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] *Departments of Computer Science and Electrical and Computer Engineering*
University of New Mexico. E-mail: `herme@unm.edu`

submitted 25 October 2005; revised XXXX; accepted XXXX

Abstract

In spite of the important research efforts in the area, the integration of powerful partial evaluation methods into practical compilers for logic programs is still far from reality. This is related both to 1) efficiency issues and to 2) the complications of dealing with practical programs. Regarding efficiency, the most successful unfolding rules used nowadays are based on *well founded orders* (wfo) or *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 improve the specialization power of unfolding while still guaranteeing termination and also to reduce the number of atoms for which the wfo or 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 any wfo or 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., any atom of the goal can be selected provided it is one of those that have been more recently introduced in the goal. Regarding dealing 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 in a practical partial evaluator, embedded in a state of the art compiler which uses global analysis extensively: the `Ciao` compiler and, specifically, its preprocessor `CiaoPP`. The performance analysis of the resulting system 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.

∗ A preliminary version of this work appeared in the Proceedings of LOPSTR 2004.

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 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 right-hand sides 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.

In spite of the important research efforts in the area, the integration of partial deduction methods into compilers seems to be still far from reality. We believe that the general uptake of partial deduction is being hindered by two factors: 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 *structural* orders 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. A main objective of this work is to contribute to the uptake of partial evaluation techniques by proposing novel solutions to some of these issues.

State-of-the-art partial evaluators integrate terminating unfolding rules for local control based on *structural* orders, 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 admissibility of 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 a novel unfolding rule which relies on the notion of covering ancestors and which allows a very efficient implementation technique based on stacks. 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 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 are promising and provide evidence that our technique pays off in practice and can thus contribute to the practicality of state-of-the-art partial evaluation techniques.

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.

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:

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

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.

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 $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. 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 (goal), 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]

[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).

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 computer 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 goal (query) 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 goals is defined as the union of the partial evaluations for each goal in the set. We refer to (Leuschel and Bruynooghe 2002) for details.

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-founded orderings (wfo) (Bruynooghe et al. 1992; Martens and De Schreye 1996) and well-quasi orderings (wqo) (Sørensen and Glück 1995; Leuschel 1998) are broadly used in the context of on-line partial evaluation techniques (see, e.g., (Gallagher 1993; Leuschel et al. 1998; Sørensen and Glück 1995)).

It is well known that the use of wfos and 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 $s_i \leq_S s_j$. If the order is a wqo, 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 strictly smaller than any previous (comparable) selected atom. Formally, 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 $\forall A_i,\ i \in \{1, \ldots, n\}\ : A \leq_S A_i$. In wfo, it is sufficient to verify that the selected atom is strictly smaller than the previous comparable one (if one exists). Let $<$ be a wfo, by $Admissible(A, (A_1, \ldots, A_n), <)$, with $n \geq 0$, we denote the truth value of the expression $A < A_n$ if $n \geq 1$ and *true* if $n = 0$.

We will denote by *structural order* a wfo or a wqo (written as $\lhd$ to represent any of them). Among the structural orders, well-quasi orderings have proved to be very powerful in practice. In particular, the *homeomorphic embedding* (Kruskal 1960) ordering is the wqo we will use in our examples. The interested reader is referred to Leuschel's work (Leuschel 1998) where a detailed description of homeomorphic embedding can be found. Informally, atom t_1 *embeds* atom t_2 if t_2 can be obtained from t_1 by deleting some operators, e.g., $\mathtt{s}(\underline{\mathtt{s}}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}+\mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$.

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

```
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

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).

Definition 4 (*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 resolvent 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.

Usually, 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). Given an atom A and a derivation D, we denote by $Ancestors(A, D)$ the sequence of ancestors of A in D as defined in Definition 4. 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 $\lhd$ over the covering ancestor subsequences in order to detect inadmissible derivations. An SLD derivation is *safe* with respect to an order (wfo or wqo) if all covering ancestor sequences of the selected atoms are admissible with respect to that order.

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 query 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 $\leftarrow qsort([1,1,1], L, [])$, the program should compute `L=[1,1,1]`, constructing a finite SLD tree. Notice that, in general, if the input arguments to a program are

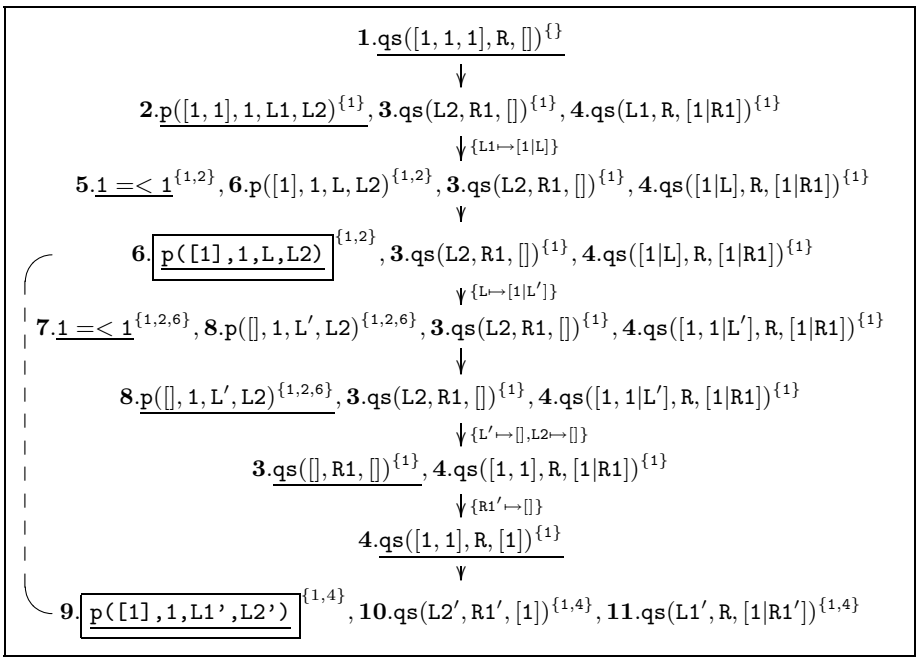

Fig. 2. Derivation with Ancestor Annotations

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 $\leftarrow qsort([1,1,1], R, [])$ 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 structural order. 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., $\mathtt{p}([1], 1, \mathtt{L}', \mathtt{L2}')$, is found for the second time. The reason is that this atom is not strictly smaller than 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 left-to-right 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.

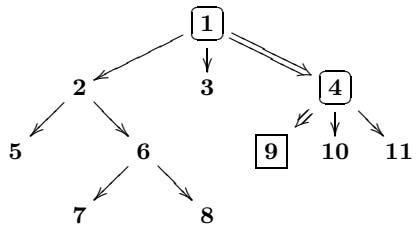

Fig. 3. Proof tree for the example.

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 strictly smaller than 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. 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. ASLD resolution in principle is not tied to local computation rules. 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 5 (*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 $depth(A_i, D)$ 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 6 (*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 $depth(\mathcal{R}(G_i), D) \geq depth(A_{ij}, D) \quad \forall j = 1, \ldots, m_i$

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 left-to-right 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.

An instrumental observation in our approach is that if the proof tree which is used in order to capture the ancestor relation is traversed depth-first, left-to-right, it can be interpreted as an *activation tree* (Aho et al. 1986). In fact, the ancestor subsequence in any point in time corresponds to the current *control word* (Rozenberg and Salomaa 1997) by simply regarding selected atoms as procedure calls. The control word for each execution state can be seen as the set of procedures whose execution has started and is not yet completed, bearing a strong relation with the stack of activation records which most compilers use as a run-time data structure. This data structure takes normally the form of a stack, and this suggests one of the central ideas of our approach: the use of stacks for storing ancestors. Another important observation is that the control word idea 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 have the same ancestor depth, they can be selected in any order and the notion of control word still applies. The advantages of computing the control word instead of the proof tree are clear: the control word corresponds to a single branch in the proof tree from the current selected atom to all its ancestors in the proof tree. Thus, the control word offers advantages both from memory and time consumption. The main difficulty for computing control words is to determine exactly when each item in the control word should be removed. To do this, we need to know when the computation of each predicate is finished. In logic programming terminology this corresponds to determining the success states for all predicates in the derivation. In principle, success states are not observable in SLD resolution other than for the top-level query.

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 whose 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. 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.

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 structural order 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 structural order being used. To handle such stacks, we will use the usual stack operations: empty, which returns an empty stack, $\mathsf{push}(AS, Item)$, 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 7 (*derive*)
Let $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ be a goal with $A_1 \neq \uparrow$. Let $S = \langle G \mid AS \rangle$ be a state and AS be a stack. Let $\lhd$ be a structural order. 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 $S' = \langle G' \mid AS' \rangle$ is *derived* from S and C via $\mathcal{R}$ if the following conditions hold:

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

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 renamed apart copy of A_R, denoted $ren(A_R)$, 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*.

Definition 8 (*pop-derive*)
Let $G = \leftarrow A_1, \ldots, A_k$ be a goal with $A_1 = \uparrow$. Let $S = \langle G \mid AS \rangle$ be a state and AS be a stack. Then $S' = \langle G' \mid AS' \rangle$ 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.

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** or **pop-derive**, can be applied depending on whether the first atom in *curr_goal*(D) is a mark $\uparrow$ or not.

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

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

Note that *derive* steps w.r.t. a clause which is a fact are always followed by a *pop-derive* and thus they are optimized in the figure (and in the implementation, described in Section 6) 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.

Finally, since the goals obtained by ASLD resolution may contain atoms of the form

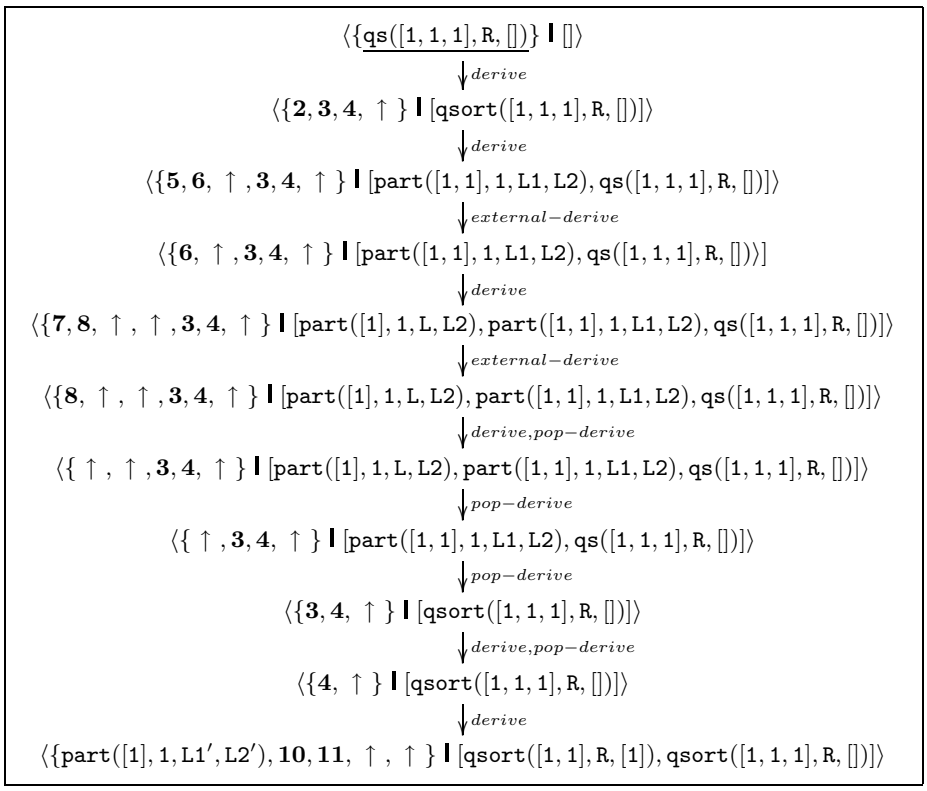

Fig. 4. ASLD Derivation for the example

↑ , 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 ↑ .

It is easy to see that for each ASLD derivation D_S there is a corresponding SLD derivation D with the same computed answer substitution and the same goal without the ↑ 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 ↑ 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 selected. The following notion of *depth-preserving* computation rule allows precisely this.

Definition 9 (*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 $\uparrow \notin \{A_1, \ldots, A_R\}$.

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 initial query G in program P via a *depth-preserving* computation rule. Let D be an SLD derivation using an equivalent local computation rule such that $simplify(D_S) = D$. If,

$$\begin{aligned} curr\_goal(D_S) &= A_1, \ldots, A_n, \uparrow, \ldots \text{ with } A_i \neq \uparrow \quad \text{for } i = 1, \ldots, n. \\ curr\_ancestors(D_S) &= AS \end{aligned}$$

Then, $\mathsf{contents}(AS) = Ancestors(A_i, D)$ for $i = 1, \ldots, n$.

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 $S_0 = \langle G \mid \mathsf{empty} \rangle$.

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 $Admissible(A_R, \mathsf{contents}(\mathsf{empty}), \lhd)$ 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 $AS' = [ren(A_R)]$. Now, we want to prove that $\mathsf{contents}([ren(A_R)]) = Ancestors(B_i, D)$, $i = 1, \ldots, m$, 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. 4), A_R is the only ancestor of B_i in D, $i = 1, \ldots, m$. Consequently, $\mathsf{contents}([ren(A_R)]) = Ancestors(B_i, D)$ 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.

- 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 using an equivalent local computation rule such that $simplify(D_{S-1}) = D'$. 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 $i = 1, \ldots, n$. Then, $\mathsf{contents}(AS_{k-1}) = Ancestors(A_i, D')$ for $i = 1, \ldots, n$.

- 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 $Admissible(A_R, \mathsf{contents}(AS_{k-1}), \lhd)$ 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

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

  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 equivalent 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 4), 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 $\mathsf{contents}(AS_{k-1}) = Ancestors(A_i, D')$, $i = 1, \ldots, n$. In particular, we have that $\mathsf{contents}(AS_{k-1}) = Ancestors(A_R, D')$ for $i = R$. Thus, we have that:

$$\begin{array}{rcl} \mathsf{contents}(AS_k) & = & push(AS_{k-1}, A_R) \\ & = & push(Ancestors(A_R, D'), A_R) \\ & = & Ancestors(B_i, D) \end{array}$$

which proves our claim.

□

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 structural orderings.

Theorem 1 (*accuracy*)
Let D be an SLD derivation for query G in a program P via a local computation rule. Let $\lhd$ be a structural order. If the derivation D is safe w.r.t $\lhd$ 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 for a wqo $\leq_S$ (it can be done similarly for a wfo). 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 $S = \langle G \mid \mathsf{empty} \rangle$ 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. 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 9, 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 the results in (Bruynooghe et al. 1992), 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 an atom of the form $A_1, \ldots, A_n, \ldots$ The equivalent local 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 D_S 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 $\langle A_1, \ldots, A_n, \uparrow, A_R, \ldots \mid [P_1|P] \rangle$ 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 $[P_1|P]$. However, the topmost atom 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 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.

In principle, our local unfolding rule based on ancestor stacks can be used within any partial deduction framework, including Conjunctive Partial Deduction (CPD). 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 clearly improve the efficiency of the partial deduction process.

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 the traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.) or basic input/output facilities. We will also consider as external predicates those defined in a different module, predicates written in another language, etc. This section deals with the difficulties which 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 (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 $\langle \theta_1, \ldots, \theta_n, \bot \rangle$, $n \geq 0$, or an infinite sequence $\langle \theta_1, \ldots, \theta_i, \ldots \rangle$, $i \in \mathbb{N}^*$, where $\mathbb{N}^*$ is the set of positive natural numbers and $\bot$ 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. We now introduce the notion of *evaluable* atom, in order to capture the requirements which allow executing external predicates at partial deduction time.

Definition 10 (*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 an atom call corresponding to an external predicate which is selected during partial deduction, then that atom call can be executed directly at partial deduction time. To formalize this, we propose to 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 express that a certain predicate is evaluable under certain conditions. The following definition introduces the notion of an eval *annotation* as (part of) a computational assertion. We use id to denote the empty substitution, i.e., $\forall\, t\ , \mathsf{id}(t) = t$.

Definition 11 (eval *annotations*)
Let p/n be an external predicate defined in module M. The assertion `:- trust comp p(X1,...,Xn) : SC + eval.` in the code for M is a correct *eval annotation* 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 eval annotations are available, the analysis tools which existing within `CiaoPP` are able to infer eval annotations in many practical cases, 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.

Example 2
The computational assertions in `CiaoPP` for the builtin predicate $\leq$ include, among others, the following one:

```
:- trust comp A =< B : (arithexpr(A), arithexpr(B)) + eval.
```

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 10. The type `arithexpr` corresponds to arithmetic expressions which, as expected, are built out of numbers and the usual arithmetic operators. The type `arithexpr` is expressed in Ciao 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 12 (*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 a renamed apart assertion `:- trust comp p(X1,...,Xn) : SC + eval.` 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
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

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

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

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. As it is

well known, the independence of the computation rule no longer holds for programs with extra logical predicates. This includes binding-sensitive predicates, predicates with side-effects, etc. 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 of the analyzers available in `CiaoPP` with this aim, though this is outside the scope of this article.

6 Experimental Results

We have implemented in our partial deduction system the unfolding rule we propose, together with other variations in order to evaluate the efficiency of our proposal. Our partial deduction 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 1.11#275 (Bueno et al. 2004), with the bytecode generation option. All of our experiments have been performed on a Pentium 4 at 2.4GHz and 512MB RAM running GNU Linux RH9.0. The Linux kernel used is 2.4.25.

The results in terms of execution time and memory consumption are presented in Table 1 and 2, respectively. The programs used as benchmarks are indicated in the **Bench** column. We have chosen a number of classical programs for the analysis and partial deduction of logic programs as benchmarks. In order to factor out the cost of global control, we have used in our experiments initial queries which can be fully unfolded using homeomorphic embedding with ancestors. The program `advisor3` is a variation of the advisor program in the DPPD (Leuschel 2002) library. The programs `query` and `zebra` are classical benchmarks for program analysis. Programs `qsort_80` and `qsort_33` correspond to the quick-sort program shown in the article with pseudo-random lists of natural numbers of length 80 and 33 respectively. `nrev_80` and `nrev_38` correspond to the well-known naive reverse with lists of 80 and 38 natural numbers. `rev_80` is a reverse program with linear complexity which uses an accumulator. The initial query is, as before, a list of 80 natural numbers. Finally, `permute` is a permutation program which uses a nondeterministic deletion predicate. It is partially evaluated w.r.t. a list of 6 and 7 elements respectively. 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 two tables, the following group of columns show execution time and memory consumption, respectively, of the unfolding process with the different implementations of unfolding:

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.

	Execution Times				Relative Speed Up		
Bench	**Relation**	**Trees**	**Stacks**	**MEcce**	**Relation**	**Trees**	**MEcce**
advisor3	144	192	106	1240	1.36	1.81	11.70
nrev_80	mem	106490	15040	64970	∞	7.08	4.32
nrev_38	998	2804	806	4370	1.24	3.48	5.42
permute_7	mem	5226	2800	34680	∞	1.87	12.39
permute_6	476	614	336	3530	1.42	1.83	10.51
query	166	214	116	1290	1.43	1.84	11.12
qsort_80	mem	98514	8970	71870	∞	10.98	8.01
qsort_33	686	2432	454	4580	1.51	5.36	10.09
rev_80	984	1102	960	1400	1.02	1.15	1.46
zebra	1562	2276	994	186620	1.57	2.29	187.75
Overall					mem	7.19	12.25

Table 1. *Comparison of Proof Trees Vs. Ancestor Stacks (Execution Time)*

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.

MEcce We have also measured the time that it takes to process the same benchmarks using Leuschel's M-Ecce (modular Ecce (Leuschel 2002)) system, compiled with the same version of Ciao and in the same machine.

In the case of M-Ecce, we have not provided figures for memory consumption since that would require a deep understanding of M-Ecce implementation in order to make a fair comparison. The last set of columns compare the relative measures of the different approaches w.r.t. the **Stacks** algorithm. Finally, 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.

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. Three entries in the **Relation** column contain the value "mem", instead of a number, to indicate that the partial deduction system has run out of memory. For each of these three cases, we have repeated the experiment with the largest possible initial query that **Relation** can handle in our system before running out of memory. This explains that the three benchmarks are specialized w.r.t. two different initial queries. As it can be seen in the column for relative speedups, **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 column **Trees**, the implementation based on proof trees has a good memory consumption but is slower than **Relation** due to the overhead of traversing the tree for retrieving the ancestors of each atom. In comparison to M-ecce, the results provide evidence that our proof tree-based implementation is indeed comparable to state of the art systems, since the execution times are similar in some cases or even better in others. The last set of columns compares the relative execution times of the different approaches w.r.t. the **Stacks** algorithm which is the fastest in all cases. Indeed, **Stacks** is even faster than the implementation based on explicitly storing all ancestors of all atoms (**Relation**) while having a memory consumption comparable to (and in fact, slightly better than) the implementation based on proof trees. The actual speedup ranges from 1.15 in the case of `rev_80` to 10.98 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 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 three 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 7 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 run the partial deducer. The figures show, for each benchmark, the high water mark minus the memory already in use when the con-

	Memory Consumption			Relative Memory Reduction	
Bench	**Relation**	**Trees**	**Stacks**	**Relation**	**Trees**
advisor3	2932232	819026	634460	4.62	1.29
nrev_80	mem	4000060	3930042	∞	1.02
nrev_38	32738070	859219	817910	40.03	1.05
permute_7	mem	2669486	1198208	∞	2.23
permute_6	7303335	191976	189484	38.54	1.01
query	2841120	7452	6916	410.80	1.08
qsort_80	mem	4175244	3940280	∞	1.06
qsort_33	34359640	861234	810474	42.39	1.06
rev_80	2499152	132736	127556	19.59	1.04
zebra	26287044	81140	75096	350.05	1.08
Overall				∞	1.18

Table 2. *Comparison of Proof Trees Vs.Ancestor Stacks (Memory Consumption)*

struction of the SLD tree was started. In order to make these numbers independent of whether automatic garbage collection is triggered or not during the different experiments, garbage collection has been turned off during these experiments.

As for the case of execution time, the **Stacks** algorithm presents lower consumption than any other algorithm for all programs studied. It can be seen that the 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 (1.18 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 deduction a practical tool.

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 structural order 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 structural orders used, well-quasi orderings in general, and *homeomorphic embedding* (Kruskal 1960; Leuschel and Bruynooghe 2002) in particular, have 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 structural orderings 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 structural orders. The main features of our unfolding rule based on ASLD resolution are: (1) it is parametric w.r.t. the structural order 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 some-

what 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 contexts. 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. 2005) by using backwards analysis to infer purity assertions which determine when a non-leftmost step is safe in the presence of impure predicates.

Acknowledgments

The authors would like to thank the anonymous referees and the participants of LOPSTR'04 for their useful comments on a preliminary version of this article. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Science and Education under the MCYT TIC 2002-0055 *CUBICO* project. Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at University of Roskilde supported by respective grants from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

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. 2005. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. LNCS. Springer-Verlag. To appear.

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. CLIP3/97.1.10(04), School of Computer Science (UPM). August. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

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

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. V., 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.

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.

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. 1996-2002. 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 Static Analysis. *Proceedings of SAS'98*, G. Levi, Ed. LNCS 1503. Springer-Verlag, Pisa, Italy, 230–245.

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., Martens, B., and De Schreye, D. 1998. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20,* 1 (January), 208–258.

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.

Rozenberg, G. and Salomaa, A., Eds. 1997. *Handbook of Formal Languages: Word Language Grammar*. Vol. 1. Springer-Verlag.

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