Efficient Local Unfolding with Ancestor Stacks[∗]

GERMÁN PUEBLA[1]

ELVIRA ALBERT[2]

MANUEL HERMENEGILDO[1,3]

[1]*School of Computer Science, Technical University of Madrid*
E28660-Boadilla del Monte, Madrid, Spain. E-mail: `german@fi.upm.es`, `herme@fi.upm.es`

[2]*School of Computer Science, Complutense University of Madrid*
E28040-Profesor José García Santesmases, s/n, Madrid, Spain. E-mail: `elvira@sip.ucm.es`

[3]*Madrid Institute for Advanced Studies in Software Development Technology,*
IMDEA Software. E-mail: `manuel.hermenegildo@imdea.org`

submitted 25 October 2005; revised 3 March 2006; accepted XXXX

Abstract

The most successful *unfolding rules* used nowadays in the partial evaluation of logic programs are based on *well quasi orders* (wqo) applied over (covering) *ancestors*, i.e., a subsequence of the atoms selected during a derivation. Ancestor (sub)sequences are used to increase the specialization power of unfolding while still guaranteeing termination and also to reduce the number of atoms for which the wqo has to be checked. Unfortunately, maintaining the structure of the ancestor relation during unfolding introduces significant overhead. We propose an efficient, practical *local* unfolding rule based on the notion of covering ancestors which can be used in combination with a wqo and allows a stack-based implementation without losing any opportunities for specialization. Using our technique, certain non-leftmost unfoldings are allowed as long as local unfolding is performed, i.e., we cover depth-first strategies. To deal with practical programs, we propose assertion-based techniques which allow our approach to treat programs that include (Prolog) built-ins and external predicates in a very extensible manner, for the case of leftmost unfolding. Finally, we report on our implementation of these techniques embedded in a practical partial evaluator, which shows that our techniques, in addition to dealing with practical programs, are also significantly more efficient in time and somewhat more efficient in memory than traditional tree-based implementations.

KEYWORDS: Partial Evaluation, Partial Deduction, Logic Programming, Prolog, SLD semantics, Local Unfolding.

1 Introduction

The main purpose of *partial evaluation* (see (Jones et al. 1993) for a general text on the area) is to specialize a given program w.r.t. part of its input data—hence

∗ A preliminary version of this work appeared in the Post-proceedings of LOPSTR'04, LNCS 3573, Springer-Verlag, 2005.

it is also known as *program specialization.* Essentially, partial evaluators are non-standard interpreters which evaluate expressions while enough information is available and residualize them otherwise. The partial evaluation of logic programs is usually known as *partial deduction* (Lloyd and Shepherdson 1991; Gallagher 1993). Informally, the partial deduction algorithm proceeds as follows. Given an input program and a set of atoms, the first step consists in applying an *unfolding rule* to compute finite (possibly incomplete) SLD trees for these atoms. This step returns a set of *resultants* (or residual rules), i.e., a program, associated to the root-to-leaf derivations of these trees. Then, an *abstraction operator* is applied to properly add the atoms in the bodies of resultants to the set of atoms to be partially evaluated. The abstraction phase yields a new set of atoms, some of which may in turn need further evaluation and, thus, the process is iteratively repeated while new atoms are introduced. The number of such new atoms which can be introduced can in general be unbounded. The termination of the partial deduction process is ensured by two control issues. Following the terminology of (Gallagher 1993), the so-called *local* control defines an unfolding rule which determines how to construct finite SLD trees. The *global* control defines an abstraction operator which guarantees that the number of new atoms is kept finite. Termination of the partial deduction algorithm involves ensuring termination both at the local and global levels. We refer to (Leuschel and Bruynooghe 2002) for a survey on both control issues. This article is centered on the local control, namely on the development of a practical, efficient unfolding rule. The techniques we will propose for local control can be used in combination with any global control strategy.

We believe that two factors limiting the general uptake of partial deduction are: 1) the relative inefficiency of the partial deduction method, and 2) the complications brought about by the treatment of real programs. Indeed, the integration of powerful strategies in the unfolding rule —like the use of wqos combined with the ancestor relation— can introduce a significant cost both in time and memory consumption of the specialization process. Regarding the treatment of real programs which include external predicates, non-declarative features, etc., the complications range from how to identify which predicates include these non-declarative features (ad-hoc but difficult to maintain tables are often used in practice for this purpose) to how to deal with such predicates during partial deduction. Also, the optimal treatment of these predicates during partial deduction often requires information which can only be available at partial deduction time if a global analysis of the program is performed. Our main objective in this work is to propose some novel solutions to these issues.

State-of-the-art partial evaluators integrate terminating unfolding rules for local control based on wqos, like homeomorphic embedding (Kruskal 1960; Leuschel and Bruynooghe 2002) which can obtain very powerful optimizations. Moreover, they allow performing the ordering comparisons over *subsequences* of the full sequence of the selected atoms. In particular, the use of *ancestors* for refining sequences of visited atoms, originally proposed in (Bruynooghe et al. 1992), greatly improves the specialization power of unfolding while still guaranteeing termination and also reduces the length of the sequences for which the embedding order for the new atoms has to be checked. Unfortunately, having to maintain dependency information for the individual atoms in each derivation during the generation of SLD trees has turned out to introduce overheads which seem to cancel out the theoretical efficiency gains expected. In order to address this issue, in this article, we introduce *ASLD resolution* as the basis for an efficient, stack-based implementation technique of a local unfolding rule relying on the notion of covering ancestors. Our technique can significantly reduce the overhead incurred by the use of covering ancestors without losing any opportunities for specialization. We outline as well a generalization that allows certain non-leftmost unfoldings with the same assurances.

In order to deal with real programs that include (Prolog) built-ins and external predicates, we extend ASLD resolution and the ancestor-based local unfolding rule to handle these predicates by relying on assertion-based techniques (Puebla et al. 2000). The use of assertions provides *extensibility* in the sense that users and developers of partial evaluators can deal with new external predicates during partial evaluation by just adding the proper assertions to these predicates —without having to maintain ad-hoc tables or modifying the partial evaluator itself. We report on an implementation of our technique in a practical, state-of-the-art partial evaluator, embedded in a production compiler which uses assertions and global analysis extensively (the `Ciao` compiler (Bueno et al. 2004) and, specifically, its preprocessor `CiaoPP` (Hermenegildo et al. 2005)). We believe that our experimental results provide evidence that our technique pays off in practice and can thus contribute to the practicality of state-of-the-art partial evaluation techniques.

An important observation is that the techniques that we propose in this article to control the unfolding process are useful in the context of *online* partial evaluation. Traditionally, two approaches to partial evaluation have been considered, *online* and *offline* partial evaluation (see (Leuschel et al. 2004; Leuschel and Bruynooghe 2002)). In online partial evaluation all control decisions are taken on the fly during the specialization phase, by keeping track of the specialization history (e.g., the ancestor subsequences). In the offline approach, all control decisions are taken before the specialization phase proper. These control decisions are based on abstract descriptions of the data instead of the actual data. The control strategy is usually represented as program annotations which are the sole decision criteria for control of the partial evaluator. For instance, regarding local control, an annotation can explicitly indicate that an atom should not be unfolded. Regarding global control, annotations typically specify for each call which arguments have to be generalised away (i.e., replaced by variables). Such annotations are generated automatically in some partial evaluators by a *binding-time analysis* (Craig et al. 2004), while in other partial evaluators they are manually provided by the user, either in part or in full. The advantages of the offline approach are that, once all control annotations are available, partial evaluation is quite simple and efficient. On the other hand, online partial evaluation while usually less efficient, it tends to have more powerful control strategy since control decisions are based on actual data instead of abstract descriptions of data. In principle, one could argue that both approaches are equally powerful (see (Christensen and Glück 2004)) and that the offline approach can be more appropriate if the output of a global program analysis is available, while online partial evaluators usually only consider local, runtime information. In this work, we are interested in proposing novel techniques which help improve the efficiency of online partial evaluation.

The structure of the article is as follows. Section 2 presents some required background on local control during partial deduction. Section 3 shows by means of an example why using ancestors is needed. Section 4 presents ASLD resolution as the basis for an efficient unfolding rule based on ancestors which allows a stack-based implementation. Section 5 extends the unfolding techniques to the case of external predicates. Section 6 presents some experimental results which compare the performance of different unfolding strategies with several implementations. Finally, Section 7 discusses some related work and concludes.

2 Background

We assume some basic knowledge on the terminology of logic programming. See for example (Lloyd 1987) for details.

Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms.

We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$), and $\sigma(X) = X$ for all other variables X. Given an atom A, $\theta(A)$ denotes the application of substitution θ to A. Given two substitutions θ_1 and θ_2, we denote by $\theta_1\theta_2$ their composition. The identity substitution is denoted by *id*.

A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$.

2.1 Basics of partial deduction

The concept of *computation rule* is used to select an atom within a goal for its evaluation.

Definition 1 (*computation rule*)
A *computation rule* is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the *selected* atom in G.

The operational semantics of definite programs is based on derivations.

Definition 2 (*derivation step*)
Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G' is *derived* from G and C via $\mathcal{R}$ if the following conditions hold:

$$\begin{aligned} \theta &= mgu(A_R, H) \\ G' \text{ is the goal } &\leftarrow \theta(B_1, \ldots, B_m, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \end{aligned}$$

The definition above differs from standard formulations (such as that in (Lloyd 1987)) in that the atoms newly introduced in G' are not placed in the same position where the selected atom A_R used to be, but rather they are placed to the left of any atom in G. For definite programs, this is correct since goals are conjunctions, which enjoy the commutative property. This modification will become instrumental to the operational semantics we propose in forthcoming sections. This is not true though for programs with extra logical predicates, as we will discuss in Section 5. Also, it is well-known that changing the atom's positions might not preserve finite failure. Although our general notion of resolution allows reordering the atoms, in a practical system, we can allow only leftmost unfolding and still obtain significant improvements (as will be explained in Section 5).

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence of *computed answer substitutions* $\theta_1, \theta_2, \ldots$ (or mgus) such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. If G_i is of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ and $G_{i+1} \equiv \theta(B_1, \ldots, B_m, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k)$ is derived from G_i (as stated in Definition 2), we say that each atom $\theta(A_i)$ with $i = 1, \ldots, R-1, R+1, \ldots, k$ is the *instance originating* from A_i. Finally, we say that the SLD derivation is composed of the *subsequent* goals $G_0, G_1, G_2, \ldots$.

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees.* A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the *computed answer* for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

In order to compute a partial deduction (Lloyd and Shepherdson 1991), given an input program and a set of atoms, the first step consists in applying an *unfolding rule* to compute finite (possibly incomplete) SLD trees for these atoms. Then, a set of *resultant*s or residual rules are systematically extracted from the SLD trees.[1]

Definition 3 (*unfolding rule*)
Given an atom A, an *unfolding rule* computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with (a composed) computed answer substitution θ_i for $i = 1, \ldots, n$ whose associated *resultants* are $\theta_i(A) \leftarrow G_i$.

A *partial evaluation* for the initial atom is then defined as the set of resultants, i.e., a program, associated to the root-to-leaf derivations for the computed SLD tree. The partial evaluation for a set of atoms is defined as the union of the partial evaluations for each atom in the set. We refer to (Leuschel and Bruynooghe 2002) for details.

[1] Let us note that the definition of a partial deduction *algorithm* requires, in addition to an unfolding rule, the so-called global control level (see Section 1).

2.2 Termination of local control

In order to ensure the local termination of the partial deduction algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-quasi orderings (wqo) (Sørensen and Glück 1995; Leuschel 1998) are broadly used in the context of on-line partial evaluation techniques.

It is well known that the use of wqos allows the definition of *admissible* sequences which are always finite. Intuitively, a sequence of elements $s_1, s_2, \ldots$ in S is called *admissible with respect to an order* $\leq_S$ (Bruynooghe et al. 1992) iff there are no $i < j$ such that $s_i \leq_S s_j$. The next definition captures this idea.

Definition 4 (*admissible –wqo*)
Let $(A_1, \ldots, A_n)$ be a sequence of atoms and A be a new atom to be added to the sequence. Let $\leq_S$ be a wqo. We denote by $Admissible(A, (A_1, \ldots, A_n), \leq_S)$, with $n \geq 0$ the truth value of the expression $\forall A_i,\ i \in \{1, \ldots, n\}\ : A \not\geq_S A_i$.

Given a derivation $G_1, G_2,\ \cdots, G_{n+1}$ in order to decide whether to evaluate G_{n+1} or not, we check that the selected atom in G_{n+1} is not strictly greater or equal to any previous *comparable* selected atom (Leuschel 2002b). Observe that the ancestor test is only applied on comparable atoms, i.e., ancestor atoms with the same predicate symbol. This corresponds to the original notion of covering ancestors (Bruynooghe et al. 1992). Note that $A_1, \ldots, A_n$ in the above definition refer to the selected atoms in $G_1, \ldots, G_n$ and A refers to the selected atom in G_{n+1}.

Among the wqo, the *homeomorphic embedding* ordering (Kruskal 1960) has proved to be very powerful in practice. We recall the definition of homeomorphic embedding, which can be found for instance in Leuschel's work (Leuschel 1998).

Definition 5 ($\trianglelefteq$)
Given two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, we say that B *embeds* A, written $A \trianglelefteq B$, if $t_i \trianglelefteq s_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over terms, also written $\trianglelefteq$, is defined by the following rules:

1. $Y \trianglelefteq X$ for all variables X, Y.
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i, $1 \leq i \leq n$.

Informally, atom t_1 *embeds* atom t_2 if t_2 can be obtained from t_1 by deleting some operators, e.g., $\underline{\texttt{f}}(\texttt{g}(\underline{\texttt{A}},\texttt{B}),\underline{\texttt{h}}(\underline{\texttt{C}},\texttt{s}(\underline{\texttt{D}})))$ embeds $\texttt{f}(\texttt{A},\texttt{h}(\texttt{C},\texttt{D}))$.

2.3 Covering ancestors

State-of-the-art unfolding rules allow performing ordering comparisons over *subsequences* of the full sequence of the selected atoms of a derivation by organizing atoms in a *proof tree* (Bruynooghe 1991), achieving further specialization in many cases while still guaranteeing termination. To do so, they maintain dependencies over the selected atoms which are chosen in such a way that only a subsequence of such selected atoms needs to be considered. The essence of the most advanced techniques is based on the notion of *covering ancestors* (Bruynooghe et al. 1992).

```
qsort([],R,R).
qsort([X|L],R,R2) :-
   partition(L,X,L1,L2),
   qsort(L2,R1,R2),
   qsort(L1,R,[X|R1]).
```

```
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-
   E =< C,
   partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :-
   E > C,
   partition(R,C,Left,Right1).
```

Fig. 1. A quick-sort program

Definition 6 (*ancestor relation*)
Given a derivation step and A_R, B_i, $i = 1, \ldots, m$ as in Definition 2, we say that A_R is the *parent* of the instance of B_i, $i = 1, \ldots, m$, in the goal and in each subsequent goal where the instance originating from B_i appears. The *ancestor* relation is the transitive closure of the parent relation.

The important observation is that a derivation can contain selected subgoals which are indeed part of a different branch in the proof tree.

Given an atom A and a derivation D, we denote by $Ancestors(A, D)$ the sequence of (comparable) ancestors of A in D as defined in Definition 6. It captures the dependency relation implicit within a *proof tree.*

It has been proved (Bruynooghe et al. 1992) that any infinite derivation must have at least one inadmissible *covering ancestor* sequence, i.e., a subsequence of the atoms selected during a derivation. Therefore, it is sufficient to check the selected ordering relation $\leq_S$ over the covering ancestor subsequences in order to detect inadmissible derivations.

Definition 7 (*safe step*)
An SLD step is *safe* with respect to a wqo if the covering ancestor sequence of the selected atom is admissible with respect to that order.

The above definition is extended to derivations as follows.

Definition 8 (*safe derivation*)
An SLD derivation is *safe* with respect to a wqo if all covering ancestor sequences of the selected atoms are admissible with respect to that order.

Otherwise, the SLD derivation is considered *unsafe.*

3 The Usefulness of Ancestors

We now illustrate some of the ideas discussed so far and, specially, the relevance of ancestor tracking, through an example. Our running example is the program in Figure 1, which implements the well known quick-sort algorithm, “`qsort`”, using difference lists. Given an initial atom of the form `qsort(`*List*`,`*Result*`,`*Cont*`)`, where *List* is a list of numbers, the algorithm returns in *Result* a sorted difference list which is a permutation of *List* and such that its continuation is *Cont.* For example, for the query `qsort([1,1,1],L,[])`, the program should compute `L=[1,1,1]`, constructing a finite SLD tree. Notice that, in general, if the input arguments to a

$$\begin{tikzcd}
\mathbf{1}.\underline{\texttt{qs}([1,1,1],\texttt{R},[])^{\{\}}} \arrow[d] \\
\mathbf{2}.\underline{\texttt{p}([1,1],1,\texttt{L1},\texttt{L2})^{\{1\}}}, \mathbf{3}.\texttt{qs}(\texttt{L2},\texttt{R1},[])^{\{1\}}, \mathbf{4}.\texttt{qs}(\texttt{L1},\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d, "\{\texttt{L1}\mapsto[1|\texttt{L}]\}"] \\
\mathbf{5}.\underline{1 =< 1}^{\{1,2\}}, \mathbf{6}.\texttt{p}([1],1,\texttt{L},\texttt{L2})^{\{1,2\}}, \mathbf{3}.\texttt{qs}(\texttt{L2},\texttt{R1},[])^{\{1\}}, \mathbf{4}.\texttt{qs}([1|\texttt{L}],\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d] \\
\mathbf{6}.\boxed{\underline{\texttt{p([1],1,L,L2)}}}^{\{1,2\}}, \mathbf{3}.\texttt{qs}(\texttt{L2},\texttt{R1},[])^{\{1\}}, \mathbf{4}.\texttt{qs}([1|\texttt{L}],\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d, "\{\texttt{L}\mapsto[1|\texttt{L}']\}"] \arrow[ddddd, dashed, bend right] \\
\mathbf{7}.\underline{1 =< 1}^{\{1,2,6\}}, \mathbf{8}.\texttt{p}([],1,\texttt{L}',\texttt{L2})^{\{1,2,6\}}, \mathbf{3}.\texttt{qs}(\texttt{L2},\texttt{R1},[])^{\{1\}}, \mathbf{4}.\texttt{qs}([1,1|\texttt{L}'],\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d] \\
\mathbf{8}.\underline{\texttt{p}([],1,\texttt{L}',\texttt{L2})^{\{1,2,6\}}}, \mathbf{3}.\texttt{qs}(\texttt{L2},\texttt{R1},[])^{\{1\}}, \mathbf{4}.\texttt{qs}([1,1|\texttt{L}'],\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d, "\{\texttt{L}'\mapsto[],\texttt{L2}\mapsto[]\}"] \\
\mathbf{3}.\underline{\texttt{qs}([],\texttt{R1},[])^{\{1\}}}, \mathbf{4}.\texttt{qs}([1,1],\texttt{R},[1|\texttt{R1}])^{\{1\}} \arrow[d, "\{\texttt{R1}'\mapsto[]\}"] \\
\mathbf{4}.\underline{\texttt{qs}([1,1],\texttt{R},[1])^{\{1\}}} \arrow[d] \\
\mathbf{9}.\boxed{\underline{\texttt{p([1],1,L1',L2')}}}^{\{1,4\}}, \mathbf{10}.\texttt{qs}(\texttt{L2}',\texttt{R1}',[1])^{\{1,4\}}, \mathbf{11}.\texttt{qs}(\texttt{L1}',\texttt{R},[1|\texttt{R1}'])^{\{1,4\}}
\end{tikzcd}$$

Fig. 2. Derivation with Ancestor Annotations

program are not sufficiently instantiated, the corresponding SLD tree can be infinite and/or contain incomplete derivations.

Consider now Figure 2, which presents an incomplete SLD derivation for our quick-sort program and the query `qsort([1,1,1],R,[])` using a leftmost unfolding rule. For conciseness, predicates `qsort` and `partition` are abbreviated as `qs` and `p`, respectively in the figure. Note that each atom is labeled with a number (an identifier) for future reference[2] and a superscript which contains the list of ancestors of that atom. Let us assume that we use the homeomorphic embedding order (Leuschel 1998) as wqo. If we check admissibility w.r.t. the full sequence of atoms, i.e., we do not use the ancestor relation, the derivation will stop when atom number **9**, i.e., $\texttt{p}([1],1,\texttt{L}',\texttt{L2}')$, is found for the second time. The reason is that this atom is greater or equal to the atom number **6** which was selected in the third step, indeed, they are equal modulo renaming.[3]

This unfolding rule is too conservative, since the process can proceed further without risking termination (in fact, the SLD tree for a leftmost computation rule for the example query is finite and thus the query can safely be fully unfolded). The crucial point is that the execution of atom number **9** does not depend on atom

[2] By abuse of notation, we keep the same number for each atom throughout the derivation although it may be further instantiated (and thus modified) in subsequent steps. This will become useful for continuing the example later.

[3] Let us note that the two calls to the builtin predicate `=<` which appear in the derivation can be executed since the arguments are properly instantiated. However, they have not been considered in the admissibility test since these calls do not endanger the termination of the derivation, as we will discuss in Section 5.

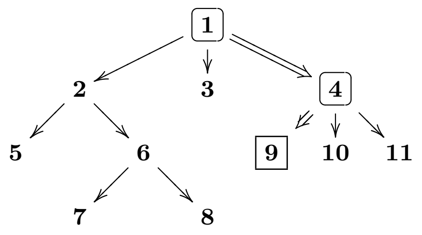

Fig. 3. Proof tree for the example

number **6** (and, actually, the unfolding of **6** has been already *completed* when atom number **9** is being considered for unfolding). In order to illustrate this, consider Figure 3 which shows the proof tree associated to this derivation. Nodes are labeled with the numbers assigned to each atom, instead of the atoms themselves. Note that, in order to decide whether or not to evaluate atom number **9**, it is only necessary to check that it is not greater or equal to atoms **4** and **1**, i.e., than those which are its *ancestors* in the proof tree. On the other hand, and as we saw before, if the full derivation is considered instead, as in Figure 2, atom **9** will be compared also with atom **6** concluding imprecisely that the derivation may not be safe.

Despite their obvious relevance, unfortunately the practical applicability of unfolding rules based on the notion of covering ancestor is threatened by the overhead introduced by the implementation of this notion. A naive implementation of the notion of ancestor keeps —for each atom— the list of its ancestors, as it is depicted in Figure 2 by using superscripts. This implementation is relatively efficient in time but presents a high overhead in memory consumption. Our experiments show that the partial evaluator can run out of memory even for simple examples. A more reasonable implementation maintains the proof tree as a global structure. In a symbolic language, this greatly reduces memory consumption but the cost of traversing the tree for retrieving the ancestors of each atom introduces a significant slowdown in the partial evaluation process. We argue that our implementation technique is efficient in time and space, overcoming the above limitations.

4 An Efficient Implementation for Local Unfolding

In this section, we first define the notion of local computation rule. We then introduce ASLD resolution, a modification of SLD which incorporates ancestor stacks and which is the basis of our efficient implementation. Interestingly, we then impose the local condition to the computation rule in order to ensure accurate results for ASLD resolution.

4.1 A local computation rule

Our definition of *local unfolding* is based on the notion of *ancestor depth.*

Definition 9 (*ancestor depth*)
Given an SLD derivation $D = G_0, \ldots, G_m$ with $G_m = \leftarrow A_1, \ldots, A_k$, $k \geq 1$, the

ancestor depth of A_i for $i = 1, \ldots, k$, denoted $\mathit{depth}(A_i, D)$ is the cardinality of the ancestor relation for A_i in D.

Intuitively, the ancestor depth of an atom in a goal is the depth at which this atom is located in the proof tree associated to the derivation.

Definition 10 (*local computation rule*)
A computation rule $\mathcal{R}$ is *local* if $\forall D = G_0, \ldots, G_n$ such that $G_i = \leftarrow A_{i1}, \ldots, A_{im_i}$ for $i = 0, .., n$, it holds that $\mathit{depth}(\mathcal{R}(G_i), D) \geq \mathit{depth}(A_{ij}, D) \quad \forall j = 1, \ldots, m_i$.

Intuitively, a computation rule is local if it always selects one of the atoms which is deepest in the proof tree for the derivation. As a result, local computation rules traverse proof trees in a depth-first fashion, though not necessarily left to right nor in any other fixed order. Thus, in principle, in order to implement a local computation rule we need to record (part of) the derivation history (i.e., its proof tree). Note that the computation rule used in most implementations of logic programming languages, such as Prolog, always selects the leftmost atom. This computation rule, often referred to as leftmost computation rule, is clearly a local computation rule. Selecting the leftmost atom in all goals guarantees that the selected atom is of maximal depth within the proof tree as it is traversed in a depth-first fashion — without the need of storing any history about the derivation.

It is interesting to note that we can allow more flexible computation rules which are not necessarily local while still ensuring termination at the cost of no accuracy assurance. A more detailed discussion on this will appear at the end of Section 4.3.

An instrumental observation in our approach is that the proof trees which are used in order to capture the ancestor relation can be seen as (a simplified version of) the *activation trees* (Aho et al. 1986) used in compiler theory for representing program executions, by simply regarding selected atoms as procedure calls. The nodes in such activation trees are *activation records*, which contain information about local variables, the current program counter, the return address, etc. of the corresponding call. Nested subprogram calls result in children activation records. In the vast majority of programming languages, execution of a program corresponds to traversing activation trees in a depth-first fashion. Therefore, for efficiency, rather than maintaining the whole activation tree in memory, run-time systems for execution of such programming languages feature a *call stack* where activation records are stored. This call stack contains exactly the sequence of activation records which are active at any point in time during the execution. This implementation strategy requires that new activation records be added to the call stack as soon as a new subprogram is called and that the top of the call stack is popped when the execution of a subprogram returns.

Our idea then is to maintain during unfolding an *ancestor stack*, whose elements are the ancestors of a goal, instead of a full proof tree. The advantages of this are clear: since the ancestor stack corresponds to a single branch in the proof tree from the current selected atom to all its ancestors in the proof tree, maintaining it should offer significant performance improvements both in terms of memory and time efficiency. As in the case of control stacks, in order to compute ancestor stacks we need to determine exactly when each ancestor should be pushed to and popped from the ancestors stack. The first part is relatively simple: any resolution step requires pushing its associated selected atom. The second part, i.e., popping elements from the stack, is more complicated since we need to know when the computation of the associated call (or subprogram) is finished. In logic programming terminology this corresponds to determining the (partial) success states for all atoms in the derivation. In principle, success states for individual atoms are not observable in SLD resolution, except for the top-level query. As a result, and as we discuss below, some changes in the operational semantics will be needed in order to make this information explicit.

Another important observation which we exploit in this paper is that the idea of using a stack for storing the active part of a tree does not need to be restricted to leftmost computation and it works equally well as long as the computation rule is local. Indeed, sibling atoms, i.e., with the same ancestor depth, can be selected in any order and the idea of using an ancestor stack still applies.

4.2 ASLD Resolution: SLD resolution with ancestor stacks

We now propose an easy-to-implement modification to SLD resolution as presented in Section 2 in which success states for all internal calls are observable —and where the control word is available at each state. We will refer to this resolution as SLD resolution with ancestor stacks, or *ASLD* for short. The proposed modification involves 1) augmenting goals with an *ancestor stack*, which at each stage of the computation contains the control word of the derivation, which corresponds to *the ancestors of the next atom which will be selected for resolution*, and 2) adding pseudo-atoms to the goals used during resolution which mark a *scope* (i.e., it separates groups of atoms which are at different depth in the proof tree). In particular, we use the pseudo-atom $\uparrow$ (read as "pop") to indicate the end of a depth scope, i.e., after it we move up in the proof tree. It is guaranteed not to clash with any existing predicate name. And its purpose is twofold: 2.1) when a mark is leftmost in a goal, it indicates that the current state corresponds to the success state for the call which is now on top of the ancestor stack, i.e., the call is completed, and the atom on top of the ancestor stack should be popped; 2.2) the atoms within the scope of the leftmost mark have maximal ancestor depth and thus a local unfolding strategy can be easily defined in the presence of these pseudo-atoms.

The following two definitions present the derivation rules in our ASLD semantics. Now, a state S is a tuple of the form $\langle G \mid AS \rangle$ where G is a goal and AS is an ancestor stack (or *stack* for short). The stack will keep track of the ancestor atoms that the new selected atoms need to be compared to (by means of the wqo being used). Thus the stack will be instrumental in being able to stop a derivation as soon as termination of the process can no longer be guaranteed by the wqo being used. To handle such stacks, we will use the usual stack operations: empty, which returns an empty stack, $\mathsf{push}(AS, \mathit{Item})$, which pushes Item onto the stack AS, and $\mathsf{pop}(AS)$, which pops an element from AS. In addition, we will use the operation

$\mathsf{contents}(AS)$, which returns the sequence of atoms contained in AS in the order in which they would be popped from the stack AS and leaves AS unmodified.

Definition 11 (*derive*)
Let $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ be a goal with $A_1 \neq \uparrow$. Let $S = \langle G \,\rule{1pt}{6pt}\, AS \rangle$ be a state and AS be a stack. Let $\leq_S$ be a wqo. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$ with $A_R \neq \uparrow$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause. Then $S' = \langle G' \,\rule{1pt}{6pt}\, AS' \rangle$ is *derived* from S and C via $\mathcal{R}$ if the following conditions hold:

$$\begin{array}{rr} & Admissible(A_R, \mathsf{contents}(AS), \leq_S) \\ & \theta = mgu(A_R, H) \\ G' \text{ is the goal} & \leftarrow \theta(B_1, \ldots, B_m, \uparrow, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \\ & AS' = \mathsf{push}(AS, A_R) \end{array}$$

The **derive** rule behaves as the one in Definition 2 but in addition: i) the mark $\uparrow$ "pop" is added to the goal, and ii) a copy of A_R is pushed onto the ancestor stack. As before, the **derive** rule is non-deterministic if several clauses in P unify with the atom A_R. However, in contrast to Definition 2, this rule can only be applied to an atom different from $\uparrow$ if 1) the leftmost atom in the goal is not a $\uparrow$ mark, and 2) the current selected atom A_R together with its ancestors do constitute an admissible sequence. If 1) holds but 2) does not, this derivation is stopped and we refer to such a derivation as *inadmissible* or unsafe (see Definition 8).

Definition 12 (*pop-derive*)
Let $G = \leftarrow A_1, \ldots, A_k$ be a goal with $A_1 = \uparrow$. Let $S = \langle G \,\rule{1pt}{6pt}\, AS \rangle$ be a state and AS be a stack. Then $S' = \langle G' \,\rule{1pt}{6pt}\, AS' \rangle$ with $G' = \leftarrow A_2, \ldots, A_k$ and $AS' = \mathsf{pop}(AS)$ is *pop-derived* from S.

The **pop-derive** rule is used when the leftmost atom in the resolvent is a $\uparrow$ mark. Its effect is to eliminate from the ancestor stack the topmost atom, which is guaranteed not to belong to the ancestors of any selected atom in any possible continuation of this derivation.

Note that *derive* steps w.r.t. a clause which is a fact are always followed by a *pop-derive* and thus they can be optimized by not pushing the selected atom A_R onto the stack and not including a $\uparrow$ mark into the goal which would immediately pop A_R from the stack. They have been also optimized in the implementation described in Section 6. Next, we present the following rule **derive-fact** with such an optimization, although we do not use it for our formal developments in Section 4.3. Indeed, its inclusion in the semantics would require that rule **derive** is only applied if $m > 0$.

Definition 13 (*derive-fact*)
Let $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ be a goal with $A_1 \neq \uparrow$. Let $S = \langle G \,\rule{1pt}{6pt}\, AS \rangle$ be a state and AS be a stack. Let $\leq_S$ be a wqo. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$ with $A_R \neq \uparrow$. Let $C = H.$ be a renamed apart fact. Then $S' = \langle G' \,\rule{1pt}{6pt}\, AS \rangle$

is *derived* from S and C via $\mathcal{R}$ if the following conditions hold:

$$\begin{aligned} & \mathit{Admissible}(A_R, \mathsf{contents}(AS), \leq_S) \\ & \theta = mgu(A_R, H) \\ G' \text{ is the goal } & \leftarrow \theta(A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \end{aligned}$$

Computation for a query G starts from the state $S_0 = \langle G \mid \mathsf{empty}\rangle$. Given a non-empty derivation D, we denote by *curr_goal(D)* and *curr_ancestors*(D) the goal and the stack in the last state in D, respectively. At each step of a derivation D at most one rule, either **derive**, **derive-fact** or **pop-derive**, can be applied.

Example 1
Figure 4 illustrates the ASLD derivation corresponding to the derivation with explicit ancestor annotations of Figure 2. Sometimes, rather than writing the atoms themselves, we use the same numbers assigned to the corresponding atoms in Figure 2. By abuse of notation, we again always use the same number assigned to an atom although further instantiation is performed. The stack contains the list of atoms exactly in the instantiation state they have when they are pushed in the stack. Each step has been appropriately labeled with the applied derivation rule. Although rule *external-derive* has not been presented yet, we can just assume that the code for the external predicate `=<` is available and has the expected behavior.

It should be noted that, in the last state, the stack contains exactly the ancestors of `partition([1],1,L1',L2')`, i.e., the atoms **4** and **1**, since the previous calls to `partition` have already finished and thus their corresponding atoms have been popped off the stack. Thus, the admissibility test for `partition([1],1,L1',L2')` succeeds, and unfolding can proceed further without risking termination. Indeed, the derivation can be totally unfolded, which results in the following (optimal) partial evaluation in which all input data have been satisfactorily consumed

$$\mathtt{qsort}([1,1,1],[1,1,1],[]).$$

Finally, since the goals obtained by ASLD resolution may contain atoms of the form $\uparrow$, resultants are cleaned up before being transferred to the global control level or during the code generation phase by simply eliminating all atoms of the form $\uparrow$.

It is easy to see that for each ASLD derivation D_S there is a corresponding SLD derivation D with the same computed answer and the same goal without the $\uparrow$ atoms. Such SLD derivation is the one obtained by performing the same *derive* steps (with exactly the same clauses) using the same computation rule and by ignoring the *pop-derive* steps since goals in SLD resolution do not contain $\uparrow$ atoms. We use *simplify*$(D_S) = D$ to denote that D is the SLD derivation which corresponds to D_S.

4.3 Accuracy results

We would now like to impose a condition on the computation rule which allows ensuring that the contents of the stack are precisely the ancestors of the atom to be

$$
\begin{gathered}
\langle\{\underline{\mathtt{qs([1,1,1],R,[])}}\} \mathbin{\pmb{|}} []\rangle \\
\Big\downarrow_{derive} \\
\langle\{\mathbf{2},\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{qsort([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{derive} \\
\langle\{\mathbf{5},\mathbf{6},\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{external-derive} \\
\langle\{\mathbf{6},\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}\rangle] \\
\Big\downarrow_{derive} \\
\langle\{\mathbf{7},\mathbf{8},\ \uparrow\ ,\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1],1,L,L2)},\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{external-derive} \\
\langle\{\mathbf{8},\ \uparrow\ ,\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1],1,L,L2)},\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{derive-fact} \\
\langle\{\ \uparrow\ ,\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1],1,L,L2)},\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{pop-derive} \\
\langle\{\ \uparrow\ ,\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{part([1,1],1,L1,L2)},\mathtt{qs([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{pop-derive} \\
\langle\{\mathbf{3},\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{qsort([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{derive-fact} \\
\langle\{\mathbf{4},\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{qsort([1,1,1],R,[])}]\rangle \\
\Big\downarrow_{derive} \\
\langle\{\mathtt{part([1],1,L1',L2')},\mathbf{10},\mathbf{11},\ \uparrow\ ,\ \uparrow\ \} \mathbin{\pmb{|}} [\mathtt{qsort([1,1],R,[1])},\mathtt{qsort([1,1,1],R,[])}]\rangle
\end{gathered}
$$

Fig. 4. ASLD Derivation for the example

selected. The following notion of *depth-preserving* computation rule allows precisely this.

Definition 14 (*depth-preserving*)
A computation rule $\mathcal{R}$ is *depth-preserving* if for each non-empty goal $G = \leftarrow A_1, \ldots, A_k$ with $A_1 \neq \uparrow$, $\mathcal{R}(G) = A_R$ and $\uparrow \notin \{A_2, \ldots, A_R\}$.

Intuitively, a depth-preserving computation rule always returns an atom which is strictly to the left of the first (leftmost) $\uparrow$ mark. Note that $\uparrow$ is used to separate groups of atoms which are at different depth in the proof tree. Thus, the notion of depth-preserving computation rules in ASLD resolution is *equivalent* to that of local computation rules in SLD resolution.

Proposition 1 (*ancestor stack*)
Let D_S be an ASLD derivation for the initial query G in program P via a *depth-preserving* computation rule. Let D be an SLD derivation such that *simplify*$(D_S) = D$. If, $curr\_goal(D_S) = A_1, \ldots, A_n, \uparrow, \ldots$ and $curr\_ancestors(D_S) = AS$, we distinguish two cases:

- if $A_1 \neq \uparrow$, then $\mathsf{contents}(AS) = Ancestors(A_i, D)$ for $A_i \neq \uparrow$ for $i = 1, \ldots, n$,
- if $A_1 = \uparrow$, then the atom on the top of AS has no descendents in $curr\_goal(D_S)$ and $\mathsf{contents}(\mathsf{pop}(AS)) = Ancestors(A_i, D)$ for $A_i \neq \uparrow$ for $i = 2, \ldots, n$.

Proof
The proof is by induction on the length k of the ASLD derivation, D_S, of the form $S_0, \ldots, S_k$ where S_i, for $i = 0, \ldots, k$, is the sequence of states corresponding to each derivation step from the initial state $S_0 = \langle G \mid \mathsf{empty} \rangle$. To simplify the proof, we do not make explicit distinction between rules **derive** and **derive-fact**.

base case ($k = 1$). Consider the initial state $S_0 = \langle G \mid \mathsf{empty} \rangle$ where the goal G is of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_n$, $n \geq 1$. Initially, all atoms in G are different from $\uparrow$, i.e., $A_i \neq \uparrow$ for $i = 1, \ldots, n$. Therefore, we can only apply rule **derive** to S_0. Let us assume that $\mathcal{R}$ is a *depth-preserving* computation rule and $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause with $\theta = mgu(A_R, H)$. The test $Admissible(A_R, \mathsf{contents}(\mathsf{empty}), \leq_S)$ holds (otherwise the derivation step is not possible). Then, the state $S_1 = \langle G' \mid AS' \rangle$ is derived from S_0 and C where $G' = \theta(B_1, \ldots, B_m, \uparrow, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_n)$ and $AS' = \mathsf{push}(\mathsf{empty}, A_R)$.
Now, we want to prove that $\mathsf{contents}(\mathsf{push}(\mathsf{empty}, A_R)) = Ancestors(B_i, D)$, $i = 1, \ldots, m$, for the equivalent SLD derivation D. Hence, we perform the corresponding SLD step from $\leftarrow A_1, \ldots, A_R, \ldots, A_m$ using the same computation rule $\mathcal{R}$ and the same clause C. In D, we derive the goal:

$$\theta(B_1, \ldots, B_m, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k)$$

By definition of ancestor (Def. 6), A_R is the only ancestor of B_i in D, $i = 1, \ldots, m$. Consequently, $\mathsf{contents}(\mathsf{push}(\mathsf{empty}, A_R)) = Ancestors(B_i, D)$ holds and our claim follows.

inductive case ($k > 1$). We decompose the ASLD derivation D_S of length k in two parts. The first part, D_{S-1}, is the derivation from S_0 to S_{k-1} of length $k-1$. The second part corresponds to the last ASLD derivation step from S_{k-1} to S_k. Let $S_{k-1} = \langle G_{k-1} \mid AS_{k-1} \rangle$ with $G_{k-1} = A_1, \ldots, A_n, \uparrow, \ldots$ and $A_i \neq \uparrow$ for $i = 1, \ldots, n$. We now distinguish two cases depending on the value of n:

($n > 0$): We first apply the inductive hypothesis to the ASLD derivation, D_{S-1}, of length $k-1$ of the form $S_0, \ldots, S_{k-1}$. Consider that D' is the equivalent SLD derivation obtained by $simplify(D_{S-1}) = D'$. Now, we perform the last ASLD derivation step from S_{k-1}. Since $A_1 \neq \uparrow$, we can only apply rule **derive** to S_{k-1}. By assumption, $\mathcal{R}$ is a *depth-preserving* computation rule. Thus, it will select an atom A_R from A_1 to A_n. In particular, assume that $\mathcal{R}(G_{k-1}) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause with $\theta = mgu(A_R, H)$. We assume that the test $Admissible(A_R, \mathsf{contents}(AS_{k-1}), \leq_S)$ holds, otherwise the step is not possible. Then, $S_k = \langle G_k \mid AS_k \rangle$ is derived from S_{k-1} and C where

$$\begin{array}{rcl} G_k & = & \theta(B_1, \ldots, B_m, \uparrow, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_n, \uparrow, \ldots) \\ AS_k & = & \mathsf{push}(AS_{k-1}, A_R) \end{array}$$

Now, we want to prove that $\mathsf{contents}(AS_k) = Ancestors(B_i, D)$, for $i = 1, \ldots, m$, for the equivalent SLD derivation D. Hence, we perform the corresponding SLD step from the last goal, named Q, in D'. We know that

Q is of the form $Q = A_1, \ldots, A_n, \ldots$ since *simplify*$(D_{S-1}) = D'$ and all $A_i \neq \uparrow$. By using the same local computation rule for SLD resolution, the selected atom is also A_R. With the same clause C, we derive the goal $\theta(B_1, \ldots, B_m, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_n, \ldots)$. Now, by applying Definition 6), the ancestors of B_i are A_R plus the ancestors of A_R in D', for $i = 1, \ldots, m$.

Finally, we proceed to put together the conclusions obtained from the two derivations. On one hand, we have that $\mathsf{contents}(AS_{k-1}) = \mathit{Ancestors}(A_i, D')$, $i = 1, \ldots, n$. In particular, we have that $\mathsf{contents}(AS_{k-1}) = \mathit{Ancestors}(A_R, D')$ for $i = R$. Thus, we have that:

$$\begin{array}{lcl} AS_k & = & \mathsf{push}(AS_{k-1}, A_R) \\ \mathsf{contents}(AS_k) & = & [A_R | AS_k] = \mathit{Ancestors}(B_i, D) \end{array}$$

which proves our claim.

$(n = 0)$: In this case, the goal is of the form $G_{k-1} = \uparrow, C_1, C_2, \ldots$. By the inductive hypothesis, we know that the atom on the top of AS_k has no descendents in *curr_goal*(D_{S-1}) and $\mathsf{contents}(\mathsf{pop}(AS_{k-1})) = \mathit{Ancestors}(C_i, D')$ for $C_i \neq \uparrow$ for $i = 1, \ldots, n$. Now, the only possibility is that $S_k = \langle G_k \,\mathsf{I}\, AS_k \rangle$ is *pop-derived* from S_{k-1} with $G_k = C_1, C_2 \ldots$ and $AS_k = \mathsf{pop}(AS_{k-1})$. Therefore, we have that $\mathsf{contents}(AS_k) = \mathsf{contents}(\mathsf{pop}(AS_k)) = \mathit{Ancestors}(C_i, D')$. Finally, in the equivalent SLD derivation step D from D', no step is performed as *simplify* removes the corresponding atom (i.e., the $\uparrow$ mark). Hence, $\mathit{Ancestors}(C_i, D) = \mathit{Ancestors}(C_i, D')$ and the result holds.

□

The above result trivially holds for leftmost unfolding which is always depth-preserving. The next theorem guarantees that we do not lose any specialization opportunities by using our stack-based implementation for ancestors instead of the more complex tree-based implementation, i.e., our proposed semantics will not stop "too early". It is a consequence of the above proposition and the results in (Bruynooghe et al. 1992) about wqo.

Theorem 1 (*accuracy*)
Let D be an SLD derivation for query G in a program P via a local computation rule. Let $\leq_S$ be a wqo. If the derivation D is safe w.r.t. $\leq_S$ then there exists an ASLD derivation D_S for G and P via a depth-preserving computation rule such that *simplify*$(D_S) = D$.

Proof
The proof is by contradiction. We consider the *safe* SLD derivation D of length k for G via a local computation rule $\mathcal{R}$. Trivially, the partial derivation D' of length $k - 1$ from G to a goal G' is safe.

Now, the assumption is that, D_S, the ASLD derivation for $S = \langle G \,\mathsf{I}\, \mathsf{empty} \rangle$ corresponding to D is *not* safe. In particular, we consider the partial ASLD derivation, D'_S, from the state S to the state S', such that *simplify*$(D'_S) = D'$ and, from which a further ASLD derivation step for S' is not safe, i.e., it would result in an inadmissible derivation. The state S' is of the form $S' = \langle G' \mid AS' \rangle$ with $G' = A_1, \ldots, A_n, \uparrow, \ldots$ and $A_i \neq \uparrow$, for $i = 1, \ldots, n$. By Definition 14, the depth-preserving computation rule can only select an atom A_i, for $i = 1, \ldots, n$.

Since a safe derivation step from S' cannot be performed, the truth value of the expression:

$$Admissible(A_i, contents(AS'), \leq_S)$$

is false for any selected atom A_i, $i = 1, \ldots, n$. By Definition 4, this means that $\forall A_i,\ \exists B \in contents(AS') : B \leq_S A_i$. By applying Proposition 1, we have that the truth value of $Admissible(A_i, Ancestors(A_i, D'), \leq_S)$ is false as well. Therefore, $\forall A_i,\ \exists B \in Ancestors(A_i, D') : B \leq_S A_i$.

Finally, since $simplify(D'_S) = D'$ and all atoms $A_i \neq \uparrow$, G' is a goal of the form $A_1, \ldots, A_n, \ldots$ The equivalent computation rule, $\mathcal{R}$, can select the same atoms A_i. However, $Admissible(A_i, Ancestors(A_i, D'), \leq_S)$ is false for all A_i, for $i = 1, \ldots, n$. Thus, the last derivation step in D is inadmissible, hence, we have a contradiction. □

Note that since our semantics disables performing any further steps as soon as inadmissible sequences are detected, not all local SLD derivations have a corresponding ASLD derivation. However, if a local SLD derivation is safe, then its corresponding ASLD derivation can be found.

It is interesting to note that we can allow more flexible computation rules which are not necessarily depth-preserving while still ensuring termination. For instance, consider a state $\langle A_1, \ldots, A_n, \uparrow, A_R, \ldots \mid P \rangle$ with $\uparrow \notin \{A_1, \ldots, A_n\}$ and a non depth-preserving computation rule which selects the atom A_R to the right of the $\uparrow$ mark. Then, rule *derive* will check admissibility of A_R w.r.t. all atoms in the stack P. However, the topmost atom of P, say P_1, is an ancestor only of the atoms A_i to the left of A_R but it is not an ancestor of A_R. The more $\uparrow$ marks the computation rule jumps over to select an atom, the more atoms which do not belong to the ancestors of the selected atom that will be in the stack, thus, the more accuracy and efficiency we lose. In any case, the stack will always be an over-approximation of the actual set of ancestors of A_R.

Our local unfolding rule based on ancestor stacks can be used within any partial deduction framework, including Conjunctive Partial Deduction (CPD) (De Schreye et al. 1999). In principle, its use within the CPD framework does not pose any particular difficulty and our unfolding rule can simply be incorporated as any other strategy within the method. Indeed, the main distinction of CPD w.r.t. non conjunctive methods is on the use of an enhanced global control which generates a set of conjunctions rather than individual atoms, while any of the existing local control strategies can be used in combination with such a global control. The only requirement is that the unfolding rule takes as input a conjunction of atoms rather than a single atom, which is always a trivial extension. It should be noted that some CPD examples may require the use of an unfolding rule which is not depth-preserving to obtain the optimal specialization. As we discuss above, we cannot ensure accuracy results (though we still have correctness) in these cases but in turn the use of local unfolding will improve the efficiency of the partial deduction process, as our experimental results will show later.

5 Assertion-based Unfolding for External Predicates

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external.* Examples of external predicates are (1) the traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.) and basic input/output facilities; (2) those predicates defined in a different module, (3) predicates written in another language, etc. This section deals with the difficulties that such *external* predicates pose during partial deduction and extends our ASLD semantics to deal with them.

5.1 The notion of evaluable atom

When an atom A, such that $pred(A) = p/n$ is an external predicate, is selected during partial deduction, it is not possible to apply the *derive* rule in Definition 2 due to several reasons. First, we may not have the code defining p/n and, even if we have it, the derivation step may introduce in the residual program calls to predicates which are private to the module M where p/n is defined. In spite of this, if the executable code for the external predicate p/n is available, and under certain conditions, it can be possible to fully evaluate calls to external predicates at specialization time. We use $\mathsf{Exec}(Sys, M, A)$ to denote the execution of atom A on a logic programming system Sys (e.g., `Ciao` or SICStus) in which the module M, where the external predicate p/n is defined, has been loaded. In the case of logic programs, $\mathsf{Exec}(Sys, M, A)$ can return zero, one, or several computed answers for $M \cup A$ and then execution can either terminate or loop. We will use substitution sequences (Le Charlier et al. 2002) to represent the outcome of the execution of external predicates. A *substitution sequence* is either a finite sequence of the form $\langle\theta_1, \ldots, \theta_n\rangle$, $n \geq 0$, or an incomplete sequence of the form $\langle\theta_1, \ldots, \theta_n, \perp\rangle$, $n \geq 0$, or an infinite sequence $\langle\theta_1, \ldots, \theta_i, \ldots\rangle$, $i \in I\!N^*$, where $I\!N^*$ is the set of positive natural numbers and $\perp$ indicates that the execution loops. We say that an execution *universally terminates* if $\mathsf{Exec}(Sys, M, A) = \langle\theta_1, \ldots, \theta_n\rangle$, $n \geq 0$.

In addition to producing substitution sequences, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes the evaluation of such atoms to be performed at partial evaluation time, since those effects need to be performed at run-time. A clear example of this are input/output facilities. In order to capture the requirements which allow executing external predicates at partial deduction time we now introduce the notion of *evaluable* atom:

Definition 15 (*evaluable*)
Let A be an atom such that $pred(A) = p/n$ is an external predicate defined in module M. We say that A is *evaluable* in a logic programming system Sys if $\mathsf{Exec}(Sys, M, A)$ satisfies the following conditions:

1. it universally terminates
2. it does not produce side-effects
3. it does not issue errors
4. it does not generate exceptions

We also say that an expression E is evaluable if 1) E is an evaluable atom, or 2) E is a conjunction of evaluable expressions, or 3) E is a disjunction of evaluable expressions.

Clearly, some of the above properties are not computable (e.g., termination is undecidable in the general case). However, it is often possible to determine some *sufficient conditions* (SC) which are *decidable* and ensure that, if an atom A satisfies such conditions, then A is evaluable. Intuitively, a sufficient condition can be thought of as a traditional precondition which ensures a certain behavior of the execution of a procedure provided they are satisfied. Then, if this process is applied to a call corresponding to an external predicate which is selected during partial deduction, then that call can be executed directly at partial deduction time. To formalize this, we propose to use the notion of *evaluable assertion.* Basically, an evaluable assertion is a pair containing a predicate descriptor and the sufficient conditions for its instances to be evaluable.

Definition 16 (*correct evaluable assertion*)
Let p/n be an external predicate defined in module M. An evaluable assertion $(p(X1, ..., Xn), SC)$ is correct for predicate p/n in a logic programming system Sys if, $\forall\theta$:

- the expression $\theta(SC)$ is evaluable, and
- if $\mathsf{Exec}(Sys, M, \theta(SC)) = \langle \mathsf{id} \rangle$ then $\theta(p(X1, ..., Xn))$ is evaluable.

In principle, assertions have to be provided manually by the supplier of the (external) code. However, for predicates that are defined in the source language and use only external predicates for which those assertions are available, existing analysis tools (like those within the `CiaoPP` system[4]) are able to infer them in many practical cases (see (Albert et al. 2006)), as we will discuss later.

One of the advantages of using this kind of assertion is that it makes it possible to deal with new external predicates (e.g., written in other languages) in user programs or in the system libraries without having to modify the partial evaluator itself. Also, the fact that the assertions are co-located with the actual code defining the external predicate, i.e., in the module M (as opposed to being in a large table inside the partial deduction system) makes it more difficult for the assertion to be left out of sync when a modification is made to the external predicate. We believe this to be very important to the maintainability of a real application or system library.

[4] In this system, evaluable assertions are called `eval` assertions.

Example 2
Let us consider the following assertion for the builtin predicate $\leq$:

$$(\texttt{A} =< \texttt{B}, (\texttt{arithexpr}(\texttt{A}), \texttt{arithexpr}(\texttt{B})))$$

which states that if predicate `=</2` is called with both arguments instantiated to a term of type `arithexpr`, then the call is evaluable in the sense of Definition 15. In our implementation, we use the "*computational* assertions" which are part of the assertion language (Puebla et al. 2000) of `CiaoPP`, the Ciao system preprocessor (Hermenegildo et al. 2005), in order to declare evaluable assertions.

The type `arithexpr` corresponds to arithmetic expressions which, as expected, are built out of numbers and the usual arithmetic operators. In our implementation in Ciao, the type `arithexpr` is expressed as a unary regular logic program. This allows using the underlying Ciao system in order to effectively decide whether a term is an `arithexpr` or not.

5.2 The extension of ASLD resolution

The following definition extends our ASLD semantics by providing a new rule, **external-derive**, for evaluating calls to external predicates. Given a sequence of substitutions $\langle\theta_1, \ldots, \theta_n\rangle$, we define $Subst(\langle\theta_1, \ldots, \theta_n\rangle) = \{\theta_1, \ldots, \theta_n\}$.

Definition 17 (*external-derive*)
Let Sys be a logic programming system. Let $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ be a goal. Let $S = \langle G \mid AS\rangle$ be a state and AS a stack. Let $\mathcal{R}$ be a computation rule such that $\mathcal{R}(G) = A_R$ with $pred(A_R) = p/n$ an external predicate from module M. Let C be an evaluable assertion $(p(X1, ..., Xn), SC)$. Then, $S' = \langle G' \mid AS'\rangle$ is *external-derived* from S and C via $\mathcal{R}$ in Sys if:

$$\begin{aligned}
\sigma &= mgu(A_R, p(X1, ..., Xn)) \\
\mathsf{Exec}(Sys, M, \sigma(SC)) &= \langle \mathsf{id} \rangle \\
\theta &\in Subst(\mathsf{Exec}(Sys, M, A_R)) \\
G' \text{ is the goal } \theta(A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \\
AS' &= AS
\end{aligned}$$

Notice that, since after computing $\mathsf{Exec}(Sys, M, A_R)$ the computation of A_R is finished, there is no need to push (a copy of) A_R into AS and the ancestor stack is not modified by the **external-derive** rule. This rule can be nondeterministic if the substitution sequence for the selected atom A_R contains more than one element, i.e., the execution of external predicates is not restricted to atoms which are deterministic. The fact that A_R is evaluable implies universal termination. This in turn guarantees that in any ASLD tree, given a node S in which an external atom has been selected for further resolution, only a finite number of descendants exist for S and they can be obtained in finite time.

Example 3

```
:- module(main_prog,[main/2],[]).
:- use_module(comp,[long_comp/2],[]).

main(X,Y)    :- problem(X,Y), q(X).

problem(a,Y):- ground(Y),long_comp(c,Y).
problem(b,Y):- ground(Y),long_comp(d,Y).

q(a).
```

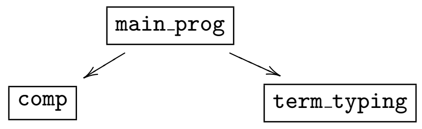

Fig. 5. Motivating Example

Consider the `Ciao` system with the assertion in Example 2 for `1=<1`. Consider also the atoms **5** and **7**, which are of the form `1=<1`, in the ASLD derivation of Figure 2. Both atoms can be evaluated because

$$\mathsf{Exec}(\mathit{ciao}, \mathit{arithmetic}, (\mathit{arithexpr}(1), \mathit{arithexpr}(1))) = \langle \mathsf{id} \rangle$$

This is a sufficient condition for $\mathsf{Exec}(\mathit{ciao}, \mathit{arithmetic}, (1 =< 1))$ to be evaluable. Its execution returns $\mathsf{Exec}(\mathit{ciao}, \mathit{arithmetic}, (1 =< 1)) = \langle \mathsf{id} \rangle$.

In addition to the conditions discussed above which allow evaluating atoms for external predicates at specialization time, an orthogonal issue is that of the correctness of non-leftmost unfolding in the presence of external predicates. For logic programs without impure predicates, non-leftmost unfolding is sound thanks to the independence of the computation rule (see for example (Lloyd 1987)).[5] Unfortunately, non-leftmost unfolding poses several problems in the context of *full* Prolog programs with *impure* predicates, where such independence does not hold anymore. For instance, `ground/1` is an *impure* predicate since, under LD resolution, the goal `ground(X),X=a` fails whereas `X=a,ground(X)` succeeds with computed answer X/a. Those executions are not equivalent and, thus, the independence of the computation rule does no longer hold. As a result, given the goal $\leftarrow$ `ground(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable `X` in the call to `ground(X)`, the goal will succeed at specialization time, whereas the initial goal fails in LD resolution at run-time. The above problem was early detected (Sahlin 1993) and it is known as the problem of *backpropagation of bindings.* Also *backpropagation of failure* is problematic in the presence of impure predicates. For instance, $\leftarrow$ `write(hello),fail` behaves differently from $\leftarrow$ `fail,write(hello)`.

In order to illustrate the problem, consider the `Ciao` program in Fig. 5, which uses the impure predicate `ground/1` and whose modular structure appears to the right. `term_typing` is the name of the module in `Ciao` where `ground/1` is defined and predicate `long_comp/2` is imported from the user module `comp`. Consider a deterministic unfolding rule and the entry declaration: "`:- entry main(X,a).`". The

[5] However, non-deterministic unfolding of non-leftmost atoms can degrade efficiency.

unfolding rule performs an initial step and derives the goal `problem(X,a),q(X)`. Then, it cannot select the leftmost atom `problem(X,a)` because its execution performs a non deterministic step. In this situation, different decisions can be taken. a) We can stop unfolding at this point. However, in general, it may be profitable to unfold atoms other than the leftmost. Interesting computation rules are able to detect the above circumstances and "jump over" the problematic atom in order to proceed with the specialization of another atom (in this case `q(X)`). We can then decide to b) unfold `q(X)` but avoiding backpropagating bindings or failure onto `problem(X,a)`. And the final possibility c) is to unfold `q(X)` while allowing backpropagation onto `problem(X,a)`. However, this will require that some additional requirements hold on the atom(s) to the left of the selected one.

There are several solutions in the literature (see, e.g.,(Leuschel 1994; Etalle et al. 1997; Albert et al. 2002; Leuschel and Bruynooghe 2002; Leuschel et al. 2004)) which allow unfolding non-leftmost atoms by avoiding the backpropagation of bindings and failure, i.e., in the spirit of possibility b). Basically, the common idea is to represent explicitly the bindings by using unification (Leuschel 1994) or residual case expressions (Albert et al. 2002) rather than backpropagating them (and thus applying them onto leftmost atoms). For our example, by using unification, we can unfold `q(X)` and obtain the resultant `main(X,a):-problem(X,a),X=a`. This guarantees that the resulting program is correct, but it definitely introduces some inaccuracy, since bindings (and failure) generated during unfolding of non-leftmost atoms are hidden from atoms to the left of the selected one. The relevant point to note is that preventing backpropagation, by using one of the existing methods, can be a bad idea for at least the following reasons:

1. *Backpropagation of bindings and failure can lead to an early detection of failure*, which may result in important speedups. For instance, if we allow backpropagating the binding `X=a` to the left atom, we get rid of the whole (failing) computation for `problem(b,a)` in the residual program.
2. *Backpropagation of bindings can make the profitability criterion for the leftmost atom to hold*, which may result in more aggressive unfolding. In the example, by backpropagating, we obtain the atom `problem(a,a)` which allows a deterministic computation rule to proceed to its unfolding.
3. *Backpropagation of bindings may allow better indexing* by further instantiating arguments in clause heads. This is often good from a performance point of view (see, e.g., (Venken and Demoen 1988)). In our example, we will obtain the clause head `main(a,a)` with better indexing than `main(X,a)`.

The bottom line is that backpropagation should be avoided only when it is really necessary since interesting specializations can no longer be achieved when it is disabled.

The problems involved in and some possible solutions to non-leftmost unfolding can be found in the literature (Leuschel 1994; Etalle et al. 1997; Albert et al. 2002; Leuschel and Bruynooghe 2002). However, there is still ample room for improvements. In particular, the intensive use of static analysis techniques in this assertion-based context seems particularly promising. We are investigating the use

```
:- module(_,[p/2]).
:- use_package(library(assertions)).

p(Y,L):- findall(X,property(X),L), \+ r(Y).

:- trust pred property/1 + (eval,sideff(free)).
property(X):- q(X), \+ r(X).

q(a).  q(b).

:- trust pred r/1 + (eval,sideff(free)).
r(b).
```

Fig. 6. Program with meta-calls

of the analyzers available in `CiaoPP` with this aim, though this is outside the scope of this article.

5.3 Handling of meta-predicates

Though not introduced in the formalization for simplicity, our partial evaluator can handle the usual Prolog *meta-predicates*, such as `call/1`, `findall/3`, `bagof/3`, and `setof/3`. Meta-predicates are characterized by receiving one or more atoms as input. For example, `call/1` receives an atom as its only input and `findall/3` receives a goal in its second argument position. The simplest possible handling of meta-predicates consists in residualizing all meta-calls, i.e., all calls to meta-predicates, and transferring the atoms which appear as arguments in such meta-calls to the global control for their subsequent partial evaluation. For this, all meta-predicates must be declared as such and the arguments which contain atoms must be known in advance. In the case of `Ciao` this is done using assertions.

As a further optimization, when the atoms which appear in meta-calls are evaluable, then rather than residualizing the meta-call, our partial evaluator evaluates both the atom itself and also the call to the meta predicate. This is an important optimization because partial evaluation loses a lot of precision when unfolding is stopped and atoms are transferred to the global control.

Another important feature of Prolog programs is negation as failure, i.e., the `\+/1` meta predicate. In order to preserve the semantics of negation as failure, evaluation of a meta-call of the form `\+ A` requires `A` to be ground. Therefore, at partial evaluation time a meta-call `\+ A` is only evaluated if both `A` is evaluable and ground. If this is not the case, the meta-call is residualized and `A` is transfered to the global control. This allows a relatively simple handling of negation as failure where `\+/1` is considered as a meta predicate with the additional evaluation requirement that its associated atom is ground.

Example 4
Figure 6 shows an example `Ciao` program containing calls to the `findall/3` meta-

```
:- module(_, [p/2] ).
:- use_package(library(assertions)).

p(A,[a]) :- \+r_1(A) .

:- trust pred r_1(_1) + (eval, sideff(free)).
r_1(b).
```

Fig. 7. Partially evaluated program with meta-calls

predicate and negation as failure. The `trust` assertions in `Ciao` syntax inform the partial evaluator that all calls to the `property/1` and `r/1` predicates are evaluable.

As a result, the `findall(X,property(X),L)` meta-call is evaluable and can be replaced by the unification `L=[a]`. However, the second meta-call, i.e., `\+ r(Y)` is residualized since `Y` remains a variable at partial evaluation time. The resulting program obtained by our partial evaluator is shown in Figure 7. Since partially evaluated atoms are renamed, the specialized version of `r(Y)` has been renamed to `r_1(Y)`. The atom `p(A,B)` keeps its original name since it is an exported predicate, in order to preserve the module interface.

6 Experimental Results

We have implemented in our partial evaluation system the unfolding rule we propose, together with other variations in order to evaluate the efficiency of our proposal. Our partial evaluation system has been integrated in a practical state of the art compiler which uses global analysis extensively: the `Ciao` compiler and, specifically, its preprocessor `CiaoPP` (Hermenegildo et al. 2005). For the tests, the whole system has been compiled using Ciao 1.13 (Bueno et al. 2009). All of our experiments have been performed on an Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.28-15.

The programs used as benchmarks are indicated in the **Bench** column. They are classical programs often used as benchmarks for analysis and partial evaluation of logic programs. They are described in more detail below. Since our proposal improves the performance of the unfolding process, i.e., the local control, we have chosen as benchmarks programs whose partial evaluation performs plenty of unfolding, since this allows observing the benefits of our proposal better. In particular, three of the benchmarks considered: `advisor3`, `query`, and `zebra` can be fully unfolded using homeomorphic embedding with ancestors. In the rest of the programs we provide initial queries which are partially instantiated in order to show that our partial evaluation system also includes global control and can partially evaluate programs whose input data is not fully instantiated. Our global control is also based on homeomorphic embedding. When a new atom is going to be specialized, we first check whether it embeds any of the previously specialized atoms. In that case, the new atom is generalized before being specialized by using the most specific generalization of the new and the embedded atom. Otherwise, the new atom is specialized as is. For our experiments, we use as input lists whose first part is instantiated to

	Relation		**Tree**		**Stack**		**Rel/Stack**		**Tree/Stack**	
Bench	**G**	**L**	**G**	**L**	**G**	**L**	**T**	**L**	**T**	**L**
advisor3	0	103	0	183	0	151	0.68	0.68	1.21	1.21
nrev_80	∞	∞	38	50622	46	7985	∞	∞	6.31	6.34
nrev_43	12	912	10	2804	13	774	1.17	1.18	3.58	3.62
permute6	4	526	4	651	9	453	1.15	1.16	1.42	1.44
query	0	92	0	102	0	86	1.07	1.07	1.19	1.19
qsort_80	∞	∞	15571	430485	15582	47923	∞	∞	7.02	8.98
qsort_23	222	797	229	1615	213	566	1.31	1.41	2.37	2.85
rev_80	3	555	2	581	2	547	1.02	1.01	1.06	1.06
zebra	0	1043	0	1682	0	1052	0.99	0.99	1.60	1.60

Table 1. *Performance of Ancestor Stacks in Terms of Execution Time*

integers and then the rest of the list is unknown, i.e., just a variable, at partial evaluation time. In the tables, we add to the name of the benchmark the number of elements in the input list which are instantiated. For example, `nrev_80` should be interpreted as the well-known naive reverse program together with a query which has as input a list of the form $[1, \ldots, 80|T]$, with T a free variable.

The `advisor3` program is a variation of the advisor program in the DPPD (Leuschel 2002a) library. The `query` and `zebra` programs are classical benchmarks for program analysis. In particular, `query` performs a query to a small Prolog database and `zebra` implements a simple logical puzzle. Program `qsort` corresponds to the quick-sort program shown in the article. The part of the list which is instantiated is not ordered. The `rev` benchmark is another list reversal program, but now with linear complexity, using an accumulator. Finally, `permute` is a permutation program which uses a nondeterministic deletion predicate. Note that two of the programs (`nrev` and `qsort`) are partially evaluated w.r.t. two different input lists. The smaller of the two corresponds to the largest possible partially instantiated list that the partial evaluator can handle using the Relation implementation explained below, without running out of memory. Importantly, none of `advisor3`, `query`, nor `zebra` can be fully unfolded using homeomorphic embedding over the full sequence of selected atoms. Also, `nrev` and, as seen in the running example, `qsort` are potentially not fully unfolded if the input lists contain repetitions unless ancestors are considered.

In the next two tables, we compare three different implementations of unfolding based on homeomorphic embedding with ancestors:

Relation We refer to an implementation where each atom in the resolvent is annotated with the list of atoms which are in its ancestor relation, as done in the example in Figure 2.

Trees This column refers to the implementation where the ancestor relations of the different atoms are organized in a proof tree.

Stacks The column **Stacks** refers to our proposed implementation based on ancestor stacks.

6.1 Execution times

Let us explain the results in Table 1. Times are in milliseconds, measuring *runtime*, and are computed as the arithmetic mean of five runs. The partial evaluation time in each implementation is split into two columns. The first one, labeled **G**, shows the time taken by global control. The second one, labeled **L**, shows the time taken by local control (i.e., unfolding). The benchmarks **nrev_80** and **qsort_80** contain the value ∞ instead of a number in the **G** and **L** columns for **Relation** to indicate that the partial evaluation system has run out of memory. For each of these two benchmarks, we have repeated the experiment with the largest possible initial query that **Relation** can handle in our system before running out of memory, i.e., **nrev_43** and **qsort_23**. **Relation** is quite efficient in time for those benchmarks it can handle, though a bit slower than the one based on stacks. However, and as can be seen in Table 2, its memory consumption is extremely high, which makes this implementation inadmissible in practice. Regarding **Trees**, this implementation, based on proof trees, has good memory consumption but it is significantly slower than **Relation** due to the overhead of traversing the tree for retrieving the ancestors of each atom.

The last four columns compare the relative specialization times of **Relation** and **Trees** w.r.t. the **Stacks** algorithm. It should be observed that these three alternatives are different implementations of the same local control strategy, and that the same global control strategy is used in all three cases. Therefore, exactly the same residual programs are obtained in the three cases. As the table shows (with values greater than one), **Stacks** is faster than **Trees** in all cases. Furthermore, **Stacks** is even faster than the implementation based on explicitly storing all ancestors of all atoms (**Relation**) for most programs, while having a memory consumption comparable to (and in fact, slightly better than) the implementation based on proof trees. Two speedups are shown per implementation. One, named **L**, only considers the time required for local control, and the other one, named **T**, considers the total time of global plus local control. The actual speedups w.r.t. **Trees** range from 1.06 in the case of `rev_80` to 8.98 **L** (7.02 **T**) in the case of `qsort_80`. This variation is due to the different shapes which the proof trees can have for the (derivations in the) SLD tree. In the case of `rev`, the speedup is low since the SLD tree consists of a single derivation whose proof tree has a single branch. Thus, in this case considering the ancestor sequence is indeed equivalent to considering the whole sequence of

	Memory Consumption			**Relative Memory Reduction**	
Bench	**Relation**	**Trees**	**Stacks**	**Relation**	**Trees**
advisor3	1667260	850612	751112	2.22	1.13
nrev_80	mem	1076384	944936	∞	1.14
nrev_43	56255068	1103980	1041490	54.01	1.06
permute_6	23361920	1959004	1431976	16.31	1.37
query	2764368	8064	7520	367.60	1.07
qsort_80	mem	5660460	5038540	∞	1.12
qsort_23	11130584	630048	598212	18.61	1.05
rev_80	2552524	144264	139076	18.35	1.04
zebra	26819712	107760	101280	264.81	1.06
Overall				∞	1.15

Table 2. *Performance of Ancestor Stacks in terms of Memory Consumption*

selected atoms. But note that this only happens for binary clauses. It is also worth noticing that the speedup achieved by the `Stacks` implementation increases with the size of the SLD tree, as can be seen in the two benchmarks which have been specialized w.r.t. different queries. The overall resulting speedup of our proposed unfolding rule over other existing ones is significant: over 8 times faster than our tree-based implementation.

6.2 Memory consumption

We have also studied the memory required by the unfolding process. Let us briefly discuss the figures depicted in Table 2 which represent, in number of bytes, memory consumption. It has been measured at each derivation step during the construction of the ASLD trees. At each step, the resulting numbers for all memory areas (stack, heap, etc.) have been added and then compared to the previous maximum value, taking always the larger of the two, thus computing the high water mark, i.e., the maximum memory required to perform unfolding. The figures show, for each benchmark, the high water mark minus the memory already in use when the construction of the SLD tree was started. In order to make these numbers closer to the actual memory used, garbage collection has remained enabled during the different experiments. In order to make memory figures comparable, we force garbage collection just before starting partial evaluation of each benchmark.

In the last row, labeled **Overall**, we summarize the results for the different benchmarks using a weighted mean, which places more importance on those benchmarks with relatively larger unfolding figures. We use as weight for each program its actual unfolding time/memory. We believe that this weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large unfolding tree is computed is more relevant than achieving this for small trees.

As Table 2 shows, the **Stacks** algorithm presents lower consumption than either of the two other algorithms studied for any of the programs. It can be seen that the amount of memory required by the **Relation** algorithm precludes it from its practical usage. Regarding the **Stacks** algorithm, not only it is significantly faster than the implementation based on trees. Also it provides a relatively important reduction (1.15 overall, computed again using a weighted mean) in memory consumption over **Trees**, which already has a good memory usage.

Altogether, when the results of Table 1 and Table 2 are combined, they provide evidence that our proposed techniques allow significant speedups while at the same time requiring somewhat less memory than tree based implementations and much better memory consumptions than implementations where the ancestor relation is directly computed. This suggests that our techniques are indeed effective and can contribute to making partial evaluation a practical tool.

6.3 Comparison with Ecce. Specialization Quality.

Finally, in Table 3, we want to compare our implementation with that of a state-of-the-art partial evaluator and see the quality of the specialized programs. To do so, we have also measured the time that it takes to process the same benchmarks using Leuschel's Ecce (Leuschel 2002a) system. For this, we have used the compiled version available at http://www.stups.uni-duesseldorf.de/˜asap/asap-online-demo/meccedownloads and run the experiments on the same machine. These execution times are provided in columns **Ecce**$_L$ and **Ecce**$_G$ which show, respectively, the time taken by local and the global control in Ecce. When compared with **L** and **G** in Table 1 for the stack implementation, the results provide evidence that our proposed stack-based implementation compares quite well with state of the art systems as regards specialization times. Indeed, the specialization times using our stack-based implementation are considerably smaller for all benchmarks with high local control times. In those benchmarks in which Ecce is faster than the **Stacks** implementation, it is due to the unfolding rules not being identical which results in Ecce performing fewer unfolding steps. Note that performing less unfolding may lead to less specialized programs, which are often less efficient.

The next columns aim at evaluating the quality of the specialized programs in Ecce and in our system by comparing their runtimes with those of the original programs. We have chosen sufficiently large input data and run the original program (column **Orig**), the specialized one by our system (column **Stacks**) and the specialized one by Ecce (column **Ecce**) on the same data and the same number of

Bench	**Ecce$_L$**	**Ecce$_G$**	**Orig**	**Stacks**	**Ecce**	**O/S**	**O/E**	**E/S**
advisor3	0	30	1149	1119	1042	1.03	1.10	0.93
nrev_80	19310	1297700	1005	30	64	33.50	15.70	2.16
nrev_43	2910	105600	864	41	102	21.07	8.50	2.49
permute_6	40	20	934	301	620	3.10	1.51	2.06
query	20	90	1106	55	570	20.11	1.94	10.32
qsort_80	85300	269070	1178	15	17	78.53	70.14	1.11
qsort_23	260	900	978	34	34	29.12	29.12	1.00
rev_80	10	730	1132	712	704	1.59	1.61	0.99
zebra	170	300	6	1069	384.90	2.30	167.00	

Table 3. *Comparison with Ecce. Specialization Quality.*

times and show the aggregated runtime. The last three columns show the speedup achieved for each benchmark. In particular, **O/S** and **O/E** show, respectively, the speedup of Stacks and Ecce w.r.t. the original program and **E/S** compares Ecce against Stacks. It should be observed that in all cases the specialized programs in both Ecce and Stacks are more efficient than the original ones and in most cases the gain is significant. The cases in which Stacks performs better than Ecce (e.g., `query` and `zebra`) are because we can fully unfold them in Stacks while Ecce stops the specialization earlier. Hence, the gain is much larger. It is also important to notice that in the example `advisor`, the specialization obtained by Stacks also performs more unfolding steps than the one in Ecce. In this case, such additional unfolding results in an unneeded over-specialization which increases the size of the residual program and leads to a less efficient execution.

7 Related Work and Conclusions

The development of powerful unfolding rules has received considerable attention during the last years (Leuschel and Bruynooghe 2002). The most successful techniques to date are based on two fundamental ingredients:

- the use of a wqo which can be used to guarantee termination while achieving very powerful unfoldings,
- structuring the atoms already visited in each derivation in a tree rather than using an unstructured collection, such as a set.

Among the well-quasi orderings, the *homeomorphic embedding* (Kruskal 1960; Leuschel and Bruynooghe 2002) has proved to be very powerful in practice. Regarding the structure to use for visited atoms, the notion of *ancestors* seems to be the best one since it guarantees termination while allowing transformations which are strictly more powerful than those achievable if unstructured collections are used.

The use of ancestors for refining sequences of visited atoms was proposed early on by (Bruynooghe et al. 1992) and significant effort has been devoted to improve the implementation of ancestors (Martens and De Schreye 1996). However, the combination of wqo and ancestors happens to be very inefficient in practice. This is mainly due to the fact that dependency information has to be maintained for the individual atoms in each derivation. In principle, the use of ancestors should not only allow more powerful transformation but also speed up unfolding since it reduces the length of sequences for which admissibility has to be checked. Unfortunately, maintaining such information about ancestors during the generation of SLD trees introduces a costly overhead which can eliminate the theoretical efficiency gains.

In this work we have proposed ASLD resolution, a novel extension over the SLD semantics to incorporate ancestor stacks which can be used as a basis for the *efficient* generation of (incomplete) SLD trees during partial deduction in combination with wqo. The main features of the implementation technique and extensions that we propose for the ancestor-based local unfolding rule, based on ASLD resolution, are: (1) it is parametric w.r.t. the wqo of interest; (2) it can handle logic programs with builtins; (3) it is guaranteed to always provide finite trees; (4) it is very easy to implement since the ancestor information is simply stored using a stack; (5) it provides a very efficient implementation of ancestor information; (6) if certain conditions are imposed on the computation rule, then it is as accurate as standard (more inefficient) unfolding rules based on ancestors. Note that, as it is the case with unfolding rules based on traditional SLD resolution, our semantics can be used in combination with a determinacy check which may decide to stop unfolding even if termination is guaranteed whenever too many alternative, non-deterministic, branches are generated in the SLD tree.

The unfolding rule proposed in this work has been implemented in the `CiaoPP` system (Hermenegildo et al. 2005), the preprocessor of the `Ciao` programming language. Experimental results are promising: they provide evidence that our proposed techniques allow significant speedups while at the same time requiring somewhat less memory than tree-based implementations and much better memory consumptions than implementations where the ancestor relation is directly computed. Though specialization time is obviously not as critical as execution time, being able to perform powerful specializations in reasonable time can only contribute to the practical takeup of partial deduction techniques.

As for future work, we plan to incorporate in our partial evaluator (embedded in `CiaoPP`) the extensions needed to perform Conjunctive Partial Deduction and to investigate whether local unfolding can be successfully used in this context. We are also investigating additional solutions for the problems involved in non-leftmost unfolding for programs with extra logical predicates beyond those presented in the literature (Leuschel 1994; Etalle et al. 1997; Albert et al. 2002; Leuschel and

Bruynooghe 2002). In particular, the intensive use of static analysis techniques in this context seems particularly promising. In our case we can take advantage of the fact that our partial deduction system is integrated in `CiaoPP`, which includes extensive program analysis facilities. A first step in this direction has been taken in (Albert et al. 2006) by using backwards analysis to infer purity assertions which determine when a non-leftmost step is safe in the presence of impure predicates.

Acknowledgments

We gratefully acknowledge the anonymous referees for many useful comments and suggestions that helped to improve this article. This work was funded in part by the Information Society Technologies program of the European Commission, under the Future and Emerging Technologies IST-231620 *HATS* project and the IST-215483 *SCUBE* project, by the Spanish Ministry of Science and Innovation under the TIN-2008-05624 *DOVES* and HI2008-0153 projects, and by the Madrid Regional Government (CM) under the S-0505/TIC/0407 *PROMESAS* project.

References

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers – Principles, Techniques and Tools.* Addison-Wesley.

Albert, E., Hanus, M., and Vidal, G. 2002. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming 2002,* 1.

Albert, E., Puebla, G., and Gallagher, J. 2006. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05).* Number 3901 in LNCS. Springer-Verlag, 115–132.

Bruynooghe, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming 10*, 91–124.

Bruynooghe, M., Schreye, D. D., and Martens, B. 1992. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing 1,* 11, 47–79.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and (Eds.), G. P. 2004. The Ciao System. Reference Manual (v1.10). Tech. rep., School of Computer Science (UPM). Available at `http://www.ciaohome.org`.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and (Eds.), G. P. 2009. The Ciao System. Ref. Manual (v1.13). Tech. rep. Available at `http://www.ciaohome.org`.

Christensen, N. H. and Glück, R. 2004. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Trans. Program. Lang. Syst. 26,* 1, 191–220.

Craig, S.-J., Gallagher, J. P., Leuschel, M., and Henriksen, K. S. 2004. Fully automatic binding-time analysis for prolog. In *LOPSTR*. 53–68.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. 1999. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming 41,* 2 & 3 (November), 231–277.

Etalle, S., Gabbrielli, M., and Marchiori, E. 1997. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97.* ACM Press, New York, 137–150.

Gallagher, J. 1993. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 88–98.

Hermenegildo, M., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York.

Kruskal, J. 1960. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society 95*, 210–225.

Le Charlier, B., Rossi, S., and Van Hentenryck, P. 2002. Sequence Based Abstract Interpretation of Prolog. *Theory and Practice of Logic Programming 2,* 1, 25–84.

Leuschel, M. 1994. Partial evaluation of the "real thing". In *Proc. of LOPSTR'94 and META'94.* LNCS 883. Springer-Verlag, 122–137.

Leuschel, M. 1996-2002a. The ecce partial deduction system and the dppd library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`.

Leuschel, M. 1998. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of SAS'98*, G. Levi, Ed. LNCS, vol. 1503. Springer-Verlag, Pisa, Italy, 230–245.

Leuschel, M. 2002b. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation - Essays dedicated to Neil Jones*, T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, Eds. LNCS 2566. Springer-Verlag, 379–403.

Leuschel, M. and Bruynooghe, M. 2002. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming 2,* 4 & 5 (July & September), 461–515.

Leuschel, M., Craig, S., Bruynooghe, M., and Vanhoof, W. 2004. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic.* Lecture Notes in Computer Science, vol. 3049. Springer, 340–375.

Leuschel, M., Jørgensen, J., Vanhoof, W., and Bruynooghe, M. 2004. Offline specialisation in prolog using a hand-written compiler generator. *TPLP 4,* 1–2, 139 – 191.

Lloyd, J. 1987. *Foundations of Logic Programming.* Springer, second, extended edition.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming 11*, 217–242.

Martens, B. and De Schreye, D. 1996. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming 28,* 2 (August), 89–146.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming.* Springer LNCS 1870, 23–61.

Sahlin, D. 1993. Mixtus: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing 12,* 1, 7–51.

Sørensen, M. and Glück, R. 1995. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95.* The MIT Press, 465–479.

Venken, R. and Demoen, B. 1988. A partial evaluation system for prolog: some practical considerations. *New Generation Computing 6*, 279–290.