An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework

Germán Puebla and *Manuel Hermenegildo*
Department of Computer Science
Technical University of Madrid (UPM)
{german,herme}@fi.upm.es

John P. Gallagher
Department of Computer Science
University of Bristol
john@cs.bris.ac.uk

Abstract

Information generated by abstract interpreters has long been used to perform program specialization. Additionally, if the abstract interpreter generates a multivariant analysis, it is also possible to perform multiple specialization. Information about values of variables is propagated by simulating program execution and performing fixpoint computations for recursive calls. In contrast, traditional partial evaluators (mainly) use unfolding for both propagating values of variables and transforming the program. It is known that abstract interpretation is a better technique for propagating success values than unfolding. However, the program transformations induced by unfolding may lead to important optimizations which are not directly achievable in the existing frameworks for multiple specialization based on abstract interpretation. The aim of this work is to devise a specialization framework which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation, and which can be implemented via small modifications to existing generic abstract interpreters. With this aim, we will relate top-down abstract interpretation with traditional concepts in partial evaluation and sketch how the sophisticated techniques developed for controlling partial evaluation can be adapted to the proposed specialization framework. We conclude that there can be both practical and conceptual advantages in the proposed integration of partial evaluation and abstract interpretation.

Keywords: Logic Programming, Abstract Interpretation, Partial Evaluation, Program Specialization.

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 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 algorithms for program specialization, 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, PH97, Jon97, PGH97, Leu98]. However, the role of analysis is so fundamental that it is natural to consider whether partial evaluation could be achieved directly by a generic, top-down 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 such a generic, top-down abstract interpreter. We then explore two main questions which arise. First, how much specialization can be performed by an abstract interpreter, compared to on-line partial evaluation? Second, how do the traditional problems of local and global control appear when placed in the setting of generic abstract interpretation? We conclude that there seem to be practical and conceptual advantages in using an abstract interpreter to perform program specialization.

2 Abstract Interpretation

Abstract interpretation [CC77] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow D$.

We recall some classical definitions in logic programming. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. We often use $\bar{t}$ to denote a tuple of terms. A *clause* is of the form H:-$B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. A *definite logic program*, or *program*, is a finite sequence of clauses.

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 in the abstract domain D_α. Such an abstract interpretation computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$, p_i is an atom and λ_i^c and λ_i^c are, respectively, the abstract call and success substitutions. Correctness of abstract interpretation guarantees:

- The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$.

- The abstract call substitutions cover all the concrete calls which appear during execution. $\forall c$ 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. $c = p_j\theta'$ and $\theta' \in \gamma(\lambda_j^c)$. This property is related to the closedness condition [LS91] required in partial deduction.

As usual 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. Note that 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. An analysis is said to be *multivariant on successes* if more than one triple $\langle p, \lambda^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^s \neq \lambda_j^s$ for some i, j may be computed for the same predicate p and call substitution λ^c. 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. Many implementations of abstract interpreters are multivariant on calls. However, most of them (such as PLAI [MH89, MH90, MH92]) are not multivariant on successes, mainly for efficiency reasons. As a result, and as we are interested in reusing existing abstract interpreters for performing partial evaluation, we will limit in principle our discussion to analyses which are multivariant on calls but not on successes. Note that this is not a strong restriction for our purposes as traditional partial evaluation is not multivariant on successes either. Also, code generation from an analysis which is multivariant on successes is not straightforward. However, multivariant successes can in fact be captured by certain abstract domains even if the analysis is not multivariant on successes, as will be discussed in Section 5. Note that for analyses not multivariant on successes when $\langle p, \lambda^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda^c, \lambda_n^s\rangle$ with $n > 1$ have been computed for the same predicate p and call substitution λ^c, the different substitutions $\{\lambda_1^s, \ldots, \lambda_n^s\}$ have to be summarized in a more general one (possibly losing accuracy) λ^s before propagating this success information. This is done by means of the *least upper bound* (lub) operator.[2]

[1] Extending the framework to sets of predicate symbols is trivial.
[2] D_α is a poset.

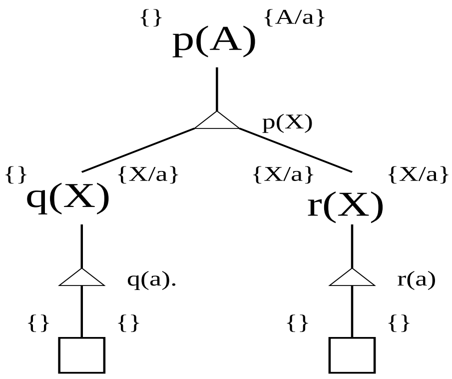

Figure 1: And–or analysis graph

In our case, in order to compute $Analysis(P, p, \lambda, D_\alpha)$, an and–or graph $AO(P, p, \lambda, D_\alpha)$ is constructed which encodes dependencies among the different triples. Such and–or graph can be viewed as a finite representation of the (possibly infinite) set of and–or trees explored by the (possibly infinite) concrete execution [Bru91]. 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]. If it is clear in the context, we will often write AO instead of $AO(P, p, \lambda, D_\alpha)$ for short.

Example 2.1 Consider the simple example program below taken from [Leu97].

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

We take as initial (exported predicate) the goal $p(A)$ with A unrestricted using the concrete domain as abstract domain. In this case, $Analysis(P, p(A), \{\}, D) = \{\langle p(A), \{\}, \{A/a\}\rangle, \langle q(X), \{\}, \{X/a\}\rangle, \langle r(X), \{X/a\}, \{X/a\}\rangle\}$ and Figure 1 depicts a possible and–or analysis graph. □

Due to space limitations, and given that it is now well understood, we do not describe in detail here how to build analysis and–or graphs. More details can be found in [Bru91, MH90, MH92, HPMS95, PH96]. 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 $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$. As before, λ_i^c and λ_i^s are, respectively, a pair of abstract call and success substitutions for the atom p_i. For clarity, in the figures the atom p_i is superscripted with λ^c to the left and λ^s to the right of p_i 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 $\langle Id, H\rangle$ where Id is a unique identifier for the node and H is the head of the clause the node refers to. In the figures, they are represented as triangles and H is depicted to the right of the triangles. Note that the substitutions (atoms) labeling and–nodes are concrete whereas the substitutions labeling or–nodes are abstract. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs

Algorithm 2.2 [Code Generation] Given $Analysis(P, p, \lambda, D_\alpha)$ and $AO(P, p, \lambda, D_\alpha)$ generated by analysis for a program P and an atomic goal $\leftarrow p$ with abstract substitution $\lambda \in D_\alpha$, and a partial concretization function $part\_conc$ do:

- For each tuple $N = \langle a(\bar{t}), \lambda^c, \lambda^s\rangle \in Analysis(P, p, \lambda, D_\alpha)$ generate a distinct predicate with name $pred_N = \mathsf{name}(\langle a(\bar{t}), \lambda^c, \lambda^s\rangle)$.
- Each predicate $pred_N$ is defined by
  - $pred_N(\bar{t})$ `:-` $fail$ — if $\lambda^s = \bot$
  - $(pred_N(\bar{t}_1)$ `:-` $b'_1)\theta_1 :: \ldots :: (pred_N(\bar{t}_n)$ `:-` $b'_n)\theta_n$ — otherwise
    where $expansion(N, AO) = O_N$ and
    $children(O_N, AO) = \langle Id_1, p_1(\bar{t}_1)\rangle :: \ldots :: \langle Id_i, p_i(\bar{t}_i)\rangle :: \ldots :: \langle Id_n, p(\bar{t}_n)\rangle$
- Each body b'_i is defined as
  - $b'_i = fail$ — if $\lambda^s_{ik_i} = \bot$
  - $b'_i = (pred_{i1}(\bar{t}_{i1}), \ldots, pred_{ik_i}(\bar{t}_{ik_i}))$ — otherwise
    where $pred_{ij} = \mathsf{name}(\langle a_{ij}(\bar{t}_{ij}), \lambda^c_{ij}, \lambda^s_{ij}\rangle)$, and
    $children(\langle Id_i, p(\bar{t}_i)\rangle, AO) = \langle a_{i1}(\bar{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1}\rangle :: \ldots :: \langle a_{ik_i}(\bar{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i}\rangle$.
- Each substitution θ_i is defined as
  - $\theta_i = \epsilon$ — if $b'_i = fail$
  - $\theta_i = mgu(std(part\_conc(\lambda^s_{i1})), \ldots, std(part\_conc(\lambda^s_{ik_i})))$ — otherwise

Figure 2: Algorithm for Code Generation

to and–nodes which represent the clauses with which the atom (possibly) unifies. And–nodes have arcs to or–nodes which represent the atoms in the body of the clause. Note that several instances of the same clause may exist in the analysis graph of a program. In order to avoid conflicts with variable names, clauses are standardized apart before adding to the analysis graph the nodes which correspond to such clause. This way, only nodes which belong to the same clause may share variables. As the head of the clause (after the standardizing renaming transformation) is stored in the and–node, we can always reconstruct (a variant of) the original clause when generating code from an and–or graph (see Section 3 below).

Intuitively, analysis algorithms are just graph traversal algorithms which given P, p, λ, and D_α build $AO(P, p, \lambda, D_\alpha)$ by adding the required nodes and computing success substitutions until a global fixpoint is reached. For a given P, p, λ, and D_α there may be many different analysis graphs. However, there is a unique *least analysis graph* which gives the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations. Each time the analysis algorithm creates a new or–node for p and λ^c and before computing the corresponding λ^s, it checks whether $Analysis(P, p, \lambda, D_\alpha)$ already contains a tuple for (a variant of) p and λ^c. If that is the case, the or–node is not expanded and the already computed λ^s stored in $Analysis(P, p, \lambda, D_\alpha)$ is used for that or–node. This is done both for efficiency and for avoiding infinite loops when analyzing recursive predicates. As a result, several instances of the same or–node may appear in AO, but only one of them is expanded. We denote by $expansion(N)$ the instance of the or–node N which is expanded. If there is no tuple for p and λ^c in $Analysis(P, p, \lambda, D_\alpha)$, the or–node is expanded, λ^s computed, and $\langle p, \lambda^c, \lambda^s\rangle$ added to $Analysis(P, p, \lambda, D_\alpha)$. Note that the success substitutions λ^s stored in $Analysis(P, p, \lambda, D_\alpha)$ are tentative and may be updated during analysis. Only when a global fixpoint is reached the success substitutions are safe approximations of the concrete success substitutions.

3 Code Generation from an And–Or Graph

The information in $Analysis(P, p, \lambda, D_\alpha)$ has long been used for program optimization. Multiple specialization is a program transformation technique which allows generating several versions $p_1, \ldots, p_n$ $n \geq 1$ for a predicate p in P. Then, we have to decide which of $p_1, \ldots, p_n$ is appropriate for each call to p. One possibility is to use run-time tests to decide which version to use. If analysis is multivariant on calls but not on successes, another possibility, as done in [Win92, PH95], is to generate code from $AO(P, p, \lambda, D_\alpha)$ instead of $Analysis(P, p, \lambda, D_\alpha)$. The arcs in $AO(P, p, \lambda, D_\alpha)$ allow determining which p_i to use at each call. Then, each version of a predicate receives a unique name and calls are renamed appropriately.

After introducing some notation, an algorithm which generates a logic program from an analysis and–or graph is presented in Figure 2 (Algorithm 2.2). Given a non-root node N, we denote by $parent(N, AO)$ the node $M \in AO$ such that there is an arc from M to N in AO, and $children(N, AO)$ 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 $children(N, AO)$ may be applied both to or– and and–nodes. We assume the existence of an injective function name which given $Analysis(P, p, \lambda, D_\alpha)$ returns a unique predicate name for each tuple and $\mathsf{name}(\langle q(\bar{t}), \lambda^c, \lambda^s\rangle) = q$ iff $q(\bar{t}) = p$ (the exported predicate) and $\lambda^c = \lambda$ (the restriction on initial goals), to ensure that top-level – exported

– predicate names are preserved.

Definition 3.1 [partial concretization] A function $part\_conc : D_\alpha \rightarrow D$ is a *partial concretization* iff $\forall \lambda \in D_\alpha \; \forall \theta' \in \gamma(\lambda) \; \exists \theta''$ s.t. $\theta' = part\_conc(\lambda)\theta''$.
$part\_conc(\lambda)$ 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 $part\_conc(\lambda) = \epsilon$ where ϵ is the empty substitution, is a trivially correct partial concretization of any λ.

Definition 3.2 [specialization] Let P be a definite program. Let $AO(P, p, \lambda, D_\alpha)$ be an and–or graph. We say that a program P' is a *specialization* of P w.r.t. $AO(P, p, \lambda, D_\alpha)$ and *part_conc* and we denote it $P' = spec(AO(P, p, \lambda, D_\alpha), part\_conc)$ iff P' can be obtained by applying Algorithm 2.2 to $AO(P, p, \lambda, D_\alpha)$ using *part_conc*.

Basically, Algorithm 2.2 for code generation creates a different version for each different (abstract) call substitution λ^c to each predicate p_i 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.[3] The interest in performing the proposed multiple specialization is that the new program may be subject to further optimizations, such as elimination of redundant type/mode checks, which are allowed in the multiply specialized program because now each version it to be used for a more restricted set of input values than in the original program. Additionally, in Algorithm 2.2 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 λ^s for $\langle p, \lambda^c, \lambda^s \rangle$ is not $\perp$, individual clauses for p whose success substitution is $\perp$ (useless clauses) for the considered λ^c are removed from the final program.

By *mgu* we denote, as usual, the *most general unifier* of substitutions. *std* represents the result of standardizing apart the results of *part_conc* in order to avoid undesired variable name clashes. Note that in Algorithm 2.2 atoms are specialized w.r.t. answers rather than calls, as is the case in traditional partial evaluation. This will in general provide further specialized (and optimized) programs as in general the success substitution (which describes answers) computed by abstract interpretation is more informative (restricted) than the call substitution. However, 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 2.2 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, as the number of unfolding steps must be finite. Additionally, as mentioned above, dead code, i.e., clauses not used to solve the considered goal, are removed.

Note that Algorithm 2.2 is an improvement over the code-generation phase of [PH95, PH97] in that it allows applying non-trivial partial concretizations of the abstract (success) substitutions. The program obtained by Algorithm 2.2 can then be further optimized by applying the notion of *abstract executability* as presented in [PH97], which reduces an atom w.r.t. an abstract substitution.

[3] The program obtained in this way is $program_0$ in the notation of [PH95].

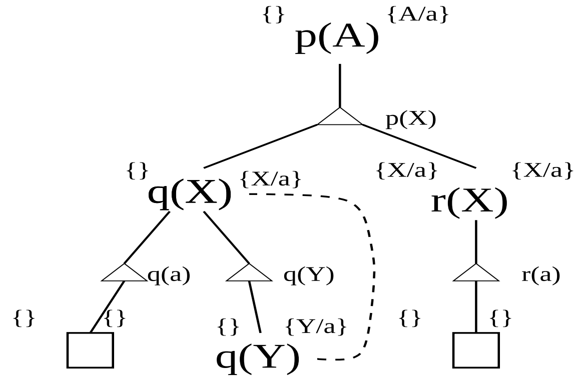

Figure 3: And–or analysis graph for a recursive program

Theorem 3.3 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 2.2. 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 of Example 2.1. The program obtained by applying Algorithm 2.2 to the and–or graph in Figure 1 is:

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

Note that Algorithm 2.2 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)`. □

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 3. The dotted arc indicates that the corresponding or–nodes have equivalent abstract call substitution. However, the set of tuples in $Analysis(P', p(A), \{\}, D)$ for the current program P' is exactly the same as in Example 2.1, in spite of the more involved and–or graph in this example. The program generated for this graph by Algorithm 2.2 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 observation already provides motivation for studying the integration of 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. 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 likely, and using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler.

5 Partial Evaluation using And–Or Graphs

We have established so far that for a given abstract interpretation of a program in a system such as PLAI (even interpretations over very simple domains such as modes) we can get some corresponding specialized source program with possibly multiple versions by applying Algorithm 2.2. 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 strongly 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 treated in a very general and flexible way by the general principles, methods, and formal results 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 program specialization greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which (specialized) code 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. 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 for ensuring termination when the abstract domain contains ascending chains which are infinite – as is the case for the concrete domain).

Note that the specialization framework we propose is very general. Depending on the kind of optimizations we are interested in performing, different domains (and widening operators) should be used and thus different $\mathbf{A}$ sets would 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 $p(ground)$ since, 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 to avoid generating 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.

Unfolding is a well known program transformation technique in which an atom in the body of a clause, which can be seen as a call to a procedure, is replaced by the code of such procedure. We now introduce the notion of *node-unfolding* which is a graph transformation technique which given an and–or graph AO and an or–node N in AO builds a new and–or graph AO'. Such graph transformation mimics the effect of unfolding an atom in a program.

Definition 5.1 [clause-unfolding]

Let $A = \langle Id, H \rangle$ be an and–or node in $AO(P, p, \lambda, D_\alpha)$ s.t. $children(A, AO) = L_1 :: \ldots :: N :: \ldots L_m$ with $m \geq 1$ and $N = \langle a, \lambda^c, \lambda^s \rangle$. Let also $C = H_C$ `:-` $B_1, \ldots, B_n$ be a clause in program P whose head H_C unifies with atom a.

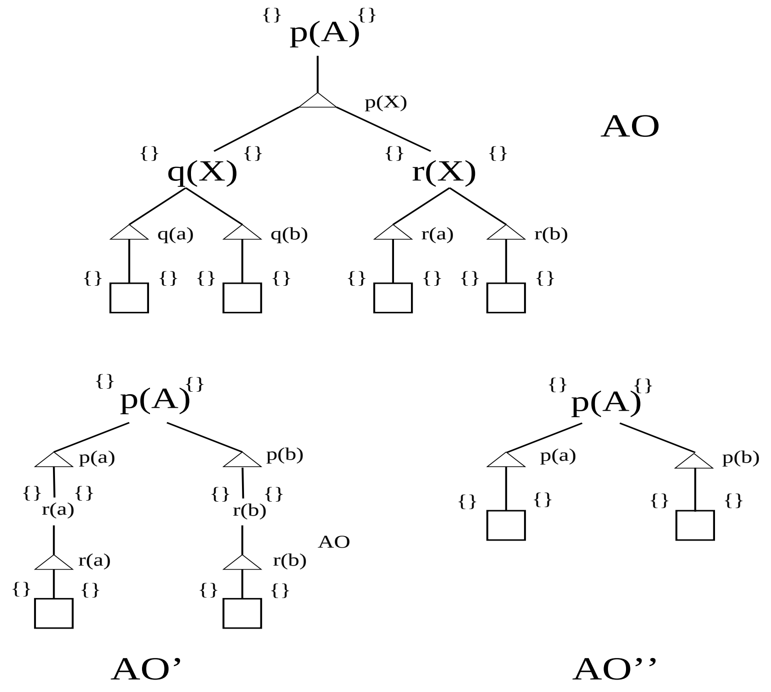

Figure 4: Example Node Unfoldings

The *clause-unfolding* of A and N w.r.t. C, denoted $cl\_unf(A, N, C)$, is the (partial) and–or graph AO' with root $A' = \langle Id, H\theta\rangle$ such that θ is a mgu of a and H_C and $children(A', AO') = L'_1 :: \ldots :: N_1 :: \ldots N_n :: \ldots L'_m$.

Each or-node N_j is of the form $\langle B_j, \lambda^c_j, \lambda^s_j\rangle$, where λ^c_j and λ^s_j have to be computed by the analysis algorithm as usual. Provided that $L_i = \langle p_i, \lambda^c_i, \lambda^s_i\rangle$ then, $L'_i = \langle p_i\theta, Aadd(\theta, \lambda^c_i), Aadd(\theta, \lambda^s_i)\rangle$ where $Aadd(\theta, \lambda)$ updates the abstract substitution λ by conjoining it with the concrete substitution θ (see for example [HPMS95]). A discussion on the effects of performing such conjoining (accuracy vs. efficiency) can be found in Section 5.3.

As usual, *mgu* denotes a most general unifier, in this case of two atoms. Clause-unfolding mimics the effect of an SLD resolution step.

Definition 5.2 [node-unfolding]
Let $N = \langle a, \lambda^c, \lambda^s\rangle$ be a non-empty or–node in AO. Let $C_1 :: \ldots :: C_n$ $n \geq 1$ be the sequence of standardized apart clauses in program P s.t. a unifies with the head of C_i. Let $parent(N, AO) = A$, and let $parent(A, AO) = GP$ with $children(GP, AO) = P_1 :: \ldots :: A :: \ldots P_i$, $i \geq 1$.

The *node-unfolding* of AO w.r.t. N, denoted *node-unfolding*(AO, N) is the and–or graph AO' obtained from AO by making $children(GP, AO')$ be $P_1 :: \ldots :: cl\_unf(A, N, C_1) :: \ldots :: cl\_unf(A, N, C_n) :: \ldots P_i$ and eliminating nodes A and N.

Node unfolding is achieved by performing clause-unfolding with all clauses in the program whose head unifies with the unfolded atom.

Theorem 5.3 [node-unfolding] Let $AO(P, p, \lambda, D_\alpha)$ be a (partially computed) and–or graph. Let *part_conc* be a partial concretization function, let $P_{AO} = spec(AO(P, p, \lambda, D_\alpha), part\_conc)$. Let N be an or–node in AO, let $AO' =$ *node-unfolding*(AO, N), and let $P' = spec(AO', part\_conc)$. 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_{AO} with computed answer θ_s.

ii) if $p\theta_c$ finitely fails in P_{AO} then $p\theta_c$ finitely fails in P'.

Theorem 5.3 guarantees correctness of node-unfolfing as it states that performing node-unfolding on an and–or graph preserves computed answers and finite failures.

Example 5.4 Reconsider the program of Example 2.1 in which an additional clause `q(b).` has been added to predicate q. The new analysis graph generated without performing any node-unfolding is shown in Figure 4 as AO, using the concrete domain as abstract domain and the *most specific generalization* (msg) as lub operator for summarizing different success substitutions into one. As discussed in Section 5.5 below, the msg is a rather crude lub operator. However, we use it for the sake of clarity of the example. AO' is an analysis graph for the same program but this time the or–node $\langle q(X), \{\}, \{\}\rangle$ has been unfolded. Finally, graph AO'' in the figure is the result of applying node-unfolding twice to AO', once w.r.t. $\langle p(a), \{\}, \{\}\rangle$ and another one w.r.t $\langle p(b), \{\}, \{\}\rangle$. The code generated by $spec(AO'', part\_conc)$ (for any *part_conc*) is the program:

```
p(a).
p(b).
```

5.3 Strategies for Local Control

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, one possibility is to obtain AO using the traditional analysis algorithm. Subsequent unfolding $spec(AO, part\_conc)$ can be done using traditional unfolding rules in order to eliminate determinate calls or some non-recursive calls, for example. The and–or analysis graph AO may be of much help in order to detect such cases.

2. A second alternative it to use abstract domains for analysis which allow propagating enough information about the success of an or–node so as to perform useful specialization on other or-nodes. This requires that the *lub* operator not lose "much" information, 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 5.5, it may allow specializations which are not possible by the methods proposed below (nor by traditional partial evaluation).

3. Another possibility is a simple modification to the algorithm for abstract interpretation in order to accommodate a node-unfolding rule. In this approach, if the node-unfolding rule decides that an or–node N in a graph AO should not be unfolded, then N is treated as in the traditional abstract interpretation algorithm. If the rule decides that N should be unfolded, $AO' = \textit{node-unfolding}(AO, N)$ is computed and analysis continues for the new graph AO'. Note that in order to unfold N it is not required to know its success substitution. Thus, the graph transformation associated to an unfolding is merely structural and can be performed before or after computing the and–or graph below N. If decisions on unfolding are taken before computing the nodes below N, the unfolding rule corresponds to those used in partial evaluation: only the history of nodes higher in the top-down algorithm is available for deciding whether to unfold or not. However, we can delay this decision until the graph below N has been computed. This allows making better decisions as also the specialization history of atoms lower in the hierarchy is known.

   In the latter case, if N is not the leftmost atom in its clause and the abstract domain is downwards closed, there is a choice of whether to apply the (sequence of) success substitution(s) for N to the sibling nodes $L_1 :: \ldots :: L_k$ to the left of N (i.e., to perform left propagation) and reanalyze such nodes with the better information or not. Both alternatives are correct. The second alternative may allow better analysis and specialization, but at a higher computational cost.

4. The last possibility we propose is to first compute AO with a trivial unfolding rule (i.e., using the traditional abstract interpretation algorithm). Once analysis has finished, further unfolding may be performed if desired, as done in the first alternative proposed. However, unlike the first approach, rather than performing unfolding externally to the analysis and without modifying the analysis graph, whenever an unfolding step is performed for a node N in AO, a new analysis graph $AO' = \textit{node-unfolding}(AO, N)$ is computed.

   This approach can be seen as an extreme case of delaying node-unfolding, not only until the graph below N has been computed (as mentioned in the third approach) but rather until a global fixpoint has been reached for the whole analysis graph. The difference with the third approach is that there, unfolding is completely integrated in abstract interpretation and the local control decisions are taken when performing analysis and as mentioned before, the only issue is whether to perform left propagation of bindings or not. The advantage over the previous approach is 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 possible it may be required to perform reanalysis in order to propagate the improved information introduced due to the additional unfolding steps, i.e., using $\langle p_i\theta, Aadd(\theta, \lambda_i^c), Aadd(\theta, \lambda_i^s)\rangle$ instead of $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$ for the nodes to the right of the unfolded node, with the associated computational cost. This cost could however be kept quite reasonable by the use of incremental analysis techniques such as those presented in [HPMS95, PH96].

Example 5.5 Consider the following program and the goal $\leftarrow$`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 $\leftarrow$`r(X)`, provided that the call substitution for `p(X)` (i.e., the success substitution for `q(X)`) contains the information that `X=a` or `X=f(Y)`. The abstract domain has to be precise enough to capture, in this case, at least 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. □

5.4 Abstract Domains and Widenings for Partial Evaluation

Once we have presented the relation between abstract domains and widening with global control in partial evaluation, in this section we 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.
- In addition, the domain needs to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation in order

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

to achieve the approach 2 for local control. A term domain whose least upper bound is based on the *msg* (most specific generalization), for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

Two examples of 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 analysis graph such as *characteristic trees* [GB91], [Leu95] (related to *neighborhoods* [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 Related Work

The integration of partial evaluation and abstract interpretation has been attempted before, both from a partial evaluation and abstract interpretation perspective. In [GCS88, Gal92] such an integration is attempted from the point of view of partial evaluation. However, the approach is only partially successful as the resulting specialization framework does not exploit the full power of abstract interpretation. Another attempt for functional rather than logic programs is presented in [CK93].

From an abstract interpretation perspective, the integration has also received considerable attention. In [GH91], abstract interpretation is used to perform multiple specialization in an ad-hoc way. Also in [GH91] the notion of *abstract executability* is presented (and later formalized in [PH97]) and applied to remove redundant builtin checks. The first complete framework for multiple specialization based on abstract interpretation is presented in [Win92]. The first implementation and experimental evaluation is presented in [PH95] together with a framework based on existing abstract interpreters. All these techniques, even though they allow important specializations often not achievable by partial evaluation, are not designed for performing unfolding, which is one of the basic optimization techniques used by partial evaluators.

On the other hand, the drawbacks of traditional partial evaluation techniques for propagating success information are identified in [LS96] and some of the possible advantages of a full integration of partial evaluation and abstract interpretation are presented in [Jon97].

To the best of our knowledge, the first framework which presents a full integration of abstract interpretation and partial evaluation is [PGH97], on which this paper is based. More recently, a different formulation of such an integration has been presented in [Leu98]. In this formulation a top-down specialization algorithm is presented which assumes the existence of an *abstract unfolding function*, possibly not based on concrete unfolding, which generalizes existing algorithms for partial evaluation. Rather strong conditions are assumed over the behaviour of the abstract unfolding function. Unfortunately, no method is given for computing interesting ones except by providing relatively simple examples based on concrete unfolding. Also, the top-down algorithm proposed suffers from the same problems as traditional partial evaluation: lack of success propagation. This problem is solved by integrating the top-down algorithm with a bottom-up abstract interpretation algorithm which approximates success patterns. Note that this alternative corresponds to alternative 3 for local control (see Section 5.2, or [PGH97]). The main difference is that in our approach a single (and already existing) top-down abstract interpretation algorithm augmented with an unfolding rule performs propagation of both the call and success patterns in an integrated fashion.

Another difference between the two approaches is that [Leu98] is capable of dealing with conjunctions and not only atoms. This allows conjunctive partial evaluation [LSdW96] but adds an additional level of complexity to the control of program specialization: in order to guarantee termination a mechanism needs to be provided for deciding when and how to split conjunctions into components. While a similar form of conjunctive partial evaluation could be easily included in our framework, there is another pragmatic reason for not doing so: in general, existing abstract interpreters (and partial evaluators) only analyze (specialize) atoms individually, and we aim at reusing as much of existing analyzers as possible, an objective which is a further difference between our work and [Leu98].

7 Conclusions

We have proposed an integration of traditional partial evaluation into standard, generic, top-down abstract interpretation frameworks. We now summarize the main conclusions which can be derived from this work. As seen in [PH95], a multiply specialized program can be associated to every abstract interpretation which is multivariant on calls. Abstract interpretation can be regarded as having the simple local control strategy of always performing one unfolding step. However, useful specialization can be achieved if the global control is powerful enough. The global control is closely related to the abstract domain which is used, since this determines the multivariance of the analysis. If the abstract domain is finite (as is often the case), global control may simply be performed by the abstraction function of the abstract domain. However, if the abstract domain is infinite (as is required for partial evaluation), global control has to be augmented with a widening operator in order to ensure termination. The strategies for global control used in partial evaluation, such as those based on characteristic trees [GB91, LD97], on global trees [MG95], and on combinations of both [LM96], are then applicable to abstract interpretation. We have discussed different alternatives for introducing more powerful local unfolding strategies in abstract interpretation, such as unfolding the specialized program derived from abstract interpretation, or incorporating unfolding into the analysis algorithm. In the latter case, it can be proved that the set of atoms $\mathbf{A}_{AI}$ computed by abstract interpretation is as good as or better approximation of the computation than the set of atoms $\mathbf{A}_{PE}$ computed by traditional on-line partial evaluation with the corresponding global and local control, due to the better success propagation of abstract interpretation.

Acknowledgments

This work was supported in part by ESPRIT project DiSCiPl and CICYT project ELLA. 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.

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. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The 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.

[Jon97] N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *Static Analysis Symposium*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.

[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.

[Leu98] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.

[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 flexible 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.

[PGH97] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[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.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[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.