Efficient Local Unfolding with Ancestor Stacks for Full Prolog

Germán Puebla[1], Elvira Albert[2], and Manuel Hermenegildo[1,3]

[1] School of Computer Science, Technical U. of Madrid, {german,herme}@fi.upm.es
[2] School of Computer Science, Complutense U. of Madrid, elvira@sip.ucm.es
[3] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico, herme@unm.edu

Abstract. 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 structural orders applied over (covering) *ancestors*, i.e., a subsequence of the atoms selected during a derivation. 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 structural order and allows a stack-based implementation without losing any opportunities for specialization. Regarding the second issue, we propose assertion-based techniques which allow our approach to deal with real programs that include (Prolog) built-ins and external predicates in a very extensible manner. 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.

1 Introduction

In spite of the important research efforts in the area, the integration of *Partial Deduction* (PD) [16, 8] methods into compilers seems to be still far from reality. We believe that the general uptake of PD methods is being hindered by two factors: the relative inefficiency of the PD method, and the complications brought about by the treatment of real programs. Indeed, the integration of powerful strategies to 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 PD. A main objective of this paper is to contribute to the uptake of PE techniques by addressing some of these issues.

State-of-the-art partial evaluators integrate terminating unfolding rules for local control based on *structural* orders, like homeomorphic embedding [14] 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 [4], 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, we introduce a novel unfolding rule based on the notion of covering ancestors 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 rely on assertion-based techniques [20]. The use of assertions provides *extensibility* in the sense that users and developers of partial evaluators can deal with new external predicates during PE 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 our 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 [5] and, specifically, its preprocessor `CiaoPP`[9]).

2 Background

We assume some basic knowledge on the terminology of logic programming. See for example [17] 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. 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:*

$$\theta = mgu(A_R, H)$$

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

The definition above differs from standard formulations (such as that in [17]) 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.

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* (PD) [16], 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. 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 $\theta_i(A) \leftarrow G_i$. Therefore, this step returns the set of resultants, i.e., a program, associated to the root-to-leaf derivations of these trees. We refer to [14] for details. In order to ensure the local termination of the PD 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) [4, 18] and well-quasi orderings (wqo) [22, 13] are broadly used in the context of on-line PE techniques (see, e.g., [8, 15, 22]). 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 (and *homeomorphic embedding* [10] in particular) have proved to be very powerful in practice.

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* [3], achieving further specialization in many cases while still guaranteeing termination. The essence of the most advanced techniques is based on the notion of *covering ancestors* [4].

```
qsort([],R,R).                partition([],_,[],[]).
qsort([X|L],R,R2) :-          partition([E|R],C,[E|Left1],Right) :-
   partition(L,X,L1,L2),         E =< C, partition(R,C,Left1,Right).
   qsort(L2,R1,R2),           partition([E|R],C,Left,[E|Right1]) :-
   qsort(L1,R,[X|R1]).           E > C,  partition(R,C,Left,Right1).
```

Fig. 1. A quick-sort program

Definition 3 (ancestor relation). *Given a derivation step and A_R, B_i, $i = 1, \ldots, m$ as in Def. 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.*

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 [4]. 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 Def. 3. It captures the dependency relation implicit within a *proof tree.*

It has been proved [4] 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 $\leftarrow$*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.

Consider now Fig. 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[4] and a superscript which contains the list of ancestors of that atom. Let us assume that we use the *homeomorphic embedding* order [13] 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

[4] 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.

$$\begin{array}{c}
\mathbf{1}.\underline{\mathtt{qs([1,1,1],R,[])}}^{\{\}} \\
\downarrow \\
\mathbf{2}.\underline{\mathtt{p([1,1],1,L1,L2)}^{\{1\}}},\mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}},\mathbf{4}.\mathtt{qs(L1,R,[1|R1])}^{\{1\}} \\
\downarrow \{\mathtt{L1}\mapsto\mathtt{[1|L]}\} \\
\mathbf{5}.\underline{\mathtt{1 =< 1}}^{\{1,2\}},\mathbf{6}.\mathtt{p([1],1,L,L2)}^{\{1,2\}},\mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}},\mathbf{4}.\mathtt{qs([1|L],R,[1|R1])}^{\{1\}} \\
\downarrow \\
\mathbf{6}.\boxed{\underline{\mathtt{p([1],1,L,L2)}}}^{\{1,2\}},\mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}},\mathbf{4}.\mathtt{qs([1|L],R,[1|R1])}^{\{1\}} \\
\downarrow \{\mathtt{L}\mapsto\mathtt{[1|L']}\} \\
\mathbf{7}.\underline{\mathtt{1 =< 1}}^{\{1,2,6\}},\mathbf{8}.\mathtt{p([],1,L',L2)}^{\{1,2,6\}},\mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}},\mathbf{4}.\mathtt{qs([1,1|L'],R,[1|R1])}^{\{1\}} \\
\downarrow \\
\mathbf{8}.\underline{\mathtt{p([],1,L',L2)}^{\{1,2,6\}}},\mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}},\mathbf{4}.\mathtt{qs([1,1|L'],R,[1|R1])}^{\{1\}} \\
\downarrow \{\mathtt{L'}\mapsto\mathtt{[]},\mathtt{L2}\mapsto\mathtt{[]}\} \\
\mathbf{3}.\underline{\mathtt{qs([],R1,[])}^{\{1\}}},\mathbf{4}.\mathtt{qs([1,1],R,[1|R1])}^{\{1\}} \\
\downarrow \{\mathtt{R1'}\mapsto\mathtt{[]}\} \\
\mathbf{4}.\underline{\mathtt{qs([1,1],R,[1])}^{\{1\}}} \\
\downarrow \\
\mathbf{9}.\boxed{\underline{\mathtt{p([1],1,L1',L2')}}}^{\{1,4\}},\mathbf{10}.\mathtt{qs(L2',R1',[1])}^{\{1,4\}},\mathbf{11}.\mathtt{qs(L1',R,[1|R1'])}^{\{1,4\}}
\end{array}$$

Fig. 2. Derivation with Ancestor Annotations

$$\begin{tikzcd}
 & & & \boxed{1} \arrow[dll] \arrow[d] \arrow[drr, Rightarrow] & & & \\
 & 2 \arrow[dl] \arrow[dr] & & 3 & & \boxed{4} \arrow[dl, Rightarrow] \arrow[d] \arrow[dr] & \\
5 & & 6 \arrow[dl] \arrow[dr] & & \boxed{9} & 10 & 11 \\
 & 7 & & 8 & & &
\end{tikzcd}$$

Fig. 3. Proof tree for the example.

atom number **9**, i.e., $\mathtt{p([1],1,L',L2')}$, 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.[5]

This unfolding rule is too conservative, since the process can proceed further without risking termination. The crucial point is that the execution of atom number **9** does not depend on atom number **6** (and, actually, the unfolding of **6** has been already *completed* when atom number **9** is being considered for unfolding). Figure 3 shows the proof tree associated to this derivation where nodes are labeled with the numbers assigned to each atom, instead of the atoms

[5] 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 Sect. 5.

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 Fig. 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 Fig. 2. 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 PE process. We argue that our implementation technique is efficient in time and space, overcoming the above limitations.

4 An Efficient Implementation for Local Unfolding

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

Definition 4 (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 5 (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 (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* [1]. In fact, the ancestor subsequence in any point in time corresponds to the current *control word* [21] 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: using 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.

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). 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 6 (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:*

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

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 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 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 does 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 7 (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. Fig. 4 illustrates the ASLD derivation corresponding to the derivation with explicit ancestor annotations of Fig. 2. Sometimes, rather than writing the atoms themselves, we use the same numbers assigned to the corresponding atoms in Fig. 2. 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. 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.

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

Fig. 4. ASLD Derivation for the example

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 substitution 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 will use $simplify(D_S) = D$ to denote that D is the SLD derivation which corresponds to D_S.

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.

Definition 8 (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 such that $simplify(D_S) = D$. Let $curr\_goal(D_S) = A_1, \ldots, A_n, \uparrow, \ldots$ with $A_i \neq \uparrow$ for $i = 1, \ldots, n$. Let $curr\_ancestors(D_S) = AS$. Then,* $\mathsf{contents}(AS) = Ancestors(A_i, D)$ *for $i = 1, \ldots, n$.*

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 [4].

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 $simplify(D_S) = D$.*

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 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 PD 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 in these cases but in turn the use of local unfolding will clearly improve the efficiency of the PD 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 PD.

When an atom A, such that $pred(A) = p/n$ is an external predicate, is selected during PD, 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 [6] 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 $\langle\theta_1, \ldots, \theta_i, \ldots\rangle$, $i \in I\!N^*$, where $I\!N^*$ 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 PE time, since those effects need to be performed at run-time. We say that an expression is *evaluable* when its execution 1) universally terminates, 2) it does not produce side-effects, 3) it is sufficiently instantiated to be executed, 4) it does not issue errors and 5) it does not generate exceptions. 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, SC can be thought of as a traditional precondition which ensures a certain behaviour of the execution of a procedure provided they are satisfied. To formalize this, we propose to use the "*computational* assertions" which are part of the assertion language [20] of `CiaoPP` 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 9 (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*

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 PD 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. 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.

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 10 (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: 1)* $\sigma = mgu(A_R, p(X1, ..., Xn))$*, 2)* $\mathsf{Exec}(Sys, M, \sigma(SC)) = \langle \mathsf{id} \rangle$*, 3)* $\theta \in Subst(\mathsf{Exec}(Sys, M, A_R))$*, 4)* G' *is the goal* $\theta(A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k)$*, 5)*$AS' = AS$*.*

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 assertion in Example 2 and the atoms **5** and **7**, which are of the form `1=<1`, in the ASLD derivation of Fig. 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 $\mathsf{Exec}(ciao, arithmetic, (1 =< 1)) = \langle \mathsf{id} \rangle$.

	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)

6 Experimental Results

We have implemented in our PD system the unfolding rule we propose, together with other variations in order to evaluate the efficiency of our proposal. Our PD system has been integrated in a practical state of the art compiler which uses global analysis extensively: the `CiaoPP` preprocessor [9]. For the tests, the whole system has been compiled using Ciao 1.11#275 [5], 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 are presented in Table 1. The programs used as benchmarks are indicated in the **Bench** column. We have chosen a number of classical programs for the analysis and PD 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 [12] 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 paper 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 table, the following group of columns show execution time 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.

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 [12]) system, compiled with the same version of Ciao and in the same machine.

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. 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.

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 PD 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, 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.

We have also studied the memory required by the unfolding process (for lack of space details are in [19]). As for the case of execution time, the **Stacks** algorithm presents lower consumption than any other algorithm for all programs studied. 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 the memory figures 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 PD a practical tool.

As for future work, we plan to provide additional solutions for the problems involved in non-leftmost unfolding for programs with extra logical predicates beyond those presented in the literature [11, 7, 2, 14]. In particular, the intensive use of static analysis techniques in this context seems particularly promising. In our case we plan to take advantage of the fact that our PD system is integrated in `CiaoPP` which includes extensive program analysis facilities.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

4. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical Report CLIP3/97.1.10(04), School of Computer Science (UPM), August 2004. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
6. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence Based Abstract Interpretation of Prolog. *Theory and Practice of Logic Programming*, 2(1):25–84, 2002.
7. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York, 1997.
8. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.
10. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
11. Michael Leuschel. Partial evaluation of the "real thing". *Proceedings of LOPSTR'94 and META'94*, Lecture Notes in Computer Science 883, pages 122–137. Springer-Verlag.
12. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
13. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
14. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
15. Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
16. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
17. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.
18. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
19. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. Technical Report CLIP2/2005.0, Technical University of Madrid, February 2005.
20. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
21. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages: Word Language Grammar*, volume 1. Springer-Verlag, 1997.
22. M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.