Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation

Germán Puebla* John Gallagher† Manuel Hermenegildo*

(Extended Abstract)

1 Introduction

Partial evaluation [JGS93, DGT96] specializes programs for known values of the input. Partial evaluation of logic programs has received considerable attention [Neu90, LS91, Sah93, Gal93, Leu97] and several algorithms parameterized by different control strategies have been proposed which produce useful partial evaluations of programs. Regarding the correctness of such transformations, two conditions, defined on the set of atoms to be partially evaluated, have been identified which which ensure correctness of the transformation: "closedness" and "independence" [LS91].

From a practical point of view, effectiveness, that is, finding suitable control strategies which provide an appropriate level of specialization while ensuring termination, is a crucial problem which has also received considerable attention. Much work has been devoted to the study of such control strategies in the context of "on-line" partial evaluation of logic programs [MG95, LD97, LM96]. Usually, control is divided into components: "local control," which controls the unfolding for a given atom, and "global control," which ensures that the set of atoms for which a partial evaluation is to be computed remains finite.

In most of the practical program specialization algorithms, the above mentioned control strategies use, to a greater or lesser degree, information generated by static program analysis. One of the most widely used techniques for static analysis is abstract interpretation [CC77, CC92]. Some of the relations between abstract interpretation and partial evaluation have been identified before [GCS88, GH91, Gal92, CK93, PH95, LS96]. However, the role of analysis is so fundamental that it can be asked whether partial evaluation could be achieved directly by a generic abstract interpretation system such as [Bru91, MH92, CV94]. With this question in mind, we present a method for generating a specialized program directly from the output (an and–or graph) of a generic abstract interpreter, in particular the PLAI system [MH89, MH90, MH92]. We then explore two main questions which arise. Firstly, how much specialization can be performed by an abstract interpreter, compared to partial evaluation? Secondly, how do the traditional problems of local and global control appear when placed in the setting of generic abstract interpretation? We conclude that although further study is needed, there seem to be some practical and conceptual advantages in using an abstract interpreter to perform program specialization.

2 Abstract Interpretation

Abstract interpretation [CC77] is a technique for static analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D).

*Department of Computer Science, Technical University of Madrid, {german,herme}@fi.upm.es
†Department of Computer Science, University of Bristol, john@compsci.bristol.ac.uk

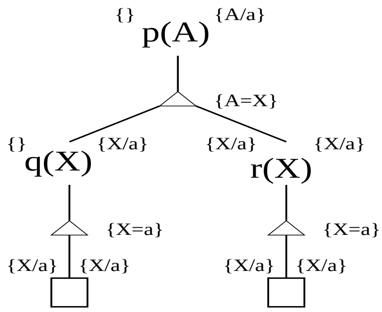

Figure 1: And–or analysis graph

Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$.

Goal dependent abstract interpretation takes as input a program P, a predicate symbol[1] p (denoting the exported predicate), and, optionally, a restriction of the run-time bindings of p expressed as an abstract substitution λ. Such an abstract interpretation computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$ such that $\forall i = 1..n\ \forall\theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$. Additionally, $\forall p_i\theta_i$ that occurs in the concrete computation of $p\theta$ s.t $\theta \in \gamma(\lambda)$ where p is the exported predicate and λ the description of the initial calls of p $\exists\langle p_j, \lambda_j^c, \lambda_j^s\rangle \in Analysis(P, p, \lambda, D_\alpha)$ s.t. $p_i = p_j$ and $\theta \in \gamma(\lambda_j^c)$. This condition is related to the closedness condition usually required in partial evaluation. In abstract interpretation, $\perp$ denotes the abstract substitution such that $\gamma(\perp) = \emptyset$. A tuple $\langle p_j, \lambda_j^c, \perp\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions. An analysis is said to be *multivariant* on calls if more than one triple $\langle p, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate.[2] If analysis is multivariant on successes, the triples in $Analysis(P, p, \lambda, D_\alpha)$ will be of the form $\langle p_i, \lambda_i^c, S_i^s\rangle$ where $S_i^s = \{\lambda_{i_1}^s, \ldots, \lambda_{i_j}^s\}$ with $j > 0$. Different analyses may be defined with different levels of multivariance [VDCM93]. However, unless the analysis is multivariant on calls, little specialization may be expected in general. Due to space limitations we will limit the discussion to analyses (such as the original analysis algorithm in PLAI) which are multivariant on calls but not on successes, though multivariant successes can also be captured by certain abstract domains, as will be discussed in Section 5. In our case, in order to compute $Analysis(P, p, \lambda, D_\alpha)$, an and–or graph is constructed which encodes dependencies among the different triples. Note that when several success substitutions have been computed for the same or–node, the different substitutions have to be summarized in a more general one (possibly losing accuracy) before propagating this success information. This is done by means of the *least upper bound* (lub) operator. Finiteness of the and–or graph (and thus termination of analysis) is achieved by considering abstract domains with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [CC77].

Example 2.1 Consider the simple example program below taken from [Leu97]. Figure 1 depicts a possible result of analysis for the goal $p(A)$ with A unrestricted using the concrete domain as abstract domain.

```
p(X):- q(X), r(X).
q(a).
r(a).
r(b).
```

[1] Extending the framework to sets of predicate symbols is trivial.

[2] If $n = 0$ then the corresponding predicate is not needed for solving any goal in the considered class (p, λ) and is thus dead-code and may be eliminated.

We do not describe here how to build analysis and–or graphs. Details can be found in [Bru91, MH90, MH92]. The graph has two sorts of nodes: those which correspond to atoms (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda^c_i, \lambda^s_i \rangle$. For clarity, in the figures the atom is superscripted with λ^c to the left and λ^s to the right of the atom respectively. For example, the or–node $\langle p(A), \{\}, \{A/a\} \rangle$ is depicted in the figure as ${}^{\{\}}p(A)^{\{A/a\}}$. And–nodes are pairs $\langle Id, Subs \rangle$ where Id is a unique identifier for the node and $Subs$ represents the head unifications of the clause the node refers to. In the figures, they are represented as triangles and the head unifications are depicted to their right. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs to and–nodes which represent the clauses with which the atom (possibly) unifies. Clearly, if an or–node has no children, the atom will fail. And–nodes have arcs to or–nodes for the corresponding predicate p and call pattern λ^c. If a node $\langle p, \lambda^c, \_ \rangle$ is already in the tree it becomes a recursive call and λ^s is obtained by means of a fixpoint computation. □

3 Code Generation from an And–Or Graph

After introducing some notation we present an algorithm which generates a logic program from an analysis and–or graph. This idea was already exploited in [Win92, PH95]. A *program* P is a sequence of clauses of the form H :- B where H is an atom and B is a possibly empty conjunction of atoms. The sequence of clauses in a program which define a predicate p is denoted by $def(p)$. We denote by $or(AO)$ the set of or–nodes in an and–or graph AO. Given a node N, $children(N)$ is the sequence of nodes $N_1 :: \ldots :: N_n$ $n \geq 0$ such that there is an arc from N to N' in AO iff $N' = N_i$ for some i and $\forall i,j = 0,\ldots,n$ N_i is to the left of N_j in AO iff $i < j$. Note that $children(N)$ may be applied both to or– and and–nodes. We assume the existence of an injective function pred which given $AO(P,p,\lambda,D_\alpha)$ returns a unique predicate name for each or–node in the graph and $\mathsf{pred}(\langle p(\bar{t}), \lambda, \lambda^s \rangle) = p$ iff $\langle p(\bar{t}), \lambda, \lambda^s \rangle$ is a root node in the graph (to ensure that top-level – exported – predicate names are maintained).

Definition 1 (partial concretization) Given an abstract substitution λ, a substitution θ is a *partial concretization* of λ and is denoted $\theta \in part\_conc(\lambda)$ iff $\forall \theta' \in \gamma(\lambda)$ $\exists \theta''$ s.t. $\theta' = \theta\theta''$.

$part\_conc(\lambda)$ can be regarded as containing (part of) the definite information about concrete bindings that the abstract substitution λ captures. Note that different partial concretizations of an abstract substitution λ with different accuracy may be considered. For example if the abstract domain is a depth-k abstraction and $\lambda = \{X/f(f(Y)) or X/f(a)\}$, a most accurate $part\_conc(\lambda)$ is $\{X/f(Z)\}$. Note also that $\forall \lambda$ $\epsilon \in part\_cont(\lambda)$ where ϵ is the empty substitution.

Basically, the algorithm for code generation (Algorithm 1) given below creates a different version for each different (abstract) call substitution λ^c to the predicate p in the original program. This is easily done by associating a version to each or–node. Note that if we always take the trivial substitution ϵ as $part\_conc(\lambda)$ for any λ (such as in [PH95]) then such versions are identical except that atoms in clause bodies are renamed to always call the appropriate version. The interest in doing the proposed multiple specialization is that the new program may be subject to further optimizations which were not possible in the original program. Additionally, in Algorithm 1 predicates whose success substitution is $\perp$ are directly defined as $p(\bar{t}) : -fail$, as it is known that they produce no answers. Even if the success substitution for the predicate (or–node) is not $\perp$, individual clauses for p whose success substitution is $\perp$ (useless clauses) are removed from the final program.

Algorithm 1 (Code Generation) Given an analysis and–or graph $AO(P,p,\lambda,D_\alpha)$ generated by analysis for a program P and an atomic goal $\leftarrow p$ with abstract substitution $\lambda \in D_\alpha$ do:

- For each non-empty or–node $N = \langle a(\bar{t}), \lambda^c, \lambda^s \rangle \in or(AO(P,p,\lambda,D_\alpha))$ generate a distinct predicate with name $pred_N = \mathsf{pred}(\langle a(\bar{t}), \lambda^c, \lambda^s \rangle)$.

- Each predicate $pred_N$ is defined by
  - $pred_N(\bar{t})$ `:-` $fail$ if $\lambda^s = \bot$
  - $(pred_N(\bar{t}_1)$ `:-` $b'_1)\theta_1 :: \ldots :: (pred_N(\bar{t}_n)$ `:-` $b'_n)\theta_n$ provided that $def(p) = p(\bar{t}_1)$ `:-` $b_1 :: \ldots :: p(\bar{t}_n)$ `:-` b_n otherwise
- Let $children(N) = \langle Id_1, unif_1\rangle :: \ldots :: \langle Id_i, unif_i\rangle :: \ldots :: \langle Id_n, unif_n\rangle$
  Let $children(\langle Id_i, unif_i\rangle) = \langle a_{i1}(\bar{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1}\rangle :: \ldots :: \langle a_{ik_i}(\bar{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i}\rangle$.
- Each body b'_i is defined as
  - $b'_i = fail$ if $\lambda^s_{ik_i} = \bot$
  - $b'_i = (pred_{i1}(\bar{t}_{i1}), \ldots, pred_{ik_i}(\bar{t}_{ik_i}))$
    where $pred_{ij} = \mathsf{pred}(\langle a_{ij}(\bar{t}_{ij}), \lambda^c_{ij}, \lambda^c_{ij}\rangle)$ otherwise
- Each substitution θ_i is defined as
  - $\theta_i = \epsilon$ if $b'_i = fail$
  - $\theta_i = \theta_{i1} \ldots \theta_{ik_i}$ provided that
    $\theta_{ij} \in part\_conc(\lambda^s_{ij})$ $j = 1 \ldots k_i$ otherwise

Note that in Algorithm 1 atoms are specialized w.r.t. answers rather than calls as in traditional partial evaluation. This cannot be done for example if the program contains calls to extra-logical predicates such as `var/1`. Other more conservative algorithms can be used for such cases and for programs with side-effects. Using Algorithm 1 it is sometimes possible to detect infinite failures of predicates and replace predicate definitions and/or clause bodies by `fail`, which is not possible in partial evaluation. Additionally, as mentioned above, dead-code, i.e., clauses not used to solve the considered goal are removed.

Theorem 3.1 Let $AO(P, p, \lambda, D_\alpha)$ be an analysis and–or graph for a definite program P and an atomic goal $\leftarrow p$ with the abstract call substitution $\lambda \in D_\alpha$. Let P' be the program obtained from $AO(P, p, \lambda, D_\alpha)$ by Algorithm 1. Then $\forall \theta_c$ s.t. $\theta_c \in \gamma(\lambda)$

i) $p\theta_c$ succeeds in P' with computed answer θ_s iff $p\theta_c$ succeeds in P with computed answer θ_s.

ii) if $p\theta_c$ finitely fails in P then $p\theta_c$ finitely fails in P'.

Thus, both computed answers and finite failures are preserved. However, the specialized program may fail finitely while the original one loops (see Example 4.2).

4 And–Or Graphs Vs. SLD Trees

It is known [LS96] that the propagation of success information during partial evaluation is not optimal compared to that potentially achievable by abstract interpretation.

Example 4.1 Consider the program and goal of Example 2.1. The program obtained by applying Algorithm 1 to the and–or graph in Figure 1 is:

```
p(a):- q(a), r(a).
q(a).
r(a).
```

Note that Algorithm1 may perform some degree of specialization even if no unfolding is performed. The information in $AO(P, p, \lambda, D_\alpha)$ allows determining that the call to `r(X)` will be performed with `X=a` and thus the second clause for `r` can be eliminated. Such information can only be propagated in partial evaluation by unfolding the atom `q(X)`. □

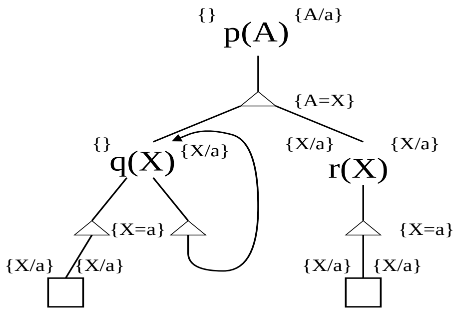

Figure 2: Recursive And–or analysis graph

Example 4.2 Consider again the goal and program of Example 2.1 to which a new clause `q(X):- q(X).` is added for predicate `q`. The and–or graph for the new program is depicted in Figure 2. The program generated for this graph by Algorithm 1 is the following:

```
p(a):- q(a), r(a).
q(a).
q(a):- q(a).
r(a).
```

The fact that `r(X)` will only be called with `X=a` cannot be determined by any finite unfolding rule. Note that the original program loops for the goal $\leftarrow$ `p(b)` while the specialized one fails finitely. □

The two examples above show that and–or graphs allow a level of success information propagation not possible in traditional partial evaluation, either because the unfolding rule is not aggressive enough (Example 4.1) or because the required unfolding would be infinite (Example 4.2). This suggests the possible interest of integrating full partial evaluation in an analysis/specialization framework based on abstract interpretation.

In addition, the fact that such a framework can work uniformly with abstract or concrete substitutions makes it more general than partial evaluation and may allow performing optimizations not possible in the traditional approaches to partial evaluation. For example, based on this idea, in [GH91, PH95, PH97], a framework for "abstract partial evaluation" was presented in which abstract substitutions were used to perform program optimizations. These optimizations used the notion of "abstract executability," which allows reducing calls to some (built-in) predicates at compile-time to the values true or false, or to a set of unifications. An additional pragmatic motivation for this work is the availability of off-the-shelf generic abstract interpretation engines such as PLAI [MH92] or GAIA [CV94] which greatly facilitate the efficient implementation of analyses. The existence of such an abstract interpreter in advanced optimizing compilers is very likely, and using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler [GH91, PH95].

5 Partial Evaluation using And–Or Graphs

We have established so far that for any abstract interpretation in the PLAI system (even interpretations over very simple domains such as modes) we can get some corresponding specialized source program with possibly multiple versions by applying Algorithm 1. Correctness of abstract interpretation ensures that the set of triples computed by analysis must cover all calls performed during execution of any instance of the given initial goal (p, λ). This condition is closely related to the closedness condition of partial evaluation [LS91]. Furthermore there are well-understood conditions and methods for ensuring termination of an abstract interpretation.

Thus, an important conceptual advantage of formalizing partial evaluation in terms of abstract interpretation is that two of the main concerns of partial evaluation algorithms – namely correctness and termination – are guaranteed by the general principles of abstract interpretation. The other important concern is the degree of specialization that is achieved, which is determined in partial evaluation by the local and global control. We now examine how these control issues appear in the setting of abstract interpretation.

5.1 Global Control in Abstract Interpretation

Effectiveness of specialization greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which a specialized version is to be generated. In partial evaluation, this mainly depends on the global control used. If we use the specialization framework based on abstract interpretation, the number of specialized versions depends on the number of or–nodes in the analysis graph. Assuming that the level of multivariance of analysis is fixed (multivariant on calls but not on successes) this is controlled by the choice of abstract domain and widening operators (if any). The finer-grained the abstract domain is, the larger the set $\mathbf{A}$ will be. In conclusion, the role of so-called global control in partial evaluation is played in abstract interpretation by our particular choice of abstract domain and widening operators (which are strictly required when the abstract domain is infinite).

The specialization framework we propose is very general. Depending on the kind of optimizations we are interested in performing, different domains should be used and thus different sets $\mathbf{A}$ will be obtained. For example, if we are interested in eliminating redundant groundness tests, our abstract domain could in principle collapse the two atoms $p(1)$ and $p(2)$ into one $p(ground)$ as from the point of view of the optimization, whether p is called with the value 1 or 2 is not relevant.

While the main aim of global control is to ensure termination and not to generate too many superfluous versions, it may often be the case that global control (or the domain) does not collapse two versions in the hope that they will lead to different optimizations. If this is not the case, a minimizing step may be performed a posteriori on the and–or graph in order to produce a minimal number of versions while maintaining all optimizations. This was proposed in [Win92], implemented in [PH95] and also discussed in [LM95]. We intend to extend the minimizing algorithm in [PH95] for the case of optimizations based on unfolding.

5.2 Local Control in Abstract Interpretation

Local control in partial evaluation determines how each atom in $\mathbf{A}$ should be unfolded. However, in traditional frameworks for abstract interpretation we usually have a choice for abstract domain and widening operators, but no choice for local control is offered. This is because by default, in abstract interpretation each or-node is related by just one (abstract) unfolding step to its children. This corresponds to a trivial local control (unfolding rule) in partial evaluation.

Several possibilities exist in order to overcome the simplicity of the local control performed by abstract interpretation:

1. According to many authors, [Gal93, LM96] global control is much harder than local control. Thus, subsequent unfolding of the specialized program generated by Algorithm 1 can be done using traditional unfolding rules to eliminate determinate calls or some non-recursive calls, for example. The and–or analysis graph may be of much help in order to detect such cases.

2. Use abstract domains which allow propagating enough information about the success of an or–node so as to perform useful specialization on other or-nodes (for example by allowing sets of abstract substitutions). The advantage of this method is that no modification of the abstract interpretation framework is required. Also, as we will see in Example 5.1, it may allow specializations which are not possible by the methods proposed below.[3]

[3] Unless multivariance on successes is performed by the analysis framework.

3. Another possibility is a simple modification to the algorithm for abstract interpretation in order to accommodate an unfolding rule. In fact, unfolding can be formalized as a transformation in an and–or graph. In this approach, if the unfolding rule decides that an or-node should not be unfolded, then it is treated as in the usual case. If the rule decides that the atom should be further unfolded, the atom would be analyzed but the corresponding or–node would not be added to the and–or graph. Then, some amount of transformation which is equivalent to the unfolding step should be performed in the analysis graph, and analysis would continue with the usual algorithm. This approach would allow introducing the full power of partial evaluation into our framework by a simple modification of the analysis algorithm. The drawback is the need for the unfolding rule.

4. The last possibility is related to the first alternative in that analysis is performed first with a trivial unfolding rule and once analysis has finished, further unfolding may be performed if desired. However, rather than performing unfolding without modifying the analysis graph as in the first approach, whenever an additional unfolding step is performed, the analysis graph is modified accordingly, using the same graph transformation rules mentioned in the previous approach. However, the difference with the previous approach is that there, unfolding is completely integrated in abstract interpretation and the local control decisions are taken when performing analysis. The advantages over the previous approach are that there is no need to modify the analysis algorithm and that unfolding is performed once the whole analysis graph has been computed. The benefits of the availability of such better information for local control still have to be explored. The disadvantage is that in order to achieve as accurate information as in the previous approach it may be required to perform reanalysis in order to propagate the improved information introduced due to the additional unfolding steps, with the associated computational cost. This cost could remain reasonable by the use of incremental analysis techniques such as those presented in [HPMS95].

Example 5.1 Consider the following program and the goal ←`r(X)`

```
r(X) :- q(X),p(X).
q(a).
q(f(X)) :- q(X).
p(a).
p(f(X)) :- p(X).
p(g(X)) :- p(X).
```

The third clause for `p` can be eliminated in the specialized program for ←`r(X)`, provided that the call substitution for `p(X)` contains the information that `X=a` or `X=f(Y)`. The abstract domain has to be precise enough to capture, in this case, the set of principal functors of the answers.

Note that no partial evaluation algorithm based on unfolding will be able to eliminate the third clause for `p`, since an atom of form `p(X)` will be produced, no matter what local and global control is used[4]. Thus, simulating unfolding in abstract interpretation (such as methods 1, 3, and 4 above do) will not achieve this specialization either. An approach such as 2 is required. □

5.3 Abstract Domains and Widenings for Partial Evaluation

Once we have presented the relation between abstract domains and widening with global control in partial evaluation, we will discuss desired features for performing partial evaluation. Ideally, we would like that

- The domain can simulate the effect of unfolding, which is the means by which bindings are propagated in partial evaluation. Our abstract domain has to be capable of tracking such bindings. This suggests that domains based on term structure are required.

[4]Conjunctive partial deduction [LSdW96] can solve this problem in a completely different way.

- In addition, the domain needs to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation in order to achieve the approach 2 for local control. A term domain whose least upper bound is based on the *msg*, for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

Two classes of domain which have the above desirable features are:

- The domain of type-graphs [BJ92], [GdW94], [HCC94]. Its drawback is that inter-argument dependencies are lost.
- The domain of sets of depth-k substitutions with set union as the least upper bound operator. However uniform depth bounds are usually either too imprecise (if k is too small) or generate much redundancy if larger values of k are chosen.

One way to eliminate the depth-bound k in the abstract domain it to depend on a suitable widening operator which will guarantee that the set of or–nodes remains finite. Many techniques have been developed for global control of partial evaluation. Such techniques make use of data structures which are very related to the and–or graph such as *characteristic trees* [GB91], [Leu95] (related to *neighbourhoods* [Tur88]), *trace-terms* [GL96], and *global trees* [MG95], and combinations of them [LM96]. Thus, it seems possible to adapt these techniques to the case of abstract interpretation and formalize them as widening operators.

6 Future Work

It remains as future work to experiment with the techniques presented in this paper. We plan to do so in the context of the PLAI system. Different abstract domains and widening operators for global control should be implemented and experimented with. Efficiency of the approach as well as quality of the specialized programs should be compared to that of existing partial evaluators.

Acknowledgments

Part of this work was performed while Germán Puebla was visiting the University of Bristol supported the Human Capital and Mobility Network *Logic Program Synthesis and Transformation*. The authors would like to thank the anonymous referees for useful comments which will help to improve future versions of this paper.

References

[BJ92] M. Bruynooghe and G. Janssens. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[CK93] C. Consel and S.C. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[CV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*. Number 1110 in LNCS. Springer, February 1996. Dagstuhl Seminar.

[Gal92] J.P. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.

[Gal93] 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.

[GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[GCS88] J. Gallagher, M. Codish, and E.Y. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[GdW94] J. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GL96] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag Lecture Notes in Computer Science, 1996.

[HCC94] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

[HPMS95] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall, New York, 1993.

[LD97] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accepted for Publication in New Generation Computing.

[Leu95] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1995.

[Leu97] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.

[LM95] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. Technical Report CW 220, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1995.

[LM96] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[LS96] Michael Leuschel and De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996.

[LSdW96] M. Leuschel, D. De Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[MG95] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing fl exible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

[Neu90] G. Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 205–217, Leuven, Belgium, 1990. K. U. Leuven.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[PH97] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *VI International Workshop on Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[Sah93] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[Tur88] V. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[VDCM93] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.