PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures *

Bern Martens Danny De Schreye Maurice Bruynooghe†

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {bern,dannyd,maurice}@cs.kuleuven.ac.be

Abstract

We present a procedure for partial deduction of logic programs, based on an automatic unfolding algorithm which guarantees the construction of sensibly and strongly expanded, finite SLD-trees. We prove that the partial deduction procedure terminates for all definite logic programs and queries. We show that the resulting program satisfies important soundness and completeness criteria with respect to the original program, while retaining the essentially desired amount of specialisation.

1 Introduction

Since its introduction in logic programming by Komorowski ([Komorowski, 1981]), partial evaluation has attracted the attention of many researchers in the field. Some, e.g. [Venken, 1984], [Venken and Demoen, 1988], [Sahlin, 1990], have addressed pragmatic issues related to the impurities of Prolog. Others were attracted by the perspective of eliminating the overhead associated with meta interpreters. Some examples are: [Gallagher, 1986], [Levi and Sardu, 1988], [Safra and Shapiro, 1986], [Sterling and Beer, 1989] and [Takeuchi and Furukawa, 1986]. Finally, a firm theoretical basis for the subject was described in [Lloyd and Shepherdson, 1991].

Just as in [Bruynooghe *et al.*, 1991a], we use the term "partial deduction" in this paper, rather than the more familiar "partial evaluation". Following [Komorowski, 1989], we do so because we want to leave the latter term for works taking into account the nonlogical features of Prolog and the order in which answers are produced. In the present paper, we adhere to the viewpoint taken in [Lloyd and Shepherdson, 1991] which states that the specialised program should have the same answers as the original one.

Indeed, the authors of [Lloyd and Shepherdson, 1991] present important criteria which, when satisfied by the specialised program, guarantee this to be the case. A partial deduction procedure imposing these criteria, is described in [Benkerimi and Lloyd, 1990]. However, termination of this procedure is not guaranteed, not even for definite logic programs. In this paper, we propose an alternative method which does terminate for all definite logic programs. A central part of any partial deduction procedure is an unfolding algorithm which builds the SLD(NF)-trees used as starting point for synthesising specialised clauses. In general, termination of this unfolding process is problematic in its own right. In [Bruynooghe *et al.*, 1991a], a general criterion for avoiding infinite unfolding is presented. In the present paper, we build on those results for formulating a terminating procedure for partial deduction, respecting the soundness and completeness conditions of [Lloyd and Shepherdson, 1991].

The paper is organised as follows. In section 2, we recapitulate (and adapt) some basic concepts in partial deduction from [Lloyd and Shepherdson, 1991], as well as the criteria for soundness and completeness presented there. We sketch the partial deduction method from [Benkerimi and Lloyd, 1990] and show an example on which the unfolding rules mentioned there do not terminate. In section 3, we introduce an automatic algorithm for finite unfolding, adapted from [Bruynooghe *et al.*, 1991a]. Next, in section 4, our partial deduction procedure is presented. We give an algorithm which implements it and prove its termination. Moreover, we prove that the method satisfies the criteria introduced in [Lloyd and Shepherdson, 1991]. We also show that the intended specialisation is indeed obtained. We conclude the paper in section 5 with a short discussion, including a brief comparison with the approach of [Benkerimi and Lloyd, 1990] and some directions for further research.

*work partially supported by ESPRIT BRA COMPULOG (project 3012)

†All authors are supported by the Belgian National Fund for Scientific Research.

2 Partial Deduction

2.1 Basic concepts, soundness and completeness

We assume familiarity with the basics of logic programming. Definitions of the following concepts can be found in [Lloyd and Shepherdson, 1991] and [Benkerimi and Lloyd, 1990]: *most specific generalisation (msg)*, *incomplete* SLD-tree, *resultant* of a derivation, *partial deduction for an atom in a program*, *partial deduction for a set of atoms in a program*, *partial deduction of a program wrt a set of atoms*, *independence* of a set of atoms, **A**-*closedness* of a set of formulas, **A**-*coveredness* of a program and goal. In [Lloyd and Shepherdson, 1991] and [Benkerimi and Lloyd, 1990], the definitions are given for normal programs and using the term "partial *evaluation*". In the present paper, we restrict ourselves to definite programs and goals and, as mentioned above, use the term "partial *deduction*". The necessary adaptations are straightforward (as exemplified in [Bruynooghe *et al.*, 1991a]).

We adapt the following theorem from [Lloyd and Shepherdson, 1991].

Theorem 2.1 Let P be a definite logic program, G a definite goal, **A** a finite, independent set of atoms, and P' a partial deduction of P wrt **A** such that $P' \cup \{G\}$ is **A**-covered. Then the following hold:

- $P' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

In other words, under the conditions stated in this theorem, computation with a partial deduction of a program is sound and complete wrt computation with the original program. This is clearly a very desirable characteristic of any procedure for partial deduction. It is therefore important to devise methods for partial deduction that ensure the conditions of theorem 2.1 are satisfied.

In [Benkerimi and Lloyd, 1990], one such method is presented. Basically, it proceeds as follows. For a given goal G and program P, a partial deduction for G in P is computed. This is repeated for any goal occurring in the resulting clauses which is not an instance of one already processed. Assuming the procedure terminates, one gets in this way a set of clauses **S** and a set **A** of partially deduced atoms such that **S** is **A**-closed. But one also wants **A** to be independent. In order to achieve this, the procedure is modified as follows. Whenever a goal occurring in **S** is not an instance (nor a variant) of one in **A**, but has a common instance with it, the latter is removed from **A** and a partial deduction is computed for their msg (which itself is therefore added to **A**) and added to **S**. The original partial deduction for the removed goal is itself also removed from **S**. The process stops if **A** becomes independent and **S** **A**-closed. **S** can then be used to synthesize a partial deduction of P wrt **A** which satisfies the conditions of theorem 2.1 for any goal G' which is an instance of G.

However, the tactic of taking msgs to make **A** independent causes an unacceptable loss of specialisation in the resulting partial deduction. To remedy this, the authors of [Benkerimi and Lloyd, 1990] introduce a renaming transformation as a pre-processing stage before running their algorithm. It amounts to duplicating and renaming the definitions of those predicates, occurring in the original goal G, which are likely to pose specialisation problems. The details can be found in [Benkerimi and Lloyd, 1990].

2.2 Unfolding

One question is left more or less unanswered until now: How to obtain the (incomplete) SLD-trees used as a basis for producing partial deductions? In other words, which computation rule should be used for building these trees (including the question of deciding when to stop the unfolding)? [Benkerimi and Lloyd, 1990] mentions 4 criteria and proposes the following one as the best : The computation rule R_v selects the leftmost atom which is not a variant of an atom already selected on the branch down to the current goal. However, this rule fails to guarantee the production of finite SLD-trees in all cases. We present a counter-example. It is the well-known "reverse" program with accumulating parameter.

Example 2.2

source program:
```
  reverse([],L,L).
  reverse([X|Xs],Ys,Zs) ← reverse(Xs,[X|Ys],Zs).
query:
  ←reverse([1,2|Xs],[],Zs).
```

The reader can verify that R_v generates an infinite SLD-tree.

Some authors have therefore combined R_v or other computation rules with a depth bound: (a.o.) [Levi and Sardu, 1988], [Sterling and Beer, 1986], [Takeuchi and Furukawa, 1986]. This does of course guarantee finiteness, but it seems a rather ad-hoc solution which does not reflect any properties of the given unfolding problem. We therefore proposed an alternative solution in [Bruynooghe *et al.*, 1991a]. (An extended version of this paper can be found in [Bruynooghe *et al.*, 1991b].)

3 An Algorithm for Finite Unfolding

In [Bruynooghe *et al.*, 1991a], a general criterion for avoiding infinite unfolding during partial deduction and a terminating unfolding algorithm based on it, are presented. In this section, we introduce a fully automatic version of that algorithm, tuned towards unfolding object-level definite logic programs. A slightly more sophisticated approach may be desirable when dealing with meta interpreters. We will not address that point in the present paper and concentrate on object-level programs. Although a slightly more accurate presentation of the algorithm itself is given, most of what follows now is adapted from [Bruynooghe *et al.*, 1991a]. The interested reader is referred to that paper for a full (and more general) account with all the technical details on the well-founded measures underlying our approach. Here, we only introduce what is necessary for a good understanding of algorithm 3.6.

For technical reasons, we will assume a numbering on the nodes of an SLD-tree (e.g. left-to-right, top-down and breadth-first). We will use the following notation for nodes in an SLD-tree: (G,i) where G is a goal of the tree having i as its associated number. (The notations "(G,i)" and "G" will be used interchangeably, as the context requires.)

We first define a weight-function on terms. It counts the number of functors in its argument.

Definition 3.1 Let **Term** denote the set of terms in the first order language used to define the theory P. We define $|.|$: **Term** $\rightarrow \mathbb{N}$ as follows:
If $t = f(t_1,\ldots,t_n), n > 0$
then $|t| = 1 + |t_1| + \cdots + |t_n|$
else $|t| = 0$

It is then possible to introduce weight-functions on atoms.

Definition 3.2 Let p be a predicate of arity n and **S**$=\{a_1,\ldots,a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$, a set of argument positions for p. We define $|.|_{p,S} : \{A|A$ is an atom with predicate symbol $p\} \rightarrow \mathbb{N}$ as follows:
$|p(t_1,\ldots,t_n)|_{p,S} = |t_{a_1}| + \cdots + |t_{a_m}|$

The next two definitions introduce useful relations on literals and goals in an SLD-tree.

Definition 3.3 Let $(G,i) = ((\leftarrow A_1,\ldots,A_j,\ldots,A_n),i)$ be a node in an SLD-tree τ, let $R(G) = A_j$ be the call selected by the computation rule R, let $H \leftarrow B_1,\ldots,B_m$ be a clause whose head unifies with A_j and let $\theta = mgu(A_j,H)$ be the most general unifier. Then (G,i) has a son (G',k) in τ, $(G',k) = ((\leftarrow A_1,\ldots,A_{j-1},B_1,\ldots,B_m,A_{j+1},\ldots,A_n)\theta,k)$. We say that $B_1\theta,\ldots,B_m\theta$ in G' are *direct descendents* of A_j in G and that A_j in G is a *direct ancestor* of $B_1\theta,\ldots,B_m\theta$ in G'.
The binary relations *descendent* and *ancestor*, defined on atoms in goals, are the transitive closures of the direct descendent and direct ancestor relations respectively. For A an atom in G and B an atom in G', A is an ancestor of B is denoted as $A >_{pr} B$ ("pr" stands for proof tree).

Notice that we also speak about one *goal* G' being an ancestor (or descendent) of another *goal* G. This terminology refers to the obvious relationships between goals in an SLD-tree and should not be confused with the proof-tree based relationships between literals, introduced in the previous definition. The following definition does introduce a relationship between goals, based on definition 3.3.

Definition 3.4 Let G and G' denote two different nodes in an SLD-tree τ. Let R be the computation rule used in τ. Then G' *covers* G iff

1. $R(G')$ and $R(G)$ are atoms with the same predicate
2. $R(G') >_{pr} R(G)$

Notice that G' covers G implies that G' is an ancestor of G.

We need one more piece of terminology.

Definition 3.5 Let G and G' denote two different nodes in an SLD-tree τ. We call G' the *youngest covering ancestor* of G iff

1. G' covers G
2. For any other node G'' such that G'' covers G, we have that G'' covers G'

We are now finally able to formulate the following algorithm:

Algorithm 3.6

Input
a definite program P
a definite goal $\leftarrow A$
Output
a finite SLD-tree τ for $P \cup \{\leftarrow A\}$
Initialisation
$\tau := \{(\leftarrow A,1)\}$
$Pr := \emptyset$
$Terminated := \emptyset$
$Failed := \emptyset$
For each recursive predicate p/n in P and for the derivation D in τ:
$S_{p,D} := \{1,\ldots,n\}$
While there exists a derivation D in τ such that $D \notin Terminated$ **do**
Let (G,i) name the leaf of D

Select the leftmost atom $p(t_1,\ldots,t_n)$ in G
satisfying the following condition:
  If p is recursive and there is
a youngest covering ancestor (G',j) of (G,i) in D
  then $|R(G')|_{p,S_{p,D}^{new}} > |p(t_1,\ldots,t_n)|_{p,S_{p,D}^{new}}$ where
    $S_{p,D}^{new} = S_{p,D} \setminus S_{p,D}^{remove}$ and
    $S_{p,D}^{remove} =$
    $\{a_k \in S_{p,D} \mid |p(t_1,\ldots,t_n)|_{p,\{a_k\}} > |R(G')|_{p,\{a_k\}}\}$
If such an atom $p(t_1,\ldots,t_n)$ can be found
then
  $R(G) := p(t_1,\ldots,t_n)$
  Let *Derive*(G,i) name the set of all derivation steps
  that can be performed
  If *Derive*$(G,i) = \emptyset$
  then
    Add D to *Terminated* and *Failed*
  else
    Let *Descend*$(R(G),i)$ name the set of
    all pairs $((R(G),i),(B\theta,j))$, where
      — B is an atom in the body of a clause
        applied in an element of *Derive*(G,i)
      — θ is the corresponding m.g.u.
      — j is the number of the corresponding
        descendent of (G,i)
    Expand D in τ with the elements of *Derive*(G,i)
    Add the elements of *Descend*$(R(G),i)$ to Pr
    For every newly created extension D' of D and
    for every recursive predicate q in P:
      if $q = p$ and (G,i) has a covering ancestor in D
      then $S_{q,D'} := S_{q,D}^{new}$
      else $S_{q,D'} := S_{q,D}$
else
  Add D to *Terminated*

Endwhile

We have the following theorem.

Theorem 3.7 Algorithm 3.6 terminates. If a definite program P and a definite goal $\leftarrow A$ are given as inputs, its output τ is a finite (possibly incomplete) SLD-tree for $P \cup \{\leftarrow A\}$.

Proof The theorem is an immediate consequence of proposition 3.1 in [Bruynooghe *et al.*, 1991a]. □

Example 3.8 The SLD-tree generated by algorithm 3.6 for the program and the query from example 2.2, are depicted in figure 1. ("reverse" has been abbreviated to "rev".)

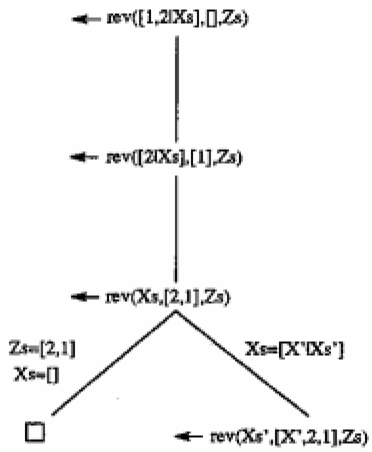

Figure 1: The SLD-tree for example 3.8.

4 Combining These Techniques

4.1 Introduction

In the previous section, we introduced an algorithm for the automatic construction of (incomplete) finite SLD-trees. In this section, we present sound and complete partial deduction methods, based on it. Moreover, these methods are guaranteed to terminate. The following example shows that this latter property is not obvious, even when termination of the basic unfolding procedure is ensured. We use the basic partial deduction algorithm from [Benkerimi and Lloyd, 1990], together with our unfolding algorithm.

Example 4.1 For the reverse program with accumulating parameter (see example 2.2 for the program and the starting query), an infinite number of (finite) SLD-trees is produced (see figure 2). This behaviour is caused by the constant generation of "fresh" body-literals which, because of the growing accumulating parameter, are not an instance of any atom that was obtained before.

In [Benkerimi and Lloyd, 1989], it is remarked that a solution to this kind of problems can be truncating atoms put into **A** at some fixed depth bound. However, this again seems to have an ad-hoc flavour to it, and we therefore devised an alternative method, described in the next section.

4.2 An algorithm for partial deduction

We first introduce some useful definitions and prove a lemma.

Definition 4.2 Let P be a definite program and p a predicate symbol of the language underlying P. Then a *pp'-renaming of P* is any program obtained in the following way:

- Take P together with a fresh—duplicate—copy of the clauses defining p.
- Replace p in the heads of these new clauses by some new (predicate) symbol p' (of the same arity as p).

- Replace p by p' in any number of goals in the bodies of (old and new) clauses.

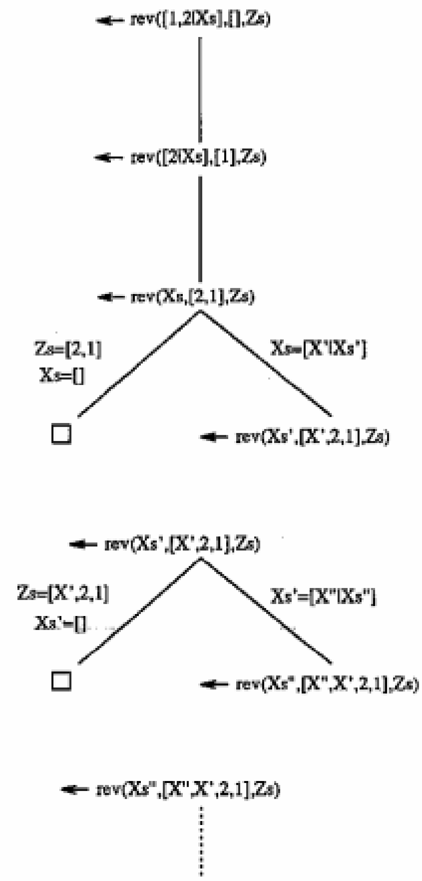

Figure 2: An infinite number of (finite) SLD-trees.

Lemma 4.3 Let P be a definite program and P_r a pp'-renaming of P. Let G be a definite goal in the language underlying P. Then the following hold:

- $P_r \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof There is an obvious equivalence between SLD-derivations and -trees for P and P_r. □

Definition 4.4 Let P be a definite program and p a predicate symbol of the language underlying P. Then the *complete pp'-renaming of* P is the pp'-renaming of P where p has been replaced by p' in all goals in the bodies of clauses.

Our method for partial deduction can then be formulated as the following algorithm.

Algorithm 4.5

Input
  a definite program P
  a definite goal $\leftarrow A = \leftarrow p(t_1, \ldots, t_n)$
   in the language underlying P
  a predicate symbol p', of the same arity as p,
   not in the language underlying P

Output
  a set of atoms $\mathbf{A}$
  a partial deduction P_r' of P_r,
   the complete pp'-renaming of P, wrt $\mathbf{A}$

Initialisation
  $P_r :=$ the complete pp'-renaming of P
  $\mathbf{A} := \{A\}$ and label A unmarked

While there is an unmarked atom B in $\mathbf{A}$ **do**
  Apply algorithm 3.6 with P_r and $\leftarrow B$ as inputs
  Let τ_B name the resulting SLD-tree
  Form P_{rB}, a partial deduction for B in P_r, from τ_B
  Label B marked
  Let $\mathbf{A}_B$ name the set of body literals in P_{rB}
  For each predicate q appearing in an atom in $\mathbf{A}_B$
   Let msg_q name an msg of all atoms having q
   as predicate symbol in $\mathbf{A}$ and $\mathbf{A}_B$
   If there is an atom in $\mathbf{A}$ having q as predicate
   symbol and it is less general than msg_q
    then remove this atom from $\mathbf{A}$
   If now there is no atom in $\mathbf{A}$ having q as
   predicate symbol
    then add msg_q to $\mathbf{A}$ and label it unmarked
  Endfor
Endwhile
Finally, construct the partial deduction P_r' of P_r wrt $\mathbf{A}$: Replace the definitions of the partially deduced predicates by the union of the partial deductions P_{rB} for the elements B of $\mathbf{A}$.

We illustrate the algorithm on our running example.

Example 4.6

complete renaming of the reverse program:
  reverse([],L,L).
  reverse([X|Xs],Ys,Zs) ← reverse'(Xs,[X|Ys],Zs).
  reverse'([],L,L).
  reverse'([X|Xs],Ys,Zs) ← reverse'(Xs,[X|Ys],Zs).

partial deduction for ←reverse([1,2|Xs],[],Zs):
  reverse([1,2],[],[2,1]).
  reverse([1,2,X|Xs],[],Zs) ← reverse'(Xs,[X,2,1],Zs).

partial deduction for ←reverse'(Xs,[X,2,1],Zs):
  reverse'([],[X,2,1],[X,2,1]).
  reverse'([X'|Xs],[X,2,1],Zs) ←
   reverse'(Xs,[X',X,2,1],Zs).

msg of reverse'(Xs,[X,2,1],Zs) and
  reverse'(Xs,[X',X,2,1],Zs): reverse'(Xs,[X,Y,Z|Ys],Zs)

```
partial deduction for ←reverse'(Xs,[X,Y,Z|Ys],Zs):
  reverse'([],[X,Y,Z|Ys],[X,Y,Z|Ys]).
  reverse'([X'|Xs],[X,Y,Z|Ys],Zs) ←
    reverse'(Xs,[X',X,Y,Z|Ys],Zs).
resulting set A:
  {reverse([1,2|Xs],[],Zs),reverse'(Xs,[X,Y,Z|Ys],Zs)}
resulting partial deduction:
  reverse([1,2],[],[2,1]).
  reverse([1,2,X|Xs],[],Zs) ← reverse'(Xs,[X,2,1],Zs).
  reverse'([],[X,Y,Z|Ys],[X,Y,Z|Ys]).
  reverse'([X'|Xs],[X,Y,Z|Ys],Zs) ←
    reverse'(Xs,[X',X,Y,Z|Ys],Zs).
```

We can prove the following interesting properties of algorithm 4.5.

Theorem 4.7 Algorithm 4.5 terminates.

Proof Due to space restrictions, we refer to [Martens and De Schreye, 1992]. □

Theorem 4.8 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let **A** be the (finite) set of atoms and $P_r{}'$ be the program output by algorithm 4.5. Then the following hold:

- **A** is independent.
- For any goal $G = \leftarrow A_1, \ldots, A_m$ consisting of atoms that are instances of atoms in **A**, $P_r{}' \cup \{G\}$ is **A**-covered.

Proof

- We first prove that **A** is independent.
  From the way **A** is constructed in the For-loop, it is obvious that **A** cannot contain two atoms with the same predicate symbol. Independence of **A** is an immediate consequence of this.
- To prove the second part of the theorem, let $P_r{}^*$ be the subprogram of $P_r{}'$ consisting of the definitions of the predicates in $P_r{}'$ upon which G depends. We show that $P_r{}^* \cup \{G\}$ is **A**-closed.
  Let A be an atom in **A**. Then the For-loop in algorithm 4.5 ensures there is in **A** a generalisation of any body literal in the computed partial deduction for A in $P_r{}'$. The **A**-closedness of $P_r{}^* \cup \{G\}$ now follows from the following two facts:
  1. $P_r{}'$ is a partial deduction of a program (P_r) wrt **A**.
  2. All atoms in G are instances of atoms in **A**.

□

Corollary 4.9 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let **A** be the set of atoms and $P_r{}'$ be the program output by algorithm 4.5. Let $G = \leftarrow A_1, \ldots, A_m$ be a goal in the language underlying P, consisting of atoms that are instances of atoms in **A**. Then the following hold:

- $P_r{}' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r{}' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof The corollary is an immediate consequence of lemma 4.3 and theorems 2.1 and 4.8. □

Proposition 4.10 Let P be a definite program and A be an atom used as inputs to algorithm 4.5. Let **A** be the set of atoms output by algorithm 4.5. Then $A \in$ **A**.

Proof A is put into **A** in the initialisation phase. From definition 4.4, it follows that no clause in P_r contains a condition literal with the same predicate symbol as A. Therefore, A will never be removed from **A**. □

This proposition ensures us that algorithm 4.5 does not suffer from the kind of specialisation loss mentioned in section 2.1: The definition of the predicate which appears in the query $\leftarrow A$, used as starting input for the partial deduction, will indeed be replaced by a partial deduction for A in P in the program output by the algorithm.

Finally, we have:

Corollary 4.11 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let $P_r{}'$ be the program output by algorithm 4.5. Then the following hold for any instance A' of A:

- $P_r{}' \cup \{\leftarrow A'\}$ has an SLD-refutation with computed answer θ iff $P \cup \{\leftarrow A'\}$ does.
- $P_r{}' \cup \{\leftarrow A'\}$ has a finitely failed SLD-tree iff $P \cup \{\leftarrow A'\}$ does.

Proof The corollary immediately follows from corollary 4.9 and proposition 4.10. □

Theorem 4.7 and corollary 4.11 are the most important results of this paper. In words, their contents can be stated as follows. Given a program and a goal, algorithm 4.5 produces a program which provides the same answers as the original program to the given query and any instances of it. Moreover, computing this (hopefully more efficient) program terminates in all cases.

5 Discussion and Conclusion

In [Lloyd and Shepherdson, 1991], important criteria ensuring soundness and completeness of partial deduction are introduced. In the present paper, we started from a recently proposed strategy for finite unfolding ([Bruynooghe *et al.*, 1991a]) and developed a procedure for partial deduction of definite logic programs. We proved this procedure produces programs satisfying the mentioned criteria and, in an important sense, showing the desired specialisation. Moreover, the algorithm terminates on all definite programs and goals.

The unfolding method as it is presented in section 3 was proposed in [Bruynooghe *et al.*, 1991a], but appears here for the first time in this detailed and automatisable form, specialised for object level programs. It tries to maximise unfolding while retaining termination. We know, however, of two classes of programs where the first goal is not achieved. First, meta programs require a somewhat more refined control of unfolding. This issue is addressed in [Bruynooghe *et al.*, 1991a]. We refer the interested reader to that paper (or to [Bruynooghe *et al.*, 1991b]) for further comments on this topic. Second, (datalog) programs where the information contained in constants appearing in the program text plays an important role, are not treated in a satisfactory way. Further research is necessary to improve the unfolding in this case. (A combination of our rule with the R_v computation rule seems promising.) As far as the used unfolding strategy does maximise unfolding, however, it probably diminishes or eliminates the need for dynamic renaming as proposed in [Benkerimi and Hill, 1989].

We now compare briefly algorithm 4.5 with the partial deduction procedure with static renaming presented in [Benkerimi and Lloyd, 1990]. First, we showed above that our procedure terminates for all definite programs and queries while the latter does not. The culprit of this difference in behaviour is (apart from the unfolding strategy used) the way in which msg's are taken. We do this predicatewise, while the authors of [Benkerimi and Lloyd, 1990] only take an msg when this is necessary to keep A independent. This may keep more specialisation (though only for predicates different from the one in the starting goal), but causes non-termination whenever an infinite, independent set A is generated (as illustrated in example 4.1). Observe, moreover, that we have kept a clear separation between the issues of control of unfolding and of ensuring soundness and completeness. The use of algorithm 3.6 — or further refinements (see above) — guarantees that all sensible unfolding — and therefore specialisation — is obtained. The way in which algorithm 4.5, in addition, ensures soundness and completeness, takes care that none of the obtained specialisation is undone. Therefore, it does not seem worthwhile to consider more than one msg per predicate. Note that one can even consider restricting the partial deduction to the predicate in the starting query and simply retaining the original clauses for all other predicates in the result program. This can perhaps be formalised as a partial deduction where only a 1-step trivial unfolding is performed for these predicates.

Next, the method in [Benkerimi and Lloyd, 1990] is formulated in a somewhat more general framework than the one presented here. A reformulation of the latter incorporating the concept of L-selectability and allowing more than one literal in the starting query seems straightforward. However, a generalisation to normal programs and queries and SLDNF-resolution while retaining the termination property, is not immediate. In e.g. [Benkerimi and Lloyd, 1990], it is proposed that during unfolding, negated calls can be executed when ground and remain in the resultant when non-ground. This of course jeopardises termination, since termination of "ordinary" ground logic program execution is not guaranteed in general. One solution is restricting attention to specific subclasses of programs (e.g. acyclic or acceptable programs, see [Apt and Bezem, 1990], [Apt and Pedreschi, 1990]). Another might be to use an adapted version of our unfolding criterion in the evaluation of the ground negative call, and to keep the latter one in the resultant whenever the SLD(NF)-tree produced is not a complete one. Yet a third way might be offered by the use of more powerful techniques related to constructive negation (see [Chan and Wallace, 1989]).

Finally, [Gallagher and Bruynooghe, 1990] presents another approach to partial deduction focusing both on soundness and completeness and on control of unfolding. The main difference is the control of unfolding by a condition based on maximal deterministic paths, where our approach is based on maximal data consumption, monitored through well-founded measures.

References

[Apt and Bezem, 1990] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 617–633, Jerusalem, June 1990. The MIT Press. Revised version in *New Generation Computing*, 9(3 & 4):335–364.

[Apt and Pedreschi, 1990] K. R. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 150–176. Springer-Verlag, November 1990.

[Benkerimi and Hill, 1989] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. Technical report, Department of Computer Science, University of Bristol, Great-Britain, 1989.

[Benkerimi and Lloyd, 1989] K. Benkerimi and J. W. Lloyd. A procedure for the partial evaluation of logic programs. Technical Report TR-89-04, Department of Computer Science, University of Bristol, Great-Britain, May 1989.

[Benkerimi and Lloyd, 1990] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 343–358. The MIT Press, October 1990.

[Bruynooghe *et al.*, 1991a] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In V. Saraswat and K. Ueda, editors, *Proceedings ILPS'91*, pages 117–131, October 1991.

[Bruynooghe *et al.*, 1991b] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. Technical Report CW-126, Departement Computerwetenschappen, K.U.Leuven, Belgium, March 1991.

[Chan and Wallace, 1989] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 299–318. MIT Press, 1989.

[Gallagher and Bruynooghe, 1990] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 732–746, Jerusalem, June 1990. Revised version in *New Generation Computing*, 9(3 & 4):305–334.

[Gallagher, 1986] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings ECAI'86*, pages 109–122, 1986.

[Komorowski, 1981] H. J. Komorowski. A specification of an abstract Prolog machine and its application to partial evaluation. Technical Report LSST69, Linkoping University, 1981.

[Komorowski, 1989] H. J. Komorowski. Synthesis of programs in the framework of partial deduction. Technical Report Ser.A, No.81, Departments of Computer Science and Mathematics, Abo Akademi, Finland, 1989.

[Levi and Sardu, 1988] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2 & 3), 1988.

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

[Martens and De Schreye, 1992] B. Martens and D. De Schreye. Sound and complete partial deduction with unfolding based on well-founded measures. Technical Report CW-137, Departement Computerwetenschappen, K.U.Leuven, Belgium, January 1992.

[Safra and Shapiro, 1986] S. Safra and E. Shapiro. Meta interpreters for real. In *Information Processing 86*, pages 271–278, 1986.

[Sahlin, 1990] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 377–398, 1990.

[Sterling and Beer, 1986] L. Sterling and R. D. Beer. Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proceedings ILPS'86*, pages 20–27. IEEE Comp. Society Press, 1986.

[Sterling and Beer, 1989] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *Journal of Logic Programming*, pages 163–178, 1989.

[Takeuchi and Furukawa, 1986] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[Venken and Demoen, 1988] R. Venken and B. Demoen. A partial evaluation system for Prolog : Some practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[Venken, 1984] R. Venken. A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In *Proceedings ECAI'84*, pages 91–100. North-Holland, 1984.