z)$
5. $\neg m(X,Y,Z) \ \lor \ d(X,Z)$
6. $\neg p(X) \ \lor \ \neg m(Y,Z,U) \ \lor \ \neg d(X,U) \ \lor \ d(X,Y) \ \lor \ d(X,Z)$
7. $\neg d(a,b)$

Figure 5: A Prime Number Theorem

The prover has two inference rules, *resolution* and a *negative ancestor check*. The aim of the specialisation will be to detect and delete calls to the negative ancestor check which are not required, since it is a costly inference rule (whose cost increases with the depth of the proof). For instance, if the object theory consists only of definite clauses then the prover is complete without any negative ancestor checks. For other theories, the check may be needed for some predicates but not for others.

The use of partial evaluation alone is unlikely to detect redundant ancestor checks (except for very simple theories). The reason is that, in order to detect that the ancestor check for a predicate, say p, is redundant, the partial evaluation must detect that an unbounded number of calls of the form member(not(p(...)), _) fail. The second argument of member is the ancestor list whose length is the same as the depth of the proof at that point. Clearly in the basic algorithm some generalisation of ancestor lists of any length has to be made, in which case most of the information about the contents of the list is lost.

In our approach, a partial evaluation first produces a specialised form of the prover in which there is a separate *prove* procedure for for each predicate and its negation. A second phase detects which of these procedures contains useless clauses. Thus cases of useless negative ancestor checks can be eliminated.

Example: Let Q be a theory of natural numbers, along with the statement of (the negation of) a prime number theorem (example from Chang and Lee [CL73]).

Partial evaluation of the prover with respect to this program yields poor results, since no negative ancestor checks are eliminated. The partial evaluation contains a separate version of the *prove* procedure for each predicate and its negation. For example the procedure for proving $d(\_,\_)$ becomes:

$prove(d(X,Y), A, D) \leftarrow member(d(X,Y), A).$
$prove(d(X,Z), A, D) \leftarrow D1 is D + 1,$
$proveall(m(X,Y,Z), [\neg d(X,Z)|A], D1)$
$\ldots$

However, by applying the deletion of useless clauses to the partially evaluated program, we get an empty approximation for the corresponding *member* procedures for $\neg p(\_)$, $\neg m(\_,\_,\_)$ and $d(\_,\_)$. It is therefore useless to search the negative ancestor list for occurrences of $\neg p(\_)$, $\neg m(\_,\_,\_)$ and $d(\_,\_)$ as these member checks can never succeed. The corresponding *member* clauses are useless with respect to any query to $prove(\_, nil)$ in the original proof procedure and can therefore be deleted.

If we restrict our initial query to *solve* to queries of the form $prove(d(\_,\_), [\ ], \_)$, it is also possible to detect that the negative ancestor checks for $p(\_)$ and $m(\_,\_,\_)$ are useless and it is therefore only necessary to search the ancestor list for occurrences of $\neg d(\_,\_)$. These results are not obvious and constitute a considerable improvement over the partially evaluated program.

6 Specialisation of Meta-Programs

Much of the interest of logic program specialisation lies in the large number of potential applications of specialising meta programs. The Futamura projections for deriving compiled code, compilers and compiler generators fall into this category, but there are many others. Specialisation of interpreters for non-standard computation rules such as coroutining, parallel or bottom-up rules has been studied [Gal86], [Tak86]. Also, language extensions and enhancements can be expressed as interpreters that perform extra operations to the standard computation. Examples include debugging interpreters, run-time error handling and protection [SS86], and expert system shells with certainty factors [Ste86]. In short, program specialisation offers a general compilation technique for the wide variety of procedural interpretations of logic programs.

6.1 Ground and Non-ground Representations

In logic programming there are two main approaches to meta-programming [HL89]. These are distinguished by the manner in which the object program is represented to the meta-program.

1. The non-ground representation: object level expressions are represented as non-ground terms in the meta-language, and in particular object level variables are represented by meta-variables.

2. The ground representation: object level expressions are represented as ground terms in the meta-language.

The difference between these two is profound. In the non-ground approach the archetypical example is the 3-line *solve* interpreter for definite clause programs. Non-ground meta-programs are concise, but unfortunately rather limited. In addition, care must be taken to ensure that the models of non-ground meta-programs are the intended ones [HL89].

The ground representation is an approach essentially related to Gödel numbering, in which the expressions (including variables) of one language are coded as ground terms in another. The ground representation is essential to perform

any complex meta-programming tasks in a sound way. However the ground representation is inherently more complex. For instance, operations at the object level such as unification, composition and application of substitutions, which the Prolog programmer takes for granted, must be explicitly coded in the meta-language. Self-application of a specialiser is an example of an application that requires the ground representation; this will be discussed further.

6.2 Program Specialisation for Efficient Programming with the Ground Representation

The effective use of ground representations is a major research issue in logic programming. The language Gödel [HL91] was designed with many system predicates to handle the ground representation efficiently. However, even with such assistance, given the overhead of programming with the ground representation, it may be thought impractical for "real" meta-programs.

Program specialisation offers a general approach to the problem, namely, the specialisation of meta-programs with respect to the representation of a given object program.

6.2.1 Example: Specialisation of Ground Representation

This example (shows how specialisation of an interpreter for a ground representation of object programs can result in the elimination of practically all operations concerning the representation. In the program below, an object program atom $p(f(X), Y)$ would be represented in the meta program by a term such as $p(f(var(0)), var(1))$. A program is represented as a single ground term which is a list of clauses. For instance the representation of the *member* program is:

```
[statement((member(var(1),[var(1)|var(2)]) :- true)),
 statement((member(var(1),[var(2)|var(3)]) :-
                     member(var(1),var(3)))) ]
```

Figure 6 shows an interpreter for programs represented in this way. The predicate `demo(G,G1,P)` is true if G is a goal, G1 is a ground instance of the goal which is provable from P. All three arguments are in the ground representation. Partial evaluation of a goal `demo(_,_,P)` where P is a given ground term representing an object program produces a program in which the `instance_of` predicate is compiled away. Figure 7 shows the result for the *member* program. (as produced by the SP program [Gal91]). It can be seen that the procedure `demo1_1` looks similar to a partial evaluation of a *solve* interpreter for *member*. It is also worth noting that the argument containing the program has been completely eliminated from `demo1_1` by the structure specialisation.

The calls to `anyterm` can also be shown to be redundant and eliminated, by an analysis similar to that described in [GdW93].

The purpose of this example is to show that ways of handling the ground representation using partial evaluation are being investigated, but it appears likely that the overhead of using the ground representation can be removed by program specialisation.

6.3 Self-Application

The production of effective and efficient self-applicable program specialisers is a major concern in logic programming

```
demo(G,G1,P) :-
    instance_of(G,G1),
    demo1(G1,P).

demo1(true,_).
demo1((G,Gs),P) :-
    demo1(G,P),
    demo1(Gs,P).
demo1(G,P) :-
    statement_instance(statement((G :- B)),P),
    demo1(B,P).

instance_of(X,Y) :-
    inst(X,Y,[],_).

inst(var(N),X,[],[(N/X)]) :-
    anyterm(X).
inst(var(N),X,[(N/X)|S],[(N/X)|S]).
inst(var(N),X,[(M/Y)|S],[(M/Y)|S1]) :-
    \+ N = M,
    inst(var(N),X,S,S1).
inst(T,T1,S,S1) :-
    T =.. [F|Xs],
    \+ F = var,
    functor(T,F,N),
    functor(T1,F,N),
    T1 =.. [F|Ys],
    inst_args(Xs,Ys,S,S1).

inst_args([],[],S,S).
inst_args([X|Xs],[Y|Ys],S,S2) :-
    inst(X,Y,S,S1),
    inst_args(Xs,Ys,S1,S2).

anyterm(X) :-
    atomic(X).
anyterm(T) :-
    T =.. [_|Xs],
    anyargs(Xs).

anyargs([]).
anyargs([X|Xs]) :-
    anyterm(X),
    anyargs(Xs).

statement_instance(S,P) :-
    member(C,P),
    instance_of(C,S).

member(X,[X|_]).
member(X,[_|Y]) :-
    member(X,Y).
```

Figure 6: A Simple Interpreter for the Ground Representation

```
demo1_1(true) :-
      true.
demo1_1((X0,X1)) :-
      demo1_1(X0),
      demo1_1(X1).
demo1_1(member(X0,[X0|X1])) :-
      anyterm(X0),
      anyterm(X1),
      demo1_1(true).
demo1_1(member(X0,[X1|X2])) :-
      anyterm(X0),
      anyterm(X1),
      anyterm(X2),
      demo1_1(member(X0,X2)).
```

Figure 7: Partially Evaluated Ground Interpreter

as in other languages. As discussed above, the achievement of this goal in logic programming requires effective handling of ground representation of programs.

In [MB93] a self-applicable partial evaluator is described but it is not clear whether this is a declarative program or not. Gurr [Gur92] describes an implementation of a self-applicable declarative partial evaluator called SAGE in the language Gödel [HL91]. In [dWG92a] the problems of specialising a unification algorithm for terms in the ground representation is discussed, which is a critical problem in effective specialisation of the ground representation.

7 Summary

We have reviewed the main results and techniques for specialisation of logic programs. There are effective logic programming specialisation systems in existence, including self-applicable specialisers. The distinctive features of logic program specialisation compared to specialisation of other programming languages was discussed.

In the future, increasing use of program analysis techniques to increase the power of specialisation can be expected. In the longer term, the merging of program specialisation and abstract interpretation is probable, since both have the same aim of exploiting knowledge about a program and its input in order to optimise the computation.

References

[BL90]   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.

[CL73]   C. Chang and R.C.-T Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CW89]   D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, MIT Press, 1989.

[Dan93] O. Danvy. Tutorial notes on partial evaluation. In *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles Of programming Languages, Charleston, Sth. Carolina*, January 1993.

[dWG92a] 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,* Manchester 1991, pages 205–221, Workshops in Computing, Springer-Verlag, 1992.

[dWG92b] D.A. de Waal and J. Gallagher. *Specialising a Theorem Prover.* Technical Report CSTR-92-33, University of Bristol, November 1992.

[FF88]   H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. In D. Bjørner, Ershov A, and N. Jones, editors, *Proc. of PEPM, Workshop on partial evaluation and mixed computation*, 1988.

[Fuj91] H. Fujita. *An Algorithm for Partial Evaluation with Constraints.* Technical Report TM-0484, ICOT, August 1991.

[Gal86] 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.

[Gal91] J. Gallagher. *A System for Specialising Logic programs.* Technical Report TR-91-32, University of Bristol, November 1991.

[Gal92] J. Gallagher. Static analysis for logic program specialisation. In M. Billaud et al., editor, *WSA-92: Analyse Statique*, pages 285 –294, Université Bordeaux I, 1992.

[GB90]   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.

[GB91]   J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 6(2), 1991.

[GCS88] J. Gallagher, M. Codish, and E.Y. Shapiro. Specialisation of prolog and fcp programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[GdW92] 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.

[GdW93] 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.

[Gur92] C. Gurr. *Specialising the Ground Representation in the Logic programming Language Gödel.* Technical Report, Department of Computer Science, University of Bristol, November 1992.

[HL89] P.M. Hill and J.W. Lloyd. Analysis of meta- programs. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, MIT Press, 1989.

[HL91] P.M. Hill and J.W. Lloyd. *The Gödel report.* Technical Report TR-91-02, University of Bristol, March 1991.

[KC84] K. Kahn and M. Carlsson. The compilation of prolog programs without the use of a prolog compiler. In *International Conference on Fifth Generation Computer Systems, Tokyo*, 1984.

[Kom82] H.J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *9th ACM Symposium on Principles of Programming Languages; Albuquerque, New Mexico*, pages 255 – 267, 1982.

[Kom89] H.J. Komorowski. *Synthesis of Programs in the Framework of Partial Deduction.* Technical Report 81, Department of Computer Science, Åbo Akademi, Lemminkäinengatan 14, SF-20520 Åbo, Finland., 1989.

[Lak89] A. Lakhotia. *A Workbench for Developing Logic Programs By Stepwise Enhancement.* PhD thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, 1989.

[Lam89] J. Lam. *Control Structures in Partial Evaluation of Pure Prolog.* Technical Report Masters Thesis, Department of Computational Science, University of Saskatchewan, 1989.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming: 2nd Edition.* Springer-Verlag, 1987.

[Lov78] D.W. Loveland. *Automated theorem proving: a logical basis.* North-Holland, 1978.

[LS88] G. Levi and G. Sardu. Partial evaluation of meta-programs in a multiple worlds logic language. In D. Bjørner, Ershov A, and N. Jones, editors, *Proc. of PEPM, Workshop on partial evaluation and mixed computation*, 1988.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[MB93] 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*, Springer-Verlag, 1993.

[MBK89] D. De Schreye M. Bruynooghe and B. Krekels. Compiling control. *Journal of Logic Programming*, 6):135–162, 1989.

[MNL88] 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.

[PG86] 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.

[Sah90] D. Sahlin. The mixtus approach to automatic partial evaluation for full prolog. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, MIT Press, 1990.

[Sah91] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology, 1991.

[Sek89] H. Seki. Unfold/fold transformations of stratified programs. In G. Levi and M. Martelli, editors, *Proc. of the 6th Int. Conf. on Logic Programming, Lisbon*, MIT Press, 1989.

[Smi91] D. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *PEPM'91, Proc. of the ACM Symposium on Partial Evaluation and Semantics-Based Program Transformation*, SIGPLAN Notices Vol. 26, No. 9, September 1991.

[SS86] S. Safra and E.Y. Shapiro. Meta interpreters for real. In H-J. Kugler, editor, *Proc. IFIP'86, Dublin*, North-Holland, 1986.

[Ste86] L. Sterling. Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proc. 3rd Int. Symp. on Logic programming, Salt Lake City, Utah*, pages 20 –27, 1986.

[Sti84] M.E. Stickel. A Prolog Technology Theorem Prover. In *International Synposium on Logic Programming,* Atlantic City, NJ, pages 211–217, Feb. 6-9 1984.

[Tak86] A. Takeuchi. Affinity between meta interpreters and partial evaluation. In H-J. Kugler, editor, *Proc. IFIP'86, Dublin*, North-Holland, 1986.

[Ven84] R. Venken. Partial evaluation of prolog and its application to source-to-source transformation and query optimisation. In *Proceedings of ECAI'84, Pisa*, 1984.

[YS90] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.