Efficient Leftmost Unfolding with Ancestor Stacks

Germán Puebla[1] and Elvira Albert[2]

[1] School of Computer Science, Technical University of Madrid, german@fi.upm.es
[2] School of Computer Science, Complutense University of Madrid, elvira@sip.ucm.es

Abstract. The most successful unfolding rules used nowadays in partial evaluation of logic programs 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. The use of ancestor (sub)sequences improves the specialization power of unfolding while still guaranteeing termination and also reduces the number of atoms for which the wfo or wqo has to be checked. Unfortunately, maintaining the structure of the ancestor relation introduces a high operational cost during unfolding.

In an attempt to facilitate the uptake of partial evaluation techniques, we herein present a very efficient unfolding based on the notion of covering ancestors which can be used in combination with any wfo or wqo as long as leftmost unfolding is performed, i.e., the leftmost atom of the goal is always selected for further unfolding. Our implementation technique is able to handle large unfolding trees more efficiently than existing state-of-the-art partial evaluation systems. We believe that our approach makes well-known orderings, like *homeomorphic embedding*, more practically applicable.

1 Background and Motivation

The main purpose of partial evaluation 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* (PD) [11]. Informally, the PD 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; it returns the 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 terms in the right-hand sides of resultants to the set of terms to be partially evaluated; the abstraction phase yields a new set of terms some of which may need further evaluation and, thus, the process is iteratively repeated while new terms are introduced. Following the terminology of [5], the so-called *local*

control defines an unfolding rule which determines when and how to terminate the construction of the SLD trees. The *global* control defines an abstraction operator which guarantees that the number of trees is kept finite. This extended abstract is centered around the local control only (we refer to [9] for a survey on both control issues).

In order to ensure the local termination of the PD algorithm while producing useful specializations, the unfolding rule must incorporate some mechanism to stop the construction of SLD trees. Nowadays, well-founded orderings (wfo) [4, 12] and well-quasi orderings (wqo) [14, 8] are broadly used in the context of online partial evaluation techniques (see, e.g., [5, 10, 14]). Intuitively, derivations are expanded as long as there is some evidence that interesting computations are performed but also guaranteed to terminate (according to the selected ordering). For instance, consider the following program which implements the well-known "`qsort`" algorithm 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]`.

```
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 illustrates an incomplete SLD-derivation for the above program and query, where predicates `qsort` and `partition` are abbreviated as `qs` and `p`, respectively. Each atom is labeled with a number (an identifier) and a superscript for future references[3]. Further resolution steps are allowed as long as new selected atoms are strictly smaller—according to the *homeomorphic embedding* order [8]—than any previously selected atom in the same derivation. Therefore, the derivation stops prematurely when the atom **9**, i.e., $\mathtt{p}([1], \mathtt{1}, \mathtt{L}', \mathtt{L2}')$, is found for the second time, since it is not strictly smaller than the atom **6** selected in the third step (indeed they are equal modulo renaming).

State-of-the-art unfolding rules allow performing ordering comparisons over *subsequences* of the full sequence of the selected atoms of a derivation, achieving further specialization in many cases. To do so, they maintain dependencies over the selected atoms which are chosen in such a way that only a subsequence of

[3] 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.

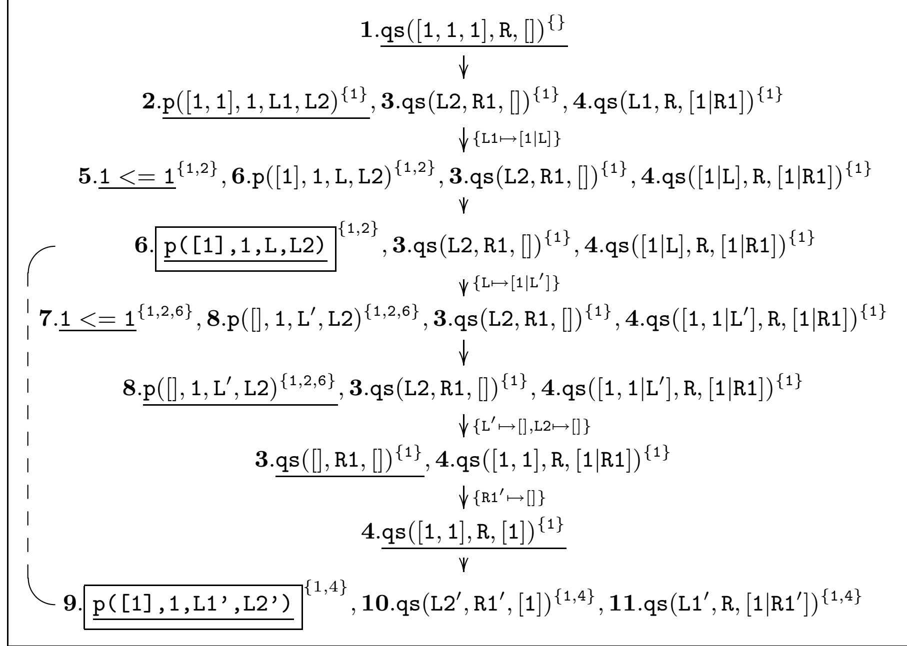

Fig. 1. Derivation with Ancestor Annotations

such selected atoms needs to be considered while still guaranteeing termination. The essence of the most advanced techniques is based on the notion of *covering ancestors* [4]. The important observation is that a derivation can contain selected subgoals which are indeed part of different subcomputations in the proof tree. Given an atom A and a derivation D, we denote by $Ancestors(A, D)$ the sequence of ancestors of A in D. It captures the dependency relation implicit within a *proof tree* [3]. Usually, the test is only applied on *comparable* ancestor atoms, i.e., ancestor atoms with the same predicate symbol. This corresponds to the original notion of covering ancestors [4].

In the above example, the proof tree associated to the derivation depicted in Figure 1 is shown in Figure 2, where we use the numbers assigned to the nodes in the tree rather than writing the precise atoms.

Therefore, in order to decide whether to evaluate or not atom **9**, one has to check only that it is strictly smaller than atoms **4** and **1**, i.e., its ancestors according in the proof tree. By considering the full sequence, the atom was compared with atom **6** which results in considering it a dangerous derivation (as shown in Fig. 1). Note that the SLD tree for the example query is finite and the query can be safely fully unfolded. 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 in its *ancestor sequence*. In wfo, it is sufficient to verify that the selected atom is strictly smaller than the previous comparable one (if one exists)

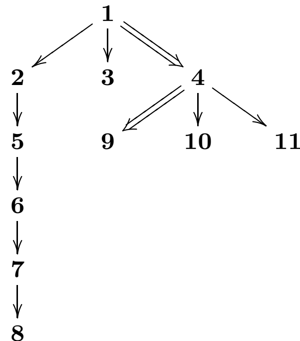

Fig. 2. Proof tree for the example.

in its ancestor sequence. The most successful techniques to-date are based on these two basic ingredients:

- The use of a structural order (wfo or wqo) which guarantees termination while still capturing structural aspects of programs and goals.
- Organizing the atoms already visited in each derivation in a (proof) tree rather than a linear collection, such as a sequence or a set.

Among the structural orders, well-quasi orderings (and *homeomorphic embedding* [7] 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, as our example illustrates. Unfortunately, the practical applicability of such unfolding rules is threatened by the overhead introduced by the implementation of the notion of covering ancestor. For the simple derivation of Fig. 1, we indicate (inside a superscript) the list of ancestors which each atom has to maintain during its evaluation. Our experiments show that the cost of maintaining such ancestor information grows very quickly with the size of derivations in the unfolding tree. In an attempt to facilitate the uptake of partial evaluation techniques, we propose a novel implementation technique for the practical integration of unfolding rules based on the notion of ancestor into state-of-the-art partial evaluators. Although further experimentation is still required, we believe our preliminary results are promising.

2 An Efficient Implementation for Leftmost Unfolding

By *leftmost unfolding* we refer to a computation rule which always selects the leftmost atom of a goal in order to perform resolution. I.e., given the goal $\leftarrow \sigma(A_1, \ldots, A_n)$, the atom $\sigma(A_1)$ will always be selected for resolution. This corresponds to the computation rule used in most logic programming languages

such as Prolog. SLD resolution restricted to the case of leftmost unfolding is referred to as LD resolution. The use of computation rules which are not leftmost during partial deduction is problematic in several ways: (1) it is well-known that if the resulting program is to be executed under the Prolog left-to-right computation rule, the specialized program may perform more computation steps than the original one due to the introduction of choice-points; (2) if the program contains meta-logical predicates such as `var/1` or `ground/1`, the resulting code can simply be incorrect unless back-propagation of bindings is avoided; (3) if the program contains calls to builtin predicates, there will often be a left-to-right information flow. Thus, unless this computation order is preserved at specialization time, chances are that calls to builtins will not be sufficiently instantiated for being reduced. As a result of all the above, leftmost unfolding during partial deduction is arguably a very sensible thing to do.

2.1 LD Resolution with Ancestor Stacks

Unfolding rules based on wfo or wqo usually monitor the construction of derivations during specialization and stop unfolding as soon as a sequence which is not admissible is found. Intuitively, a sequence of elements $s_1, s_2, \ldots$ in S is called *admissible with respect to an order* $\leq_S$ [4] iff there are no $i < j$ such that $s_i \leq_S s_j$. It has been proved [4] that any infinite derivation must have at least one infinite covering ancestor sequence. Therefore, it is sufficient to check the above ordering relation $\leq_S$ over the covering ancestor subsequences in order to detect inadmissible derivations. There are a number of well-known orders in the literature which allow the definition of *admissible* sequences that are always finite for which our technique directly applies.

A central observation which motivated this work is that the proof tree which is used in order to capture the ancestor relation can be interpreted as an *activation tree* [1]. Since such proof tree is traversed depth-first, left-to-right during LD resolution, the notion of ancestors for SLD resolution corresponds to the notion of *control word* [13]. The control word for each execution state can be seen as the set of procedures whose execution has started and is not yet completed. In order to compute control words it is essential to know exactly when each item of the stack should be popped. Such information is readily available in proof trees: an atom A_i in the stack should be removed as soon as the depth-first traversal moves up in the proof tree from such atom A_i to another atom A_j such that A_j is the father of A_i.

The difficulty relies in that success states (end of the execution of a call) are not observable in SLD (nor LD) resolution other than for the top-level query. We herein propose an easy-to-implement modification of LD resolution in which success states for all internal calls are observable. This involves adding marks to the goals used during resolution which indicate that the last procedure which has been started is now finished. The atom corresponding to the last started procedure will be on the top of the stack and thus can be popped. Essentially, in the augmented operational semantics we propose, goals are enhanced with an *ancestor stack*, which at each stage of the computation contains the atoms whose

computation has started but has not yet finished, i.e., *the ancestors of the next atom which will be selected for resolution.*

Although developed in a different context and for completely different purposes, the OLDT semantics [15] used in logic programming with tabulation also augments LD semantics in order to explicitly represent procedure call and return states which are needed for the table where previously computed answers are stored.

Let us state some notation. In our operational semantics, states are 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 allow disabling resolution as soon as termination of the resolution process can no longer be guaranteed by the structural order being used. Let $<$ be a wfo. By $Admissible_{wfo}(A, (A_1, \ldots, A_n), <)$, with $n \geq 0$, we denote the truth value of the expression $A < A_1$ if $n \geq 1$ and $true$ if $n = 0$. Let $\leq_S$ be a wqo. By $Admissible_{wqo}(A, (A_1, \ldots, A_n), \leq_S)$, with $n \geq 0$, we denote the truth value of the expression $\forall A_i : A \leq_S A_i$. We will denote by *structural order* a wfo or a wqo (written as $\triangleleft$ to represent any of them).

To handle stacks, we will use the usual stack operations: empty, which returns an empty stack, $\mathsf{push}(S, Item)$, which pushes $Item$ onto the stack S, and $\mathsf{pop}(S)$, which pops an element from S. In addition, we will use the operation $\mathsf{contents}(S)$, which returns the sequence of atoms contained in S in the order in which they would be popped from the stack S and leaves S unmodified.

Definition 1 (reduction step in LD with ancestor stack). *Let $\sigma_i(A_1, \ldots, A_n)$ be a goal and AS_i be a stack. Let $\triangleleft$ be a structural order. A* reduction step in LD with ancestor stack *from $\langle \sigma_i(A_1, \ldots, A_n) \mid AS_i \rangle$ is obtained by applying one of the reduction rules:*

resolve *If the two following conditions are satisfied:*

1. *There is a renamed apart rule $H \leftarrow B_1, \ldots, B_m$ in P with $\theta = mgu(\sigma_i(A_1), H)$, and*
2. *$Admissible_{\triangleleft}(\sigma_i(A_1), \mathsf{contents}(AS), \triangleleft)$ holds*

then $\langle \sigma_i(A_1, \ldots, A_n) \mid AS_i \rangle$ is reduced to:

$\langle \sigma_{i+1}(B_1, \ldots, B_m, \$pop, A_2, \ldots, A_n) \mid AS_{i+1} \rangle$, where $\sigma_{i+1} = \sigma_i\theta$ and $AS_{i+1} = \mathsf{push}(AS_i, ren(\sigma_i(A_1)))$.

builtin *If the three following conditions are satisfied:*

1. *If $\sigma_i(A_1)$ is an atom corresponding to a builtin predicate, and*
2. *$eval(\sigma_i(A_1))$ holds, and*
3. *$call(\sigma_i(A_1))$ succeeds with substitution θ,*

then $\langle \sigma_i(A_1, \ldots, A_n) \mid AS_i \rangle$ is reduced to $\langle \sigma_{i+1}(A_2, \ldots, A_n) \mid AS_{i+1} \rangle$, where $\sigma_{i+1} = \sigma_i\theta$ and $AS_{i+1} = AS_i$.

pop *If* $\sigma_i(A_1) = \$pop$ *then it is reduced to* $\langle \sigma_{i+1}(A_2, \ldots, A_n) \mid AS_{i+1} \rangle$*, where* $\sigma_{i+1} = \sigma_i$ *and* $AS_{i+1} = \mathsf{pop}(AS_i)$*.*

The `resolve` rule is non-deterministic if several clauses in P unify with the atom $\sigma(A_1)$. Note that, for each state, at most one of the three resolution rules can be applied. Thus, no further non-determinism is introduced by LD resolution with ancestor stack which is not already present in LD resolution.

Intuitively, the **resolve** rule corresponds to performing one resolution step for the leftmost atom $\sigma(A_1)$ in the current goal. In addition, the mark *$pop* is added to the goal, where *$pop* is a pseudo-atom which is guaranteed not to clash with any existing predicate name. Finally, a renamed apart copy of $\sigma(A_1)$, denoted $ren(\sigma(A_1))$, is pushed onto the ancestor stack. Note that in contrast to traditional LD resolution, this rule can only be applied if the current leftmost atom together with its ancestors do constitute an admissible sequence.

The **builtin** rule is useful for evaluating calls to builtin predicates, or in general predicates which can be executed but whose code is not available. In many cases, builtin predicates impose certain degree of instantiation for being executed. This is indicated by the boolean function *eval* which returns *true* only if (1) the atom $\sigma(A_1)$ is sufficiently instantiated and (2) the execution of $\sigma(A_1)$ does not produce side-effects. In the latter case, the builtin cannot be executed until run-time regardless of its instantiation state at specialization time. Unlike ordinary predicates, it is assumed that the execution of builtins (1) is guaranteed to terminate finitely if $eval(\sigma(A_1))$ takes the value *true*, and (2) does not perform calls to the predicates defined in P. Thus, there is no need to push the atom $\sigma(A_1)$ into AS since there can be no calls from builtin predicates to other predicates defined in P.

Finally the **pop** rule, unlike the two previous rules, does not provide any new bindings. This rule is used when the leftmost atom in the resolvent is a mark *$pop*. 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 $\leftarrow G$ starts from the state $S_0 = \langle id(G) \mid \mathsf{empty} \rangle$. We use $S \leadsto_P S'$ to indicate that in program P a reduction can be applied to state S to obtain state S'. Also, $S \leadsto_P^* S'$ indicates the transitive closure of this relation. A *derivation* for a query $\leftarrow G$ in program P is a sequence of states $S_0 \leadsto_P S_1 \leadsto_P \ldots \leadsto_P S_n$ where $S_0 = \langle id(G) \mid \mathsf{empty} \rangle$ and there is a reduction from each S_i to S_{i+1}. 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 of the derivation, respectively. E.g., if D is the derivation $S_0 \leadsto_P^* S_n$ with $S_n = \langle \sigma(G) \mid AS \rangle$ then $curr_goal(D) = \sigma(G)$ and $curr_ancestors(D) = AS$. Since the resultants obtained by LD derivation with ancestor stack may contain atoms of the form *$pop*, resultants are cleaned up before being transferred to the global control level or during the code generation phase by simply eliminating all atoms of the form *$pop*.

Example 1. The derivation with explicit ancestor annotations depicted in Fig. 1 corresponds to the LD derivation with ancestor stack which appears in Fig. 3.

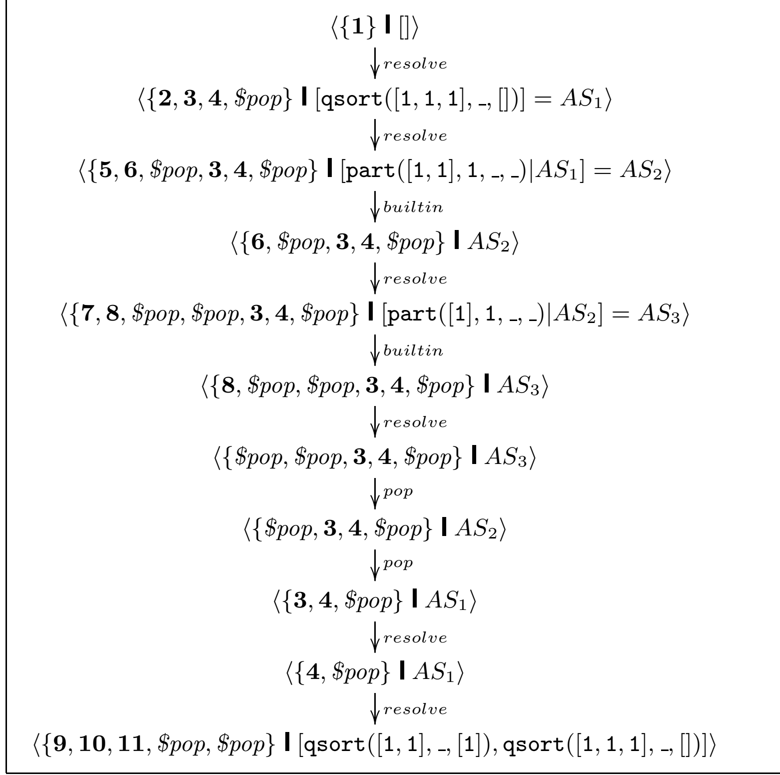

Fig. 3. LD Derivation with Ancestor Stack

Each step is appropriately labeled with the applied reduction rule (the c.a.s. are omitted here). At each state, we write the list of numbers assigned to the corresponding atoms rather than the terms itself by using the labeling of Fig. 1. By abuse of notation, we always use the same number assigned to an atom although further instantiation is performed. However, the stack contains the list of atoms exactly in the instantiation state they have when they are pushed in the stack. Note that *resolve* steps w.r.t. a clause which is a fact are optimized in the figure (and in the implementation) by not pushing $\sigma(A_1)$ onto the stack and not including a *$pop* mark into the goal which would immediately pop $\sigma(A_1)$ from the stack.

It should be noted that, in the last step, the stack contains exactly the ancestors of `partition([1],1,L1,L2)`, since the previous calls to `partition` have already finished and thus their corresponding atoms have been popped off the stack. Consequently, unfolding can proceed further without termination risk. Indeed, the derivation can be totally unfolded which results in the following (optimal) partial evaluation in which all input data have been satisfactorily consumed: `qsort([1,1,1],[1,1,1],[])`.

It is easy to see that for each LD derivation with ancestor stack D_S there is a corresponding LD derivation D which computes the same computed answer substitution and the same resultant modulo the *$pop* atoms. Such derivation is the one obtained in LD resolution by performing the *resolve* (with exactly the same clauses) and *builtin* steps and by ignoring the *pop* steps. We will use $simplify(D_S) = D$ to denote that D is the LD derivation which corresponds to D_S.

Proposition 1 (ancestor stack). *Let D_S be a LD derivation with ancestors stack for initial query $\leftarrow G$ in program P such that $simplify(D_S) = D$. Let $curr_goal(D_S) = A_1, \ldots, A_n$. Let $curr_ancestors(D_S) = AS$. Then $Ancestors(A_1, D) = \mathsf{contents}(AS)$.*

Correctness of this proposition is central to proving correctness of the two theorems below.

Theorem 1 (termination). *Let $\leftarrow G$ be a query, P be a program, $\mathcal{B}$ a set of builtins, and let $\lhd$ be a structural order. Then the derivation tree obtained by applying reduction steps in LD with ancestor stack as defined in Def. 1 is finite.*

Theorem 1 guarantees termination at the local level during partial deduction. It follows from Proposition 1 and the fact that the use of ancestor subsequence guarantees local termination [4].

Theorem 2 (accuracy). *Let $D = G_0, \ldots, G_n$ be a LD derivation for query $\leftarrow G_0$ in a program P with set of builtins $\mathcal{B}$ and whose selected atoms are $A_0, \ldots, A_{n-1}$. Let A_n be the selected atom in G_n. Let $\lhd$ be a structural order. If $Admissible(A_n, Ancestors(A_n, D), \lhd)$ holds then there exists a LD derivation with ancestor stack D_S such that*
$simplify(D_S) = D$.

Theorem 2 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". Note that since our semantics disables further resolution as soon as inadmissible sequences are detected, not all LD derivations have a corresponding LD derivation with ancestor stack. However, if the LD derivation is admissible, then its corresponding D_S derivation can be found.

3 Discussion and Future Work

The use of ancestors for refining sequences of visited atoms was early proposed [4] and important effort has been devoted to improve the implementation of ancestors [12]. However, current state-of-the art partial deduction systems (e.g., the Ecce PD system or the Curry partial evaluator [2]) are not be very efficient when local control based on the combination of structural orderings and ancestors is used. Mainly, due to the fact that one has to maintain dependency information

for the individual atoms in each derivation. In principle, the use of ancestors should not only allow more powerful transformation but also speedup 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 seems to neglect the theoretical efficiency gains.

In this work we propose an extension over the LD semantics which can be used as a basis for the generation of (incomplete) SLD trees during partial deduction in combination with structural orders and ancestor sequences. The main features of the operational semantics we propose 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 LD 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. 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 `CiaoPP` [6], the `Ciao` system preprocessor. Preliminary experimental results are very promising: the query to `qsort` with a list of identical numbers can be fully unfolded, which can only be done if ancestors rather than the full sequence is considered, and the time required to do so for the particular case of leftmost unfolding is significantly lower than the time required by other state-of-the-art partial evaluation systems. Though it can be argued that specialization time is not very relevant, the takeup of partial deduction techniques also depends on the power of existing systems and tools which we hope our proposal can contribute to its improvement.

In an extended version of this abstract, we plan to handle several lines of ongoing and future work. Though our technique has been originally developed for leftmost unfolding, it can be applied for other less restrictive unfolding rules which may select any of the "most recent" atoms in the goal, i.e., the computation rule may select any of the siblings of the leftmost atom as far as the first *$pop* pseudo-atom appears. Moreover, we plan to perform a thorough experimental evaluation of the proposed techniques both in terms of resources required to perform the specialization, i.e., time and memory consumption, and quality of the resulting program (together with a comparison w.r.t. state-of-the-art systems). Also, we are working on the extension of our unfolding technique for full Prolog, including the precise formulation of *eval* and *call* functions.

Acknowledgments

The authors would like to thank the anonymous referees for their useful comments. This work has been supported in part by the Information Society Technologies programme of the European Commission, Future and Emerging Tech-

nologies under the IST-2001-38059 "ASAP" project, by MCYT project TIC 2002-0055 "CUBICO" and FEDER infrastructure UNPM-E012.

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. 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.
6. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
7. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
8. 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.
9. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming, Special issue on program development*, 2(4 & 5):461–515, July 2002.
10. 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.
11. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
12. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
13. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages: Word Language Grammar*, volume 1. Springer-Verlag, 1997.
14. 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.
15. H. Tamaki and M. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.