# ABSTRACT MULTIPLE SPECIALIZATION AND ITS APPLICATION TO PROGRAM PARALLELIZATION

GERMAN PUEBLA AND MANUEL HERMENEGILDO

▷ Program specialization optimizes programs for known values of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation, specialization then being with respect to abstract values (substitutions), rather than concrete ones. We study the multiple specialization of logic programs based on abstract interpretation. This involves in principle, and based on information from global analysis, generating several versions of a program predicate for different uses of such predicate, optimizing these versions, and, finally, producing a new, "multiply specialized" program. While multiple specialization has received theoretical attention, little previous evidence exists on its practicality. In this paper we report on the incorporation of multiple specialization in a parallelizing compiler and quantify its effects. A novel approach to the design and implementation of the specialization system is proposed. The resulting implementation techniques result in identical specializations to those of the best previously proposed techniques but require little or no modification of some existing abstract interpreters. Our results show that, using the proposed techniques, the resulting "abstract multiple specialization" is indeed a relevant technique in practice. In particular, in the parallelizing compiler application, a good number of run-time tests are eliminated and invariants extracted automatically from loops, resulting generally in lower overheads and in several cases in increased speedups.

**Keywords:** Program Specialization, Abstract Interpretation, Partial Evaluation, Static Analysis, Parallelization, Loop Invariant Detection. ◁

---

This work is a combined, revised and enhanced version of [51] and [53] presented at *ACM Partial Evaluation and Semantic Based Program Manipulation'95* and *Logic Program Synthesis and Transformation'96* respectively.

*Address correspondence to* G. Puebla, Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660-Boadilla del Monte, Madrid, SPAIN. email: `german@fi.upm.es`

*THE JOURNAL OF LOGIC PROGRAMMING*



655 Avenue of the Americas, New York, NY 10010 xxxx-xxxx/xx/$7.00

## 1. INTRODUCTION

Compilers often use static knowledge regarding invariants in the execution state of a program in order to optimize this program for the identified particular cases [1]. Standard optimizations of this kind include dead-code elimination, constant propagation, conditional reduction, code hoisting, etc. A good number of source to source program optimizations can be seen as special cases of partial evaluation [14, 33, 16]. The main objective of partial evaluation is to automatically overcome losses in performance which are due to general purpose algorithms by specializing the program for known values of the inputs. In the case of logic programs partial evaluation takes the form of partial deduction [38, 36], which is closely related to other techniques used in functional languages such as "driving" [23]. Much work has been done in logic program partial deduction and specialization of logic programs (see, e.g., [20, 21, 29]).

### *1.1. Abstract Specialization*

It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation [15]. Abstract interpretation of logic programs and the related implementation techniques are well understood for several general types of analysis of Prolog [18, 3, 48, 17, 41, 10]. Specialization can then be performed with respect to abstract values, rather than concrete ones. Such abstract values are safe approximations in a "representation domain" of a set of concrete values. Standard safety results imply that the set of concrete values represented by an abstract value is a superset (dually, a subset) of the concrete values that may appear at a certain program point in all possible program executions. Thus, any optimization allowed in the superset will also be correct for all the run-time values. The possible optimizations include again dead-code elimination, (abstract) constant propagation, conditional reduction, code hoisting, etc., which can again be viewed as a special case of a form of "abstract partial evaluation." Consider, for example, the following general purpose addition predicate which can be used when at least any two of its arguments are bound to integers at call time:

```
plus(X,Y,Z):-
     integer(X),integer(Y),!,Z is X + Y.
plus(X,Y,Z):-
     integer(Y),integer(Z),!,X is Z - Y.
plus(X,Y,Z):-
     integer(X),integer(Z),!,Y is Z - X.
```

If, for example, for all calls to this predicate in the program it is known from global analysis that the first and second arguments are always integers, then the program can be specialized as follows:

```
plus(X,Y,Z):-
     Z is X + Y.
```

which would clearly be more efficient because no tests are executed. The optimization above is based on "abstractly executing" the tests, i.e. reducing predicate calls to `true`, `fail`, or a set of primitives (typically, unifications) based on the information available from abstract interpretation. The notion of *abstract executability* was first introduced informally in [22] and later formalized in [53], and is instrumental in the optimization process. For completeness, we summarize the formalization of abstract executability in Section 6.1.

The class of optimizations which can be performed using abstract executability can be made to cover also traditional lower-level optimizations, provided the lower-level code to be optimized is "reflected" at the source level. Consider the optimization of general builtin predicates into simpler versions which are specialized for particular cases. This can be done by providing a reflexive version of the builtin in which the tests that detect the different uses appear explicitly and are then available for abstract execution. For example, the Prolog builtin predicate `arg/3` typically checks whether the third argument is an (unaliased) variable. If this check is made explicitly in a source representation of the built-in it can be simplified using abstract execution in the same way as the `integer` checks in the example above. Similarly, at a lower level, the same technique can be used to improve the actual code being generated by the compiler [58].

### *1.2. Multiple Specialization*

It is also often the case that a procedure has different uses within a program, i.e. it is called from different places in the program with different (abstract) input values. In principle, optimizations are then allowable only if the optimization is applicable to all uses of the predicate. However, it is possible that in several different uses the input values allow different and incompatible optimizations and then none of them can take place. This can be overcome by means of "multiple program specialization" [29, 22, 3, 61] (the counterpart of polyvariant specialization [9]), where different versions of the predicate are generated for each use. Each version is then optimized for the particular subset of input values with which it is to be used. In contrast, a program specialization in which (at most) one implementation is generated for each predicate in the original program will be referred to as *monovariant*.

For example, in order to allow maximal optimization, different versions of the `plus/3` predicate should be generated for the following calls:

```
..., plus(X1,Y1,Z1), plus(X2,Y2,Z2), ...
```

if, for example, `X1`, and `Y1` are known to be bound to integers, but no information is available on `X2`, `Y2`, and `Z2`.

While the technique outlined above is very interesting in principle, many practical issues arise, some of which have been addressed in different ways in previous work [29, 22, 3, 61]. One is the method used for selection of the appropriate version for each call at run-time. This can be done quite simply by renaming calls and predicates. In the example above this would result in the following calls and the additional optimized version `plus1/3` of the `plus/3` predicate:

```
..., plus1(X1,Y1,Z1), plus(X2,Y2,Z2), ...

plus1(X,Y,Z):-
     Z is X + Y.
```

This approach has the potential problem that, in order to create a "path" from the call to an optimized version of a predicate, multiple versions for some intermediate predicates may have to also be generated even if no optimization is performed for them. Clearly, this results in an additional increase in code size. Jacobs et al. [29] propose instead the use of simple run-time tests to discern the different possible call modes and determine the appropriate version dynamically. This is attractive in that it avoids the "spurious" versions of the previous solution (and thus reduces code size. However, it is also dangerous as such run-time tests themselves imply a cost which may be in unfavorable cases higher than the gains obtained due to multiple specialization.

Another problem is that it is not straightforward to decide the optimum number of versions for each predicate. In general, the more versions generated, the more optimizations possible, but this can lead to an unnecessarily large increase in program size.

## *1.3. Main Contributions*

Multiple specialization has received considerable theoretical attention. In [61] Winsborough presents the first powerful framework for automatic implementation of multiple specialization of logic programs. This framework solves the two problems outlined above while provably producing a program with multiple versions of predicates in such a way that it allows the maximum optimizations possible while having the minimal number of versions for each predicate.

The body of work in the area and Winsborough's fundamental results, plus the fact that abstract interpretation is becoming a practical tool in logic program compilation [27, 59, 48, 55, 7], suggests that it may be worthwhile to study whether multiple specialization could be useful in practice. However, little evidence on the practicality of abstract interpretation driven multiple specialization in logic programs has been provided previous to our work [51, 53]. Improvements for a few small, hand-coded examples were reported in [39, 59]. More recently, an implementation of multiple specialization has also been reported in [34, 35], applied to CLP($\mathcal{R}$). Also recently, further evidence on the potential of multiple specialization for optimization of logic programs has been reported in [37]. There, some unifications which satisfy certain conditions are specialized, thus obtaining more efficient programs. However, the method is not based on abstract interpretation and does not seem directly applicable to other kinds of optimizations.

We report on the implementation of multiple specialization in a parallelizing compiler for Prolog which incorporates an abstract interpretation-based global analyzer. We present a performance analysis of multiple specialization in this system, in which a minimization of the number of versions is performed. We argue that our results show that multiple specialization is indeed practical and useful in the application, and also that such results shed some light on its possible practicality in other applications.

We also propose a novel technique for the practical implementation of multiple specialization. While the analysis framework used by Winsborough is interesting in itself, several generic analysis engines, such as PLAI [48, 45] and GAIA [10], which greatly facilitate construction of abstract interpretation analyzers, are available, well understood, and in comparatively wide use. We believe that it is of practical interest to specify a method for multiple specialization which can be incorporated in a compiler using a minimally modified existing generic analyzer. We propose a framework which achieves the same results as those of Winsborough's but with only a slight modification of a standard abstract interpreter. Our algorithm can be seen as an implementation technique for Winsborough's method in the context of standard analyzers.

### 1.4. Organization

The structure of the paper is as follows. Section 2 briefly recalls the main concepts in abstract interpretation. In Section 3 we propose a naïve implementation method for multiple specialization based on abstract interpretation. In Section 4 we present an algorithm for minimizing the number of versions. Then Section 5 presents the application where multiple specialization will be applied: automatic parallelization. Section 6 shows the design of the abstract specializer and an example of a specialized program. Section 7 presents the experimental results, which are then discussed in Section 8. Related work is discussed in Section 9. Finally, Section 10 concludes.

## 2. ABSTRACT INTERPRETATION

We start by introducing some notation. A *program* is a sequence of *clauses*. Clauses are of the form $H :- B_1, \ldots, B_n$, where $H$ is an atom, $n \geq 0$, and $\forall i = 0 \ldots n$ $B_i$ is a literal.[1] $H$ is referred to as the head and $B_1, \ldots, B_n$ as the body of the clause.

Abstract interpretation [15] is a useful technique for performing global analysis of a program in order to compute at compile-time characteristics of the run-time behaviour of the program. The interesting aspect of abstract interpretation vs. classical types of compile-time analyses is that it offers a well founded framework which can be instantiated to produce a rich variety of types of analysis with guaranteed correctness with respect to a particular semantics [15, 4, 31, 42, 47].

### 2.1. Abstract Domains

In abstract interpretation, execution of the program is simulated on an *abstract domain* ($D_\alpha$) which is simpler than the actual, *concrete domain* ($D$). Thus, *abstract substitutions* ($\lambda$) are used instead of actual substitutions ($\theta$). An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain.

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* $\gamma : D_\alpha \mapsto D$. The usual definition for partial order ($\sqsubseteq$) over abstract domains, is $\forall \lambda, \lambda' \in D_\alpha$ $\lambda \sqsubseteq \lambda'$ iff $\gamma(\lambda) \subseteq \gamma(\lambda')$. In addition, each primitive operation $u$ of the language (unification being a notable example) is abstracted to an operation $u'$ over the abstract domain. Soundness of the analysis requires that each concrete operation $u$ be related to

[1] Our implementation supports essentially all the builtins of ISO-Prolog [5]. However, for simplicity we avoid their discussion except in cases where it may be especially relevant. This includes for example programs which have if-then-else's in the body of clauses, such as those generated by automatic parallelization, as will be seen in Section 5.2. This construct poses no additional theoretical difficulties: the same effect (modulo perhaps some run-time overhead) can be achieved using conjunctions of literals and the cut.

its corresponding abstract operation $u'$ as follows: for every $x$ in the concrete computational domain, $u(x) \subseteq \gamma(u'(\alpha(x)))$.

### 2.2. Goal Dependent Abstract Interpretation

The goal of many of the standard analysis engines used in logic programming is, for a given abstract domain, to annotate the program with abstract information about the possible run-time environments (i.e., the values of variables), at each program point. Usual relevant program points are entry to the clause, the point between each two literals in a clause, and return from the clause. In particular, we will be interested in the *abstract call substitution* $\lambda$ for each literal $L$ which is the abstract substitution just before calling $L$.

Correctness of the analysis requires that annotations be valid for any call (program execution). If the analysis is *goal dependent* (a.k.a. goal oriented), then the abstract interpreter receives as input, in addition to the program, a set of *calling patterns* which are descriptions of the calling modes into the program. Information inferred by goal dependent analysis may be more accurate as it "only" has to be valid when executing calls described by the calling patterns. In its minimal form (least burden on the programmer) the calling patterns may simply be the names of the predicates which can appear in user queries. In order to increase the precision of the analysis, it is often possible to include a description of the set of abstract (or concrete) substitutions allowable for each predicate by means of *entry* declarations [5].

For simplicity, in the presentation only one calling pattern for analysis is given. A calling pattern for an abstract domain $D_\alpha$ consists of a predicate symbol $p$ together with a restriction of the run-time bindings of $p$ expressed as an abstract substitution $\lambda \in D_\alpha$. Extending the framework to sets of calling patterns is trivial. Goal dependent 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 $\theta_s$ 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 $\lambda$ 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 [38] usually required in partial evaluation.

### 2.3. Multivariant Analyses

In order to increase accuracy, analyzers are usually *multivariant*. An analysis is said to be multivariant on calls if more than one triple $\langle p, \lambda_1^c, \lambda_1^s \rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s \rangle$ $n > 1$ with $\lambda_i^c \neq \lambda_j^c$ for some $i, j$ may be computed for the same predicate $p$. 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 $j > 0$. Actual analyzers differ in the degree of multivariance supported [57] and in the way such multivariance is represented, but, in general, most analyzers generate all possible versions since this allows the most accurate analysis [3, 48, 45, 10]. In multivariant analysis, a single program point (in the original program) may be annotated with several abstract substitutions. Normally, the results of the analysis are simply "folded back" into the program: information which correspond to the same points is combined using the *least upper bound* (lub) operator.

```
go(A,B):-
     p(A,B,_), p(A,_,B).
p(X,Y,Z):-
     plus(X,Y,Z),
     write(Z), write(' is '),
     write(X), write(' + '), write(Y), nl.
```

**FIGURE 2.1.** Example Program

We will limit the discussion to analyses which are multivariant on calls but not on successes, such as the analysis algorithm in PLAI and in the framework of [61]. Note that if analysis is not multivariant on successes when several success substitutions $\{\lambda^s_{i_1}, \ldots, \lambda^s_{i_j}\}$ with $j > 1$ have been computed for the same predicate $p_i$ and call substitution $\lambda^c_i$, the different substitutions have to be summarized in a more general one (possibly losing accuracy) $\lambda^s_i$ before propagating this success information. This is done by means of the lub operator.

## *2.4. Analysis And–Or Graphs*

Traditional, goal dependent abstract interpreters for logic programs based on Bruynooghe's analysis framework [3], in order to compute $Analysis(P, p, \lambda, D_\alpha)$, construct an and–or graph which corresponds to (or approximates) the abstract semantics of the program. We will denote by $AO(P, p, \lambda, D_\alpha)$ the and–or graph computed by the analyzer for a program $P$ with calling pattern $p, \lambda$ using the domain $D_\alpha$. 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. Concrete and–or trees which are infinite can be represented finitely through a widening into a rational tree. Also, the use of abstract values instead of concrete ones allows representing infinitely many concrete execution trees with a single abstract analysis graph.

Finiteness of $AO(P, p, \lambda, D_\alpha)$ (and thus termination of analysis) is achieved by considering an abstract domain $D_\alpha$ with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [15].

We do not describe here how to build $AO(P, p, \lambda, D_\alpha)$. Details can be found in [3, 45, 48, 25]. The graph has two sorts of nodes: those which correspond to literals (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda^c_i, \lambda^s_i \rangle$ and the set of or–nodes in $AO(P, p, \lambda, D_\alpha) = Analysis(P, p, \lambda, D_\alpha)$. And–nodes are also triples $\langle H_j, \lambda^c_j, \lambda^s_j \rangle$ where $H_j$ is the head of the clause the node corresponds to. Or–nodes have arcs to and–nodes which correspond to the clauses with which the literal (possibly) unifies. An and–node for a clause $H$ :- $B_1, \ldots, B_n$ has $n$ arcs to or–nodes. Each one of such or–nodes corresponds to a literal in the body of the clause.

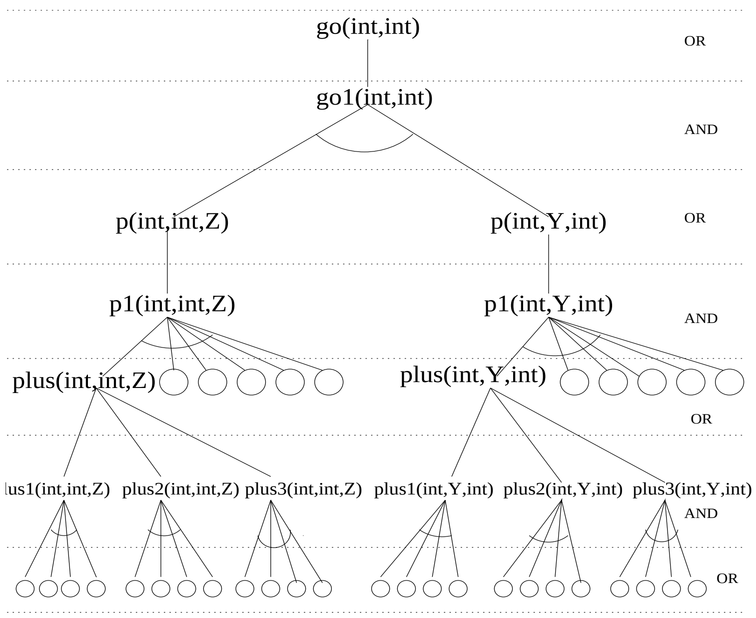


**FIGURE 2.2.** Example Analysis Graph

## *2.5. Example*

Consider the example program $P$ presented in Figure 2.1, where the predicate *plus/3* is defined as in Section 1.1 and *go/2* is known to be always called with both arguments bound to integers.

Consider also the abstract domain $D_\alpha$ consisting of the five elements $\{bottom$, $int$, $float$, $free$, $top\}$. These elements respectively correspond to the empty set of terms, the set of all integers, the set of floating point numbers, the set of all unbound variables, and the set of all terms. Figure 2.2 shows $AO(P, go/2, go(int, int), D_\alpha)$. For simplicity, success substitutions are not shown. For and–nodes, a number is added to the predicate name to distinguish the different clauses which define the predicate. Finally, circles are used to represent calls to builtin predicates. Clearly, as there are infinitely many integer values, such graph represents an infinite number of concrete graphs.

# 3. MULTIPLE SPECIALIZATION USING ABSTRACT INTERPRETATION

The traditional approach to analysis-based optimizing compilers is to first analyze the program and then use the information in $Analysis(P, p, \lambda, D_\alpha)$ to perform monovariant program optimization.

Let $\{\langle p_j, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_j, \lambda_n^c, \lambda_n^s\rangle\}$ $n \geq 0$ be the tuples in $Analysis(P, p, \lambda, D_\alpha)$ for predicate $p_j$.[2] The main idea that we will exploit is to generate a different

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

version of $p_j$ for each tuple $\langle p_j, \lambda_i^c, \lambda_i^s \rangle$. Then, each version can be specialized w.r.t. $\lambda_i^c$ regardless of the rest of the call substitutions $\lambda_j$ $\forall j \neq i$. Hopefully, this will lead to further oportunities for optimization in each particular version. Note that if analysis terminates the number of tuples in $Analysis(P, p, \lambda, D_\alpha)$ for each predicate must be finite, and thus the resulting program will be finite.

An important issue in this approach is to decide, given a predicate $p_j$ in $P$ for which $n$ versions are to be generated, which of the $n$ versions is appropriate for each call to $p_j$. As mentioned before, one possibility is to use run-time tests to decide which version to use. Another possibility, as in [61, 51] and which is the one we adopt, is to determine at compile-time the appropriate version to use at each call.

### *3.1. Analyses with Explicit Construction of the And-Or Graph*

As mentioned before, some formulations of goal dependent abstract interpretation for logic programs, such as the original one in Bruynooghe's seminal work [3], are based on explicitly building an abstract version of the and–or tree which contains a different or–node for each different call substitution $\lambda_i^c$ to a predicate $p_j$ which has been detected during analysis [43, 30]. This has the advantage that, while not directly represented in $AO(P, p, \lambda, D_\alpha)$, it is quite straightforward to derive a fully multiply specialized program (i.e. with all possible versions) from such graph and the original program. The arcs in $AO(P, p, \lambda, D_\alpha)$ allow determining at compile-time which version to use at each call. Each call in each clause body in the multiply specialized program is replaced with a call to the unique predicate name corresponding to the successor or–node in the graph. We will refer to the program constructed as explained above as the *extended program*.

The correctness of this multiply specialized program is given by the correctness of the abstract interpretation procedure, as the extended program is obtained by simply materializing the (implicit) program with multiple versions from which the analysis has obtained its information.

### *3.2. Tabulation-based Analyses*

For efficiency reasons, most practical analyzers [18, 27, 48, 10, 40] do not explicitly build and store $AO(P, p, \lambda, D_\alpha)$. In most systems, some or all of the graph structure is lost, and the data available after analysis essentially corresponds to $Analysis(P, p, \lambda, D_\alpha)$. However, this suffices if only monovariant specialization is performed.

For concreteness, we consider here the case of PLAI [48, 45]. In the standard implementation of this analyzer only entries which correspond to or–nodes, i.e., $Analysis(P, p, \lambda, D_\alpha)$ are stored. And–nodes are also computed and used, but they are not stored. This information is not enough to determine at compile time the version to use at each call (program point in the extended program). However, it is easy to compute $Analysis\_ancestors(P, p, \lambda, D_\alpha)$ which encodes the arcs among the different nodes either by a simple modification to the PLAI algorithm or as a postprocessing phase after analysis. In $Analysis\_ancestors(P, p, \lambda, D_\alpha)$ the tuples $\langle p_i, \lambda_i^c, \lambda_i^s \rangle \in Analysis(P, p, \lambda, D_\alpha)$ are augmented with two more fields, resulting in $\langle id_i, p_i, \lambda_i^c, \lambda_i^s, A_i \rangle$. $id_i$ is a unique identifier for the tuple (version) and $A_i$ contains the *ancestor* information for such tuple, i.e., the (list of) program point(s) in the extended program where this version $id_i$ is used. A program point

| $id_i$ | $p_i$ | $\lambda_i^c$ | $\lambda_i^s$ | $A_i$ |
|---|---|---|---|---|
| 1 | go/2 | $go(int, int)$ | $go(int, int)$ | {(query,1)} |
| 2 | p/3 | $p(int, int, free)$ | $p(int, int, int)$ | {(go/2/1/1,1)} |
| 4 | p/3 | $p(int, free, int)$ | $p(int, int, int)$ | {(go/2/1/2,1)} |
| 3 | plus/3 | $plus(int, int, free)$ | $plus(int, int, int)$ | {(p/3/1/1,2)} |
| 5 | plus/3 | $plus(int, free, int)$ | $plus(int, int, int)$ | {(p/3/1/1,4)} |

**TABLE 3.1.** *Analysis_ancestors* for the Example Program

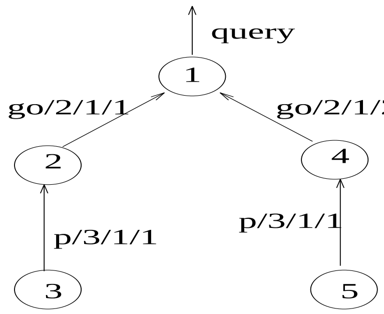


**FIGURE 3.1.** Ancestor Information for the Example Program

in the extended program is uniquely identified by a pair $(literal, id_j)$. It should be noted that the traditional fixpoint algorithm in PLAI had to be modified slightly so that this information is correctly stored, but this modification is straightforward. The newer fixpoint algorithm [52] recently integrated into PLAI computes $Analysis\_ancestors(P, p, \lambda, D_\alpha)$ rather than $Analysis(P, p, \lambda, D_\alpha)$. The ancestor information is useful for guiding iterations and for performing incremental analysis and thus, in that case, no modification of the fixpoint algorithm is needed at all.

In the case of [61], the abstract interpretation performed is based on minimal-function graph semantics rather than and–or graphs. In a postprocessing phase, using the analysis information, an automaton is constructed which contains as many states as versions in the extended program. For each literal in each version, there is a transition in the automaton to the version which must be executed. Such automaton is used at compile time to rename calls in clause bodies to the appropriate version. The solution we adopt is equivalent but the ancestors information contains reversed transition information (for implementation reasons).

*Example 3.1.* Table 3.1 shows $Analysis\_ancestors(P, go/2, go(int, int), D_\alpha)$ for the example program $P$ and abstract domain $D_\alpha$ in Section 2.5. A literal is identified using the following format: *Predicate/Arity/Clause/Literal*. For example, *go/2/1/2* stands for the second literal in the first clause of predicate *go/2*. If programs are as defined in Section 2, this format allows uniquely identifying a literal in a program.

Figure 3.1 represents the ancestor information graphically. For clarity, each tuple

```
go(A,B) :-
        p1(A,B,_), p2(A,_,B).

p1(X,Y,Z) :-
        plus1(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.
p2(X,Y,Z) :-
        plus2(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.

plus1(X,Y,Z) :-
        Z is X+Y.
plus2(X,Y,Z) :-
        Y is Z-X.
```

**FIGURE 3.2.** Extended Program for the Example Program

(or–node) is represented by its identifier. It is clear that the ancestor information can be interpreted as backward pointers in the analysis graph. The special literal `query` indicates the calling pattern for goal dependent analysis. PLAI admits any number of calling patterns. They are identified by the second number of the pair (query,id). Finally, the extended program is given in Figure 3.2.

## 4. MINIMIZING THE NUMBER OF VERSIONS

The number of versions in the extended program does not depend on the possible optimizations but rather on the number of versions generated during analysis. Even if no benefit is obtained, the extended program may have more than one version of each predicate. In this section we address the issue of finding a minimal program that allows the same set of optimizations as the extended program and which can be implemented without introducing run-time tests to select among different versions of a predicate.

After analysis and prior to the execution of the minimizing algorithm, we compute the optimizations that would be allowed in each version of the extended program. We assume the existence of a function *opt* which given a tuple $\langle id_i, p_i, \lambda_i^c, \lambda_i^s, A_i \rangle$ computes the set of optimizations allowed in such tuple (version). A simple but very inefficient way of implementing *opt* would be to materialize the extended program, let the optimizer run on this program, and collect the optimizations performed in each version. However, in many cases, such as in our specializer presented in Section 6, *opt* is computed without having to materialize the extended program. We will denote by $Analysis\_optimizations(P, p, \lambda, D_\alpha, opt)$ the set of tuples of the form $\langle id_i, p_i, A_i, S_i \rangle$ obtained by adding to each tuple in

$Analysis\_ancestors(P, p, \lambda, D_\alpha)$ the field $S_i = opt(\langle id_i, p_i, \lambda_i^c, \lambda_i^s, A_i\rangle)$ and removing the fields $\lambda_i^c$, and $\lambda_i^s$ which are not needed by the minimization algorithm. In an abuse of notation, we will write $Analysis\_optimizations(P, p, \lambda, D_\alpha, opt)$ simply as $Analysis\_optimizations$. Note that the minimizing algorithm is independent of the kind of optimizations being performed. In fact, *opt* is just a parameter of $Analysis\_optimizations(P, p, \lambda, D_\alpha, opt)$. The only requirement is that sets of optimizations $S_i$ be comparable for equality. As an example, in our implementation an optimization is a pair $(literal, value)$, where value is `true` or `fail` (or a list of unifications), generated via abstract executability (see Section 6.1). The algorithm receives as input $Analysis\_optimizations$. The output of the algorithm is a partition of $Analysis\_optimizations$ into equivalence classes. As many versions are generated for each predicate in the original program as equivalence classes exist for it. The optimizations will be materialized after the minimization phase.

### 4.1. Basic Definitions

The minimizing algorithm is not very complex. The main interest in the formalization we provide is that some of the definitions presented help in understanding the desirable properties that multiply specialized programs should have, such as being minimal, of maximal optimization, feasible, etc. At this point, we will not be concerned with termination (see Section 4.4).

*Definition 4.1.* [Or–record] An *or–record* is a tuple $o = \langle id, p, A, S\rangle \in Analysis\_optimizations$.

*Definition 4.2.* [Version] A set of or–records $v = \{\langle id_1, p_1, A_1, S_1\rangle, \ldots, \langle id_n, p_n, A_n, S_n\rangle\}$ $n > 0$ is a *version* if $\forall i, j = 1, \ldots, n \;\; p_i = p_j$.
I.e., a version is a set of or–records for the same predicate.

*Definition 4.3.* [Program] A set of versions $P = \{v_1, \ldots, v_n\}$ $n \geq 0$ is a *program* if $\forall o \in Analysis\_optimizations \;\; \exists!\, v \in P : o \in v$.
I.e., a program is a partition of the set of or–records for each predicate.

*Definition 4.4.* [Feasible Version] A version $v$ in a program $P$ is *feasible* if it does not use two different versions for the same literal, i.e. if $\forall o_i, o_j \in v$ :

$$\forall\, lit( \left.\begin{array}{l} (\exists v_k \in P\; \exists o_l = \langle id_l, p_l, A_l, S_l\rangle \in v_k | (lit, id_i) \in A_l) \wedge \\ (\exists v_m \in P\; \exists o_n = \langle id_n, p_n, A_n, S_n\rangle \in v_m | (lit, id_j) \in A_n)) \end{array}\right\} \Longrightarrow k = m$$

A program is feasible if all the versions in the program are feasible. Programs with versions that are not feasible cannot be implemented without run-time tests to decide the version to use. Infeasible programs use for the same literal *sometimes* a version and *sometimes* another. This *sometimes* must be determined at run-time.

*Definition 4.5.* [Equivalent Or–records] Two or–records $o_i = \langle id_i, p_i, A_i, S_i\rangle, o_j = \langle id_j, p_j, A_j, S_j\rangle$ are *equivalent*, denote by $o_i \equiv_v o_j$, if

$$p_i = p_j, S_i = S_j, \text{ and } \{o_i, o_j\} \text{ is a feasible version.}$$

*Definition 4.6.* [Minimal Program] A program $P$ is *minimal* if $\forall o_i, o_j \in$

$Analysis\_optimizations$

$$o_i \equiv_v o_j \Longrightarrow \exists v \in P \text{ such that } o_i, o_j \in v$$

*Definition 4.7.* [Program of Maximal Optimization] A program $P = \{v_1, \ldots, v_n\}$ is of *maximal optimization* if

$$\forall k = 1, \ldots, n \; \forall o_i = \langle id_i, p_i, A_i, S_i \rangle, o_j = \langle id_j, p_j, A_j, S_j \rangle \in v_k \;\; S_i = S_j$$

I.e., no two or–records with different optimizations are placed in the same version.

According to these definitions, monovariant specialization is feasible, and minimal, but in general, not of maximal optimization.

## *4.2. Phase 1: Reunion*

The aim of this phase is, given $Analysis\_optimizations$ (the extended program), to obtain a program which is of maximal optimization while remaining minimal.

*Definition 4.8.* [$Program_i$] $Program_i = \{v_1, \ldots, v_n\}$ is the program such that $\forall o_i = \langle id_i, p_i, A_i, S_i \rangle, o_j = \langle id_j, p_j, A_j, S_j \rangle \in Analysis\_optimizations$ :

$$\exists v_k \in Program_i \text{ s. t. } o_i, o_j \in v_k \Longleftrightarrow p_i = p_j \wedge s_i = s_j$$

$Program_i$ corresponds to the program in which the set of or–records for each predicate is partitioned into equivalence classes using the equality of sets of optimizations as equivalence relation.

*Theorem 4.1. $Program_i$ is of maximal optimization, and minimal.*

Unfortunately, $Program_i$ is not feasible in general. This is because two or–records that allow the same set of optimizations cannot be blindly collapsed since they may use different versions for the same literal.

## *4.3. Phase 2: Splitting*

The aim of this phase is to obtain a program which is feasible. As the program obtained in phase 1, it should also be minimal and of maximal optimization.

The concept of restriction is instrumental during phase 2. It is used to split versions that are not feasible. It allows expressing in a compact way the fact that several or–records must be in different versions. For example $\{\{1\}, \{2, 3\}, \{4\}\}$ can be interpreted as: or–record 1 must be in a different version than 2, 3, and 4. Also or–records 2 and 3 cannot be in the same version as 4 (2 and 3 can, however, be in the same version).

*Definition 4.9.* [Restriction from a Predicate to a Literal] Let $\mathcal{V}_{Pred} = \{v_1, v_2, \cdots, v_i, \cdots, v_n\}$ be the set versions for the predicate *Pred* in a program $P$, and let *lit* be a literal of the program. The restriction from *Pred* to *lit* is

$$\mathcal{R}_{lit,Pred} = \{r_1, r_2, \cdots, r_i, \cdots, r_n\}$$

where $r_i$ is $\{id|\, \exists o = \langle id, p, A, S\rangle \in v_i$ such that $(lit, id) \in A\}$[3]

*Definition 4.10.* [A Restriction Holds] A restriction $\mathcal{R}$ *holds* in a version $v$ if

$$\forall o_i, o_j \in v\, \forall r_k, r_l \in \mathcal{R} : id_i \in r_k \wedge id_j \in r_l \Longrightarrow k = l$$

*Definition 4.11.* [Splitting of Versions by Restrictions] Given a version $v$ and a restriction $\mathcal{R}$, the result of splitting $v$ with respect to $\mathcal{R}$ is written $v \otimes \mathcal{R}$ and is

$$v \otimes \mathcal{R} = \begin{cases} \{v\} & \text{if the restriction } \mathcal{R} \text{ holds in } v \\ \{v_1, v_2\} & \text{otherwise} \end{cases}$$

where $v_1 = \{o = \langle id, p, A, S\rangle|\; o \in v \wedge id \in r_k\}$ and $v_2 = v - v_1$. The new program $P'$ is $P' = P - \{v\} \bigcup (v \otimes \mathcal{R})$.

*Example 4.1.* Consider the splitting of version $\{1,2,3,5\}$ by restriction $\{\{1\},\{2,3,4\},\{5\}\}$. $\{1,2,3,5\} \otimes \{\{1\},\{2,3,4\},\{5\}\} = \{\{1\},\{2,3,5\}\}$, but in $\{2,3,5\}$ the restriction does not hold yet. $\{2,3,5\} \otimes \{\{1\},\{2,3,4\},\{5\}\} = \{\{2,3\},\{5\}\}$. Now the restriction holds. Thus, the initial version is split into 3 versions: $\{\{1\},\{2,3\},\{5\}\}$.

*Theorem 4.2. Let $P'$ be a program obtained by applying* splitting of versions *to a program $P$. If $P$ is of maximal optimization, and minimal then $P'$ is also of maximal optimization and minimal.*

*Definition 4.12.* [$Program_f$] $Program_f$ is the program obtained from $program_i$ by splitting when all the restrictions hold , i.e., when a fixpoint is reached.

*Theorem 4.3 Multiple Specialization Algorithm. $Program_f$ is of maximal optimization, minimal, and feasible.*

By Theorem 4.2 $Program_f$ is of maximal optimization and minimal. We can see that it is also feasible because otherwise there would be a restriction that would not hold. This is in contradiction with the assumption that phase 2 (splitting) has terminated.

## *4.4. Structure of the Set of Programs and Termination*

As shown above, given $Analysis\_optimizations(P, p, \lambda, D_\alpha, opt)$, several programs may be generated from it. They may differ in size, optimizations, and even feasibility. In this section we discuss the structure of the set of such programs and the relations among its elements.

The set of programs as defined in Definition 4.3 forms a complete lattice under the $\sqsubseteq$ operation defined as follows. $P \sqsubseteq P'$ iff $\forall v \in P\ \exists v' \in P'$ s.t. $v \subseteq v'$, i.e., all the versions in $P$ are equal or more specific than the versions in $P$. The $\perp$ element of such a lattice will be given by the program with most specific versions. This is

[3]Note that $r_i$ may be $\emptyset$.

| $id_i$ | $p_i$ | $A_i$ | $S_i$ |
|---|---|---|---|
| 1 | go/2 | {(query,1)} | $\emptyset$ |
| 2 | p/3 | {(go/2/1/1,1)} | $\emptyset$ |
| 4 | p/3 | {(go/2/1/2,1)} | $\emptyset$ |
| 3 | plus/3 | {(p/3/1/1,2)} | {(plus/3/3/2,fail), (plus/3/2/1,true), (plus/3/1/2,true)} |
| 5 | plus/3 | {(p/3/1/1,4)} | {(plus/3/3/2,true), (plus/3/2/1,fail), (plus/3/1/2,fail)} |

**FIGURE 4.1.** *Analysis_optimizations* for the Example Program

the program with the greatest number of versions, i.e., the extended program. The $\top$ element is the program with most general versions, i.e, the one in which all the or–records that correspond to the same predicate are in the same version. This program corresponds to monovariant specialization.

Although not formally stated, the splitting operation used during phase 2 of the minimizing algorithm is an operator defined on this lattice since it receives a program as input and produces another program as output. Phase 2 starts with $Program_i$ and applies the splitting operator moving down in the lattice. Each splitting step transforms an infeasible program $P$ into (a less) infeasible program $P'$ s.t. $P' \sqsubseteq P$, until we reach a feasible program ($program_f$), which is a fixpoint of the splitting operator. As the splitting operator is monotonic and the lattice is finite phase 2 terminates.

## *4.5. Example*

We now apply the minimizing algorithm to the example program in Section 2.5. Figure 4.1 shows the starting point for the multiple specialization algorithm. The set of optimizations is empty in the or–record for go/2 and in the two or–records for p/3. It has three elements in the or–records for plus/3 that indicate the value that the test `integer` will take in execution. Note that the set of optimizations is different in these two or–records for plus/3. We represent each or–record only by its identifier. The two or–records for p/3 have the same optimizations (none) and can be joined. At the end of phase 1 we are in the following situation:

$Program_i$:

| go/2 | p/3 | plus/3 |
|---|---|---|
| {{1}} | {{2,4}} | {{3},{5}} |

Now we execute phase 2. Only plus/3 can produce restrictions. The other two predicates only have one version. The only restriction will be $\mathcal{R}_{p/3/1/1,plus/3} = \{\{2\},\{4\}\}$. The intuition behind this restriction is that or–record 2 must be in a different version than or–record 4. The restriction does not hold and thus $\{2,4\} \otimes \{\{2\},\{4\}\} = \{\{2\},\{4\}\}$. Now we must check if this splitting has introduced new

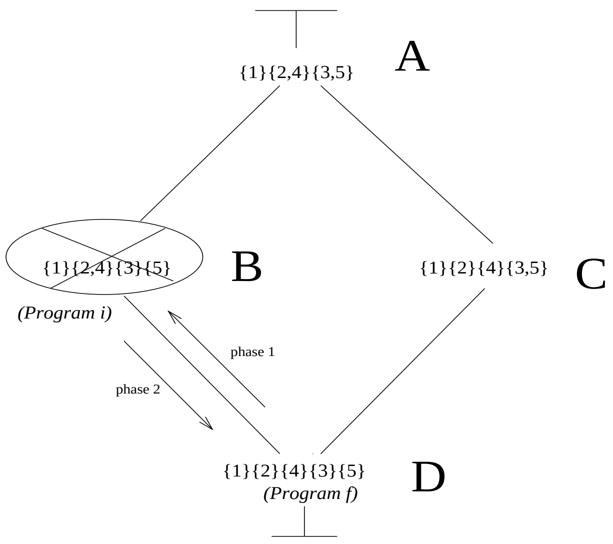


**FIGURE 4.2.** Lattice for the Example Program

restrictions. No new restriction appears because there is no literal that belongs to the ancestor information of both or–record 2 and or–record 4. Thus, the result of the algorithm will be:

$Program_f$:

| go/2 | p/3 | plus/3 |
|---|---|---|
| {{1}} | {{2},{4}} | {{3},{5}} |

The program that the minimization algorithm indicates that should be built coincides in this case with the extended program, which was already depicted in Figure 3.2.

Figure 4.2 shows the lattice of programs for the example. The node marked with a cross (B) corresponds to $program_i$ and is infeasible. That is why during phase 2 we move down in the lattice and return to node D Nodes B and D are of maximal optimization. A and C are not because or–records with different optimizations (3,5) are in the same version. Nodes A, C, and D are feasible. B is not feasible because for the literal p/3/1/1 it uses both or–record 3 and 5 (we cannot decide at compile-time which one to use). All the nodes in the lattice are minimal. A program is not minimal if two or–records that are equivalent are in different versions. No two or–records are equivalent and thus all the programs in the lattice are minimal.

## 5. THE APPLICATION: COMPILE-TIME PARALLELIZATION

The final aim of parallelism is to achieve the maximum speed (effectiveness) while computing the same solution (correctness) as the sequential execution. The two main types of parallelism which can be exploited in logic programs are well known [13, 11]: or-parallelism and and–parallelism. In this work we concentrate on the case of and–parallelism. And-parallelism refers to the parallel execution of the literals in the body of a clause (or, more precisely, of the goals in a resolvent). Several models have been proposed to take advantage of such opportunities (see, for example, [11] and its references).

Guaranteeing correctness and efficiency in and–parallelism is complicated by the fact that dependencies may exist among the goals to be executed in parallel, due to the presence of shared variables at run–time. It turns out that when these dependencies are present, arbitrary exploitation of and–parallelism does not guarantee efficiency. Furthermore, if certain impure predicates that are relatively common in Prolog programs are used, even correctness cannot be guaranteed.

However, if only *independent goals* are executed in parallel, both correctness and efficiency can be ensured [13, 26]. Thus, the dependencies among the different goals must be determined, and there is a related parallelization overhead involved. It is vital that such overhead remain reasonable. In order to achieve this, herein we follow the approach proposed initially in [60, 27] (see their references for alternative approaches) which combines local analysis and run–time checking with a data-flow analysis based on abstract interpretation [15].

## *5.1. The Annotation Process and Run-time Tests*

The annotation (parallelization) process can be viewed as a source to source transformation from standard Prolog to a parallel dialect. Herein, we will use the &-Prolog [24, 8] language as the target. This language is an extension to Prolog in which literals in a clause which may be executed in parallel are separated by `&` instead of the usual comma (,) symbol. Execution of literals separated by `&` is performed in parallel if sufficient processors are available. Otherwise they will be executed sequentially.

The task of deciding which literals may be executed in parallel is not an easy one because, as said before, if the involved literals are not *independent*, parallel execution may introduce inefficiency and even incorrectness. This is why it is desirable to automate the process of program parallelization. Herein, we will follow the approach used in the &-Prolog system [24, 8]. The automatic parallelization process is performed as follows [6]: Firstly, if requested by the user, the Prolog program is analyzed using one or more global analyzers. These analyzers [28, 48, 47] are aimed at inferring useful information for detecting independence. These analyses use the optimized fixpoint algorithm presented in [52]. Secondly, since side–effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [44] which propagates the side–effect characteristics of builtins determining the scope of side–effects. In the current implementation, side-effecting literals are not parallelized. Finally, the *annotators* perform a source–to–source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before.

The annotation process is divided into three subtasks. The first one is concerned with identifying the dependencies between each two literals in a clause and generating the conditions which ensure their independence. The second task aims at simplifying such conditions by means of the information inferred by the local or global analyzers. In other words, transforming the conditions into the minimum

```
:-module(mmatrix,[mmultiply/3]).

mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
        multiply(V1,V0,Result), mmultiply(Rest, V1, Others).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
        vmul(V0,V1,Result), multiply(Rest, V1, Others).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
        Product is H1*H2, vmul(T1,T2, Newresult),
        Result is Product+Newresult.
```

**FIGURE 5.1.** mmatrix.pl

number of tests which, when evaluated at run–time, ensure the independence of the goals involved. Finally, the third task is concerned with the core of the annotation process [6, 46], namely the application of a particular strategy to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step.

## *5.2. An Example: Matrix Multiplication*

We illustrate the process of automatic program parallelization with an example. Figure 5.1 shows the code of a Prolog program for matrix multiplication. The declaration `:-module(mmatrix,[mmultiply/3]).` is used by the (goal dependent) analyzer to determine that only calls to *mmatrix/3* may appear in top-level queries. In this case, no information is given about the arguments in calls to the predicate *mmatrix/3* (however, this could be done using one or more *entry* declarations [5]). If, for example, we want to specialize the program for the case in which the first two arguments of *mmatrix/3* are ground values and we inform the analyzer about this, the program would be parallelized without the need for any run-time tests. However, for the purposes of studying multiple specialization, we will consider the case in which no information at all is provided by the user regarding calling patterns, beyond the exported predicate information present in the module declaration. In this case the analyzer must in principle assume no knowledge regarding the instantiation state of the arguments at the module entry points.

Figure 5.2 contains the result of automatic parallelization under these assumptions. `if-then-else`s are written `(cond -> then ; else)`, i.e., using standard Prolog syntax. The `&` signs between goals indicate, as mentioned before, that they can be executed in parallel. The predicate *vmul/3* is not shown in Figure 5.2 because automatic parallelization has not detected any profitable parallelism in it (due to granularity control) and its code remains the same as in the original program.

It is clear from Figure 5.2 that a good number of run-time tests have been in-

```
mmultiply([],_,[]).
mmultiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply(V1,V0,Result) & mmultiply(Rest,V1,Others)
        ;    multiply(V1,V0,Result), mmultiply(Rest,V1,Others)).

multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply(Rest,V1,Others)).
```

**FIGURE 5.2.** Parallel mmatrix

troduced in the parallelization process. These tests are necessary to determine independence at run-time, given that nothing is known about the input arguments. If the tests succeed the parallel code is executed. Otherwise the original sequential code is executed. As usual, `ground(X)` succeeds if `X` contains no variables. `indep(X,Y)` succeeds if `X` and `Y` have no variables in common. For conciseness and efficiency, a series of tests `indep(X1,X2), ..., indep(Xn-1,Xn)` is written as `indep([[X1,X2], ..., [Xn-1,Xn]])`.

Even though groundness and independence tests are executed by efficient builtin predicates in the &-Prolog system, these tests may still cause considerable overhead in run-time performance, to the point of not even knowing at first sight if the parallelized program will offer speedup, i.e., if it will run faster than the sequential one. Our purpose is to study whether multiple specialization can be used to reduce the run-time test overhead and to increase speedups.

## 6. MULTIPLE SPECIALIZATION IN THE &-PROLOG COMPILER

Figure 6.1 (picture on the left) presents the role of abstract multiple specialization in the &-Prolog system. As stated in the previous section, automatic parallelization may introduce run-time tests and conditionals if the information available does not allow determining the dependence/independence of literals statically. As mentioned before, it is this checking overhead that the multiple specialization which has been added to the &-Prolog compiler and is the subject of our performance study is aimed at reducing. Note that because of the way the parallelization process is performed, if the same abstract domain is used to provide information to both the parallelization and specialization phases, none of the run-time tests introduced during parallelization is superfluous and thus none of them can be eliminated by the specializer unless multiple specialization is performed.

Even though not depicted in Figure 6.1, analysis information is not directly available at all program points after automatic parallelization, because the process modifies certain parts of the program originally analyzed. However, the &-Prolog

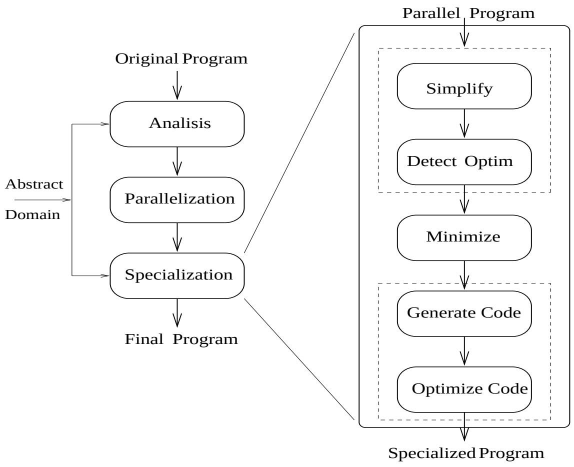


**FIGURE 6.1.** Program Parallelization and Abstract Multiple Specialization

system uses incremental analysis techniques to efficiently obtain updated analysis information from the one generated for the original program [25, 52].

Conceptually, the process of abstract multiple specialization is composed of five steps, which are shown in Figure 6.1 (picture on the right). In the first step (*simplify*) the program optimizations based on abstract execution are performed whenever possible. This saves having to optimize the different versions of a predicate when the optimization is applicable to all versions. Any optimization that is common to all versions of a predicate is performed at this stage. The output is a monovariant abstractly specialized program. This is also the final program if multiple specialization is not performed. The remaining four steps are related to *multiple* specialization.

In the second step (*detect optimizations*) information from the multivariant abstract interpretation is used to detect (but not to perform) the optimizations allowed in each version. Note that only one step of analysis is required in our system in order to both compute the set of or–records for each predicate and the optimizations allowed for each one of them. This is only possible if we can identify the abstract substitutions for the different or–records at each program point. In our analyzer this is done by just storing the or–record identifiers along with each substitution generated by multivariant analysis. Even though the addition of this identifier to abstract substitutions may seem an overhead, they will be used as detailed dependencies while computing the analysis graph. This will allow analysis to be more efficient [52], i.e., converging faster to a fixpoint, and incremental [25].

Note that the source for the multiply specialized program has not been generated yet (this will be done in the fourth step, *generate code*) but rather the code generated in the first step is used, considering several abstract substitutions for each program point instead of their least upper bound, as is done in the first step. The output of this step is $Analysis\_optimizations$. Note that these optimizations are not possible without multiple specialization, otherwise the optimization would have already been performed in the first step (*simplify*).

The third step (*minimize*) implements the minimizing algorithm presented in

Section 4.

In the fourth step (*generate code*) the source code of the minimal multiply specialized program is generated. Each version receives a unique name. Also, literals must also be renamed appropriately for a predicate with several implementations.

In the fifth step (*optimize code*), the particular optimizations associated with each implementation of a predicate are performed. Other simple program optimizations like eliminating literals in a clause to the right of a literal abstractly executable to false, eliminating a literal which is abstractly executable to true from the clause it belongs instead of introducing the builtin `true/1`, dead code elimination, etc. are also performed in this step.

In the implementation, for the sake of efficiency, the first and second steps, and the fourth and fifth are performed in one pass (this is marked in Figure 6.1 by dashed squares), thus reducing to two the number of passes through the source code. The third step is not performed on source code but rather on a synthetic representation of sets of optimizations and versions. The core of the multiple specialization technique (steps *minimize* and *generate code*) is independent of the actual optimizations being performed.

## *6.1. Abstract Execution*

In the &-Prolog compiler, most optimizations that are relevant in our context are performed by means of abstract executability. This concept was, to our knowledge, first introduced informally in [22]. It allows reducing at compile-time certain literals in a program to the value *true* or *false* using information obtained with abstract interpretation. That work also introduced some simple semantics-preserving program transformations and showed the potential of the technique, including elimination of invariants in loops. We summarize in the following an improved formalization of abstract executability. A more detailed formalization can be found in [53]. In what follows, the set of variables in a literal $L$ is represented as $var(L)$. The restriction of the substitution $\theta$ to $var(L)$ is denoted $\theta|_L$.

Operationally, each literal $L$ in a program $P$ can be viewed as a procedure call. Each run-time invocation of the procedure call $L$ will have a local *environment* $e$, which stores the particular values of each variable in $var(L)$ for that invocation. We will write $\theta \in e(L)$ if $\theta$ is a substitution such that the value of each variable in $var(L)$ is the same in the environment $e$ and the substitution $\theta$.

*Definition 6.1.* [Run-time Substitution Set] Given a literal $L$ from a program $P$ we define the *run-time substitution set* of $L$ in $P$ as

$$RT(L,P) = \{\theta|_L : e \text{ is a run-time environment for } L \text{ and } \theta \in e(L)\}$$

$RT(L,P)$ is not computable in general. However, we can use information on $RT(L,P)$ provided by abstract interpretation, i.e., the abstract call substitution for $L$.

*Definition 6.2.* [Trivial Success Set] Given a literal $L$ from a program $P$ we define the *trivial success set* of $L$ in $P$ as

$$TS(L,P) = \begin{cases} \left\{ \begin{array}{l} \theta|_L : L\theta \text{ succeeds exactly once in } P \\ \text{with empty answer substitution } (\epsilon) \end{array} \right\} & \text{if L is pure} \\ \emptyset & \text{otherwise} \end{cases}$$

*Definition 6.3.* [Finite Failure Set] Given a literal $L$ from a program $P$ we define the *finite failure set* of $L$ in $P$ as

$$FF(L,P) = \begin{cases} \{\theta|_L : L\theta \text{ fails finitely in } P\} & \text{if L is pure} \\ \emptyset & \text{otherwise} \end{cases}$$

*Definition 6.4.* [Elementary Literal Replacement] *Elementary Literal Replacement* (ER) of a literal $L$ in a program $P$ is defined as:

$$ER(L,P) = \begin{cases} true & \text{if } RT(L,P) \subseteq TS(L,P) \\ false & \text{if } RT(L,P) \subseteq FF(L,P) \\ L & \text{otherwise} \end{cases}$$

The idea is to optimize a program by replacing whenever possible the execution of $L\theta$ with the execution of either the builtin predicate *true* or $fail$, which can be executed in zero or constant time. Even though the above optimization may seem not very widely applicable, for many builtin predicates such as those that check basic types or meta-logical predicates that inspect the instantiation state of terms and as we will see in Section 7, this optimization is indeed very relevant. Another example of this not related to program parallelization is the optimization of delay conditions in logic programs with dynamic scheduling [50].

Unfortunately, elementary replacement is not directly applicable because $RT(L,P)$, $TS(L,P)$, and $FF(L,P)$ are generally not known at specialization time. However, we will identify sufficient conditions which guarantee its applicability.

## 6.2. Abstract Execution of Builtin Predicates

Even though abstract executability is applicable to any predicate, in what follows we will concentrate on builtin predicates. This is because the semantics of builtin predicates does not depend on the particular program in which they appear. As a result, we can compute sets of abstract values $A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ once and for all for each builtin predicate $B$, where $\overline{B}$ stands for the *base form* of B, i.e., all the arguments of $B$ contain distinct free variables. Such sets will be applicable to all literals that call the builtin predicate in any program.

*Definition 6.5.* [Operational Abstract Execution of Builtins] *Operational abstract execution* (OAEB) of a literal $L$ with abstract call substitution $\lambda$ that calls a builtin predicate $B$ is defined as:

$$OAEB(L, D_\alpha, \lambda) = \begin{cases} true & \begin{array}{l} \text{if } \exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \\ \quad call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda' \end{array} \\ false & \begin{array}{l} \text{if } \exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \\ \quad call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda' \end{array} \\ L & \text{otherwise} \end{cases}$$

$A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ are approximations of $TS(\overline{B}, P)$ and $FF(\overline{B}, P)$ respectively for any P (this is possible because the semantics of the builtin predicates

| Domain | $TS(gr(X_1))$ | $FF(gr(X_1))$ | $TS(ind(X_1))$ | $FF(ind(X_1))$ |
|---|---|---|---|---|
| sharing | O | N | O | N |
| sh+fr | O | S | O | S |
| asub | O | N | O | N |

**TABLE 6.1.** Optimality of Different Domains

does not depend on the program in which they appear). For soundness it is required that both $\forall \lambda \in A_{TS}(\overline{B}, D_\alpha)\ \gamma(\lambda) \subseteq TS(\overline{B}, P)$ and $\forall \lambda \in A_{FF}(\overline{B}, D_\alpha)\ \gamma(\lambda) \subseteq FF(\overline{B}, P)$.

There is no automated method that we are aware of to compute $A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ for each builtin predicate $\overline{B}$. However, we believe that a good knowledge of $D_\alpha$ allows finding safe approximations, and that in many cases it is easy to find the best possible approximations $A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$.

*Example 6.1.*

Suppose we are interested in optimizing calls to the builtin predicate *ground/1* by reducing them to the value true. Then, $TS(ground(X_1)) = \{\{X_1/g\}$ where $g$ is any term without variables $\}$. Suppose also that we use the abstract domain $D_\alpha$ of Section 2.5 consisting of the five elements $\{bottom, int, float, free, top\}$. Then, we can take $A_{TS}(ground(X_1), D_\alpha) = \{int, float\}$. Consider the following clause containing the literal $ground(X)$:

$$p(X,Y) :\!-\ q(Y), ground(X), r(X,Y).$$

Assume now that analysis has inferred the abstract substitution just before the literal $ground(X)$ to be $\{Y/free, X/int\}$. Then $OAEB(ground(X), D_\alpha, X/int) = true$ (the literal can be replaced by $true$) because $call\_to\_entry(ground(X), ground(X_1), D_\alpha, \{X/int\}) = \{X_1/int\}$, and $X_1/int \sqcup X_1/int = X_1/int$.

If we were also interested in reducing literals that call *ground/1* to false, the most accurate $A_{FF}(ground(X_1), D_\alpha) = \{free\}$

## *6.3. Abstract Domains for Specialization*

The abstract specializer is parametric with respect to the abstract domain used. Currently, the specializer can work with all the abstract domains implemented in the analyzer in the &-Prolog system. In order to augment the specializer to use the information provided by a new abstract domain ($D_\alpha$), correct $A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ sets must be provided to the analyzer for each builtin predicate $B$ whose optimization is of interest. Alternatively, and for efficiency issues, the specializer allows replacing the conditions in Definition 6.5 with specialized ones because in $\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda'$ all values are known before specialization time except for $\lambda$ which will be computed by analysis. I.e., conditions can be partially evaluated with respect to $D_\alpha$, $\overline{B}$ and a set of $\lambda'$, as they are known in advance.

Table 6.1 shows the accuracy of a number of abstract domains (*sharing* [28, 48], *sharing+freeness* (sh+fr) [47], and *asub* [56, 12]) present in the &-Prolog system

```
mmultiply([],_,[]).
mmultiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply1(V1,V0,Result) & mmultiply1(Rest,V1,Others)
        ;    multiply2(V1,V0,Result), mmultiply(Rest,V1,Others)).
mmultiply1([],_,[]).
mmultiply1([V0|Rest],V1,[Result|Others]) :-
        (indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply1(V1,V0,Result) & mmultiply1(Rest,V1,Others)
        ;    multiply1(V1,V0,Result), mmultiply1(Rest,V1,Others)).

multiply1([],_,[]).
multiply1([V0|Rest],V1,[Result|Others]) :-
        (ground(V1), indep([[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply3(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply1(Rest,V1,Others)).
multiply2([],_,[]).
multiply2([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply4(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply2(Rest,V1,Others)).
multiply3([],_,[]).
multiply3([V0|Rest],V1,[Result|Others]) :-
        (indep([[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply3(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply3(Rest,V1,Others)).
multiply4([],_,[]).
multiply4([V0|Rest],V1,[Result|Others]) :-
        (indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply4(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply4(Rest,V1,Others)).
```

**FIGURE 6.2.** Specialized mmatrix

with respect to the run-time tests (i.e., `ground/1`, `indep/1`). The three of them are optimal for abstractly executing both types of tests to true, i.e., it is possible to find a set $A_{TS}(\overline{B}, D_\alpha)$ s.t. $\gamma(A_{TS}(\overline{B}, D_\alpha)) = TS(\overline{B})$. However, only sharing+freeness (sh+fr) allows abstractly executing these tests to false, even though not in an optimal way, i.e., $\emptyset \subset \gamma(A_{FF}(\overline{B}, D_\alpha)) \subset FF(\overline{B})$).

*Example 6.2.*
The resulting program after abstract multiple specialization is performed is shown in Figure 6.2. The program generated in our implementation is equivalent to the one presented except that internal names are used for specialized versions to avoid clashes with other user defined predicates. Two versions have been generated for the predicate *mmultiply/3* and four for the predicate *multiply/3*. As in

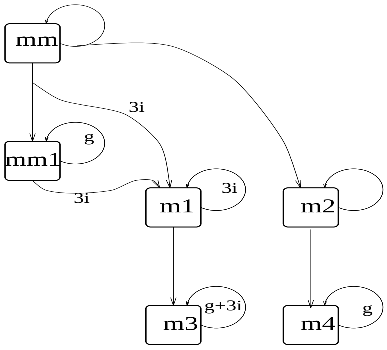


**FIGURE 6.3.** Call Graph of Specialized mmatrix

Figure 5.2, the predicate *vmul/3* is not presented in the figure because its code is identical to the one in the original program in Figure 5.1 (and the parallelized program). Only one version has been generated for this predicate even though multivariant abstract interpretation generated eight different variants for it. As no further optimization is possible by implementing several versions of *vmul/3*, the minimization algorithm has collapsed all the different versions of this predicate into one.

It is important to mention that abstract multiple specialization is able to automatically detect and extract some invariants in recursive loops: once a certain run-time test has succeeded it does not need to be checked in the following recursive calls [22]. Figure 6.3 shows the call graph of the specialized program of Figure 6.2. `mm` stands for mmultiply and `m` for multiply. Edges are labeled with the number of tests which are avoided in each call to the corresponding version with respect to the non specialized program. For example, `g+3i` means that each execution of this specialized version avoids a groundness and three independence tests. It can be seen in the figure that once the groundness test in any of `mm`, `m1`, or `m2` succeeds, it is detected as an invariant, and the more optimized versions `mm1`, `m3`, and `m4` respectively will be used in all remaining iterations.

## 7. EXPERIMENTAL RESULTS

In this section we present a series of experimental results. The primary aim of these experiments is to assess whether performing abstract multiple specialization for improving automatically parallelized programs is profitable or not. This assessment will be realized by studying some of the cost/benefit tradeoffs involved in multiple specialization, in terms of time and space. Even though the results have been obtained in the context of a particular implementation and type of optimizations, we believe that it is possible to derive some conclusions from the results regarding the cost and benefits of multiple specialization in general.

The benchmarks considered have been automatically parallelized in the &-Prolog system using *strict* independence as a safety and efficiency condition for paralleliza-

| Bench | Ana | Par | ReA | Spec | Total | SD |
|---|---|---|---|---|---|---|
| aiakl | 1.40 | 1.59 | 0.36 | 0.06 | 3.41 | 1.14 |
| ann | 3.24 | 2.11 | 2.01 | 0.90 | 8.26 | 1.54 |
| bid | 0.34 | 0.20 | 0.24 | 0.23 | 1.00 | 1.88 |
| boyer | 0.97 | 0.27 | 0.37 | 0.32 | 1.94 | 1.56 |
| browse | 0.17 | 0.11 | 0.26 | 0.25 | 0.78 | 2.82 |
| deriv | 0.18 | 0.20 | 0.42 | 0.13 | 0.93 | 2.46 |
| hanoiapp | 0.28 | 0.22 | 0.14 | 0.04 | 0.69 | 1.37 |
| mmatrix | 0.13 | 0.08 | 0.17 | 0.08 | 0.45 | 2.20 |
| occur | 0.12 | 0.06 | 0.18 | 0.08 | 0.43 | 2.38 |
| progeom | 0.09 | 0.06 | 0.03 | 0.05 | 0.22 | 1.53 |
| qplan | 0.74 | 1.23 | 0.15 | 0.46 | 2.58 | 1.31 |
| query | 0.04 | 0.04 | 0.04 | 0.06 | 0.19 | 2.26 |
| read | 7.26 | 2.39 | 0.02 | 0.63 | 10.31 | 1.07 |
| serialize | 0.26 | 0.18 | 0.03 | 0.06 | 0.52 | 1.19 |
| warplan | 1.21 | 0.21 | 0.11 | 0.26 | 1.79 | 1.26 |
| zebra | 2.27 | 17.25 | 2.02 | 0.06 | 21.59 | 1.11 |
| | | | | | **Overall** | 1.23 |

**TABLE 7.1.** Specialization and Parallelization Times (Using No Call Pattern Info)

tion [26], the *mel* [46] heuristic algorithm for the generation of parallel expressions and the $sharing + freeness$ abstract domain [47] to introduce as few run-time tests as possible. Such combination of techniques has been experimentally shown [7] to be capable of effectively parallelizing logic programs with quite reasonable run-time overhead for checking independence, producing useful speedups in parallel execution.

In [7], in order to compute $Analysis(P, p, \lambda, sharing + freeness)$, reasonable calling patterns $(p, \lambda)$ for each program $P$ were given: $p$ is the exported predicate (i.e., the predicate accessible from outside the module being analyzed) and $\lambda$ an accurate description of the instantiation state of the arguments of $p$. However, in the current set of experiments we study what is a very unfavorable situation for automatic parallelization: the calling pattern for each program $P$ is $(p, \top)$. As before, $p$ is the exported predicate in $P$. $\top$ is the most general abstract substitution for $p$, which is equivalent to providing no information to the analyzer regarding the possible input values for $p$. This situation is interesting in that it appears when modules written by naïve users are compiled in isolation. Since as a result of this the analyzer will sometimes have incomplete information, a large number of run-time tests will in some cases be included in the resulting programs, which are then potential targets for multiple specialization. The relatively wide set of benchmarks considered is the subset of the benchmarks used in [7] for automatic parallelization (available at `http://clip.dia.fi.upm.es`) which cannot be parallelized without the need of run-time tests when using $(p, \top)$ as calling pattern. The other benchmarks (fib, qsortapp, tak, and witt) are parallelized without run-time tests even in this case and are therefore not studied in this work.

| Bench | Pred | Max | min | Ind | M(%) | m(%) | I(%) | Ratio |
|---|---|---|---|---|---|---|---|---|
| aiakl | 9 | 4 | 0 | 0 | 44 | 0 | 0 | 1.44 |
| ann | 77 | 70 | 29 | 16 | 90 | 37 | 21 | 1.39 |
| bid | 22 | 39 | 9 | 4 | 177 | 40 | 18 | 1.97 |
| boyer | 27 | 57 | 9 | 7 | 211 | 33 | 26 | 2.33 |
| browse | 9 | 19 | 15 | 7 | 211 | 166 | 78 | 1.17 |
| deriv | 5 | 5 | 5 | 1 | 100 | 100 | 20 | 1.00 |
| hanoiapp | 3 | 10 | 2 | 1 | 333 | 66 | 33 | 2.60 |
| mmatrix | 3 | 11 | 4 | 0 | 366 | 133 | 0 | 2.00 |
| occur | 5 | 15 | 7 | 3 | 300 | 140 | 60 | 1.67 |
| progeom | 10 | 5 | 0 | 0 | 50 | 0 | 0 | 1.50 |
| qplan | 48 | 17 | 6 | 4 | 35 | 12 | 8 | 1.20 |
| query | 6 | 1 | 0 | 0 | 16 | 0 | 0 | 1.17 |
| read | 25 | 52 | 0 | 0 | 208 | 0 | 0 | 3.08 |
| serialize | 6 | 3 | 0 | 0 | 50 | 0 | 0 | 1.50 |
| warplan | 37 | 130 | 42 | 29 | 351 | 113 | 78 | 2.11 |
| zebra | 7 | 10 | 0 | 0 | 142 | 0 | 0 | 2.43 |
| **Overall** | | | | | 147 | 43 | 24 | 1.73 |
| **Relative Overall** | | | | | 208 | 80 | 33 | 1.72 |

**TABLE 7.2.** Number of Versions

## *7.1. The Cost of Multiple Specialization*

In order to assess the cost of specialization in terms of compilation time, Table 7.1 compares the analysis, parallelization, and specialization times for each benchmark. We argue that it is reasonable to compare these times as the programs that accomplish those tasks are implemented using the same technology, are integrated in the same system, they share many data structures, and work with the same input program (or slightly modified versions of it). Times are in seconds on a Sparc 1000. **Ana** is the time taken to analyze the original program, **Par** is the parallelization time, **ReA** is the reanalysis time required to update analysis information after parallelization in an incremental way, using the algorithms described in [25], and **Spec** is the multiple specialization time which includes computing the possible optimizations in each version using the notion of abstract executability, minimizing the number of versions, and materializing the new program in which the new versions are optimized (using source to source transformations). The time required for automatic parallelization is the sum of **Ana** and **Par**. The cost of multiple specialization should be viewed as **ReA** plus **Spec** as specialization requires analysis information to be up to date. **Total** gives the total time required for the whole process. The last column, **SD** is the slow-down introduced by multiple specialization in the parallelization process and is computed as **Total**/(**Ana**+**Par**). Finally, **Overall** gives the slowdown obtained by taking for each column the sum of times for all benchmarks. The results can be interpreted as indicating that performing multiple specialization after parallelization slows down the compilation process over all benchmarks approximately by a factor of 1.23.

It appears that the time required for multiple specialization, at least in this ap-

| Bench | Orig | Par | Spec | P/O | S/O | S/P |
|---|---|---|---|---|---|---|
| aiakl | 3317 | 4667 | 4386 | 1.41 | 1.32 | 0.94 |
| ann | 43368 | 55402 | 66776 | 1.28 | 1.54 | 1.21 |
| bid | 10242 | 14159 | 17031 | 1.38 | 1.66 | 1.20 |
| boyer | 37340 | 38273 | 43030 | 1.02 | 1.15 | 1.12 |
| browse | 3460 | 5977 | 11013 | 1.73 | 3.18 | 1.84 |
| deriv | 2747 | 5957 | 10299 | 2.17 | 3.75 | 1.73 |
| hanoiapp | 1115 | 2120 | 3014 | 1.90 | 2.70 | 1.42 |
| mmatrix | 1257 | 3048 | 5802 | 2.42 | 4.62 | 1.90 |
| occur | 2093 | 3270 | 6377 | 1.56 | 3.05 | 1.95 |
| progeom | 3510 | 4334 | 4174 | 1.23 | 1.19 | 0.96 |
| qplan | 35155 | 36679 | 38501 | 1.04 | 1.10 | 1.05 |
| query | 7313 | 8816 | 8563 | 1.21 | 1.17 | 0.97 |
| read | 23147 | 23718 | 23556 | 1.02 | 1.02 | 0.99 |
| serialize | 2994 | 3749 | 3622 | 1.25 | 1.21 | 0.97 |
| warplan | 22788 | 23047 | 19922 | 1.01 | 0.87 | 0.86 |
| zebra | 3645 | 4912 | 4842 | 1.35 | 1.33 | 0.99 |
| **Overall** | | | | 1.17 | 1.33 | 1.14 |
| **Relative Overall** | | | | 1.18 | 1.39 | 1.18 |

**TABLE 7.3.** Size of Programs

plication, is reasonable. However, a potentially greater concern than compilation time can be the increase in program size. Table 7.2 shows a series of measurements relevant to this issue. **Pred** is the number of predicates in the original program. **Max** is the number of additional (versions of) predicates that would be introduced if the minimization algorithm were not applied (when adding it to **Pred** this is also the number of versions that the analyzer implicitly uses internally during analysis). **Min** is the number of additional versions if the minimization algorithm is applied. As mentioned before, sometimes, in order to achieve an optimization some additional versions have to be created just to create a "path" to another optimized version, i.e. to make the program feasible (using the terminology of Section 6.1). The impact of this is measured by **Ind** which represents the number of such "Indirect" versions in the minimized program that have been included during phase 2 of the algorithm. I.e., this is the number of versions which have the same set of optimizations as an already existing version for that predicate.

We observe that for some benchmarks **Min** is 0. This means that multiple specialization has not been able to optimize the benchmark any further. That is, the final program equals the original program. However, note that if we did not minimize the number of versions the program size would be increased even though no additional optimization is achieved. **M(%)** is computed as $\frac{Max}{Preds} \times 100$. **m(%)** and **I(%)** are computed similarly but replacing **Max** by **Min** and **Ind** in the formula respectively. Finally **Ratio** is the relation between the sizes (in number of predicates) of the multiply specialized programs with and without minimization. The last rows of Table 7.2 show two different overall figures. The first is computed considering all the benchmark programs and the second considering only the programs

| Bench | Orig/Par | Orig/Spec | Par/Spec | Improv(%) |
|---|---|---|---|---|
| ann | 0.68 | 0.69 | 1.01 | 4 |
| bid | 0.63 | 0.71 | 1.11 | 28 |
| boyer | 0.82 | 0.85 | 1.03 | 18 |
| browse | 0.64 | 53.54 | 84.03 | — |
| brow_nf | 0.86 | 0.89 | 1.04 | 27 |
| deriv | 0.19 | 0.21 | 1.11 | 12 |
| hanoiapp | 0.60 | 0.75 | 1.26 | 51 |
| mmatrix | 0.43 | 0.86 | 2.02 | 88 |
| occur | 0.84 | 1.01 | 1.21 | 107 |
| qplan | 0.97 | 0.99 | 1.02 | 70 |
| warplan | 1.00 | 1.00 | 1.00 | — |

**TABLE 7.4.** Sequential Performance

in which the specialization method has obtained some optimization (**Min**$> 0$).

According to the overall figures, the specialized program has 43% additional versions with respect to the original program. However, this average greatly depends on the number of possible optimization points in the original program (in our case run-time tests) and cannot be taken as a general result. Of much more relevance are the ratios between **M(%)** and **N(%)**, and between **I(%)** and **m(%)**. The first ratio measures the effectiveness of the minimization algorithm. This ratio is 3.41 or 2.6 using global or relative averages respectively. I.e., the minimizing algorithm is able to reduce to a third the number of additional versions needed by multiple specialization. The second ratio represents how many of the additional versions are indirect. It is 56% or 41% (Global or Relative). This means that half of the additional versions are due to indirect optimizations. Another way to look at this result is as meaning that on the average there is one intermediate, indirect predicate between an originating call to an optimized, multiply specialized predicate and the actual predicate. It seems that this can in many cases be an acceptable cost in return for no run-time overhead in version selection.

Another pragmatic and very significant way of comparing the cost in program size incurred by multiple specialization is by comparing the size of the compiled programs (in bytecode quick-load format) before and after multiple specialization. For reference, we also compare to the size of the byte code for the original program. Table 7.3 presents the size in bytes of the original (**Orig**), parallelized (**Par**), and specialized (**Spec**) programs in bytes for &-Prolog . **P/O** gives the increase in size due to parallelization, and **S/O** the increase due to the composition of specialization and parallelization with respect to the original program. **S/P** presents the cost in space incurred by multiple specialization alone. As in Table 7.2, two cases have been considered for computing the overall space cost of multiple specialization: **Overall**, in which all benchmarks are considered, and **Relative Overall** in which only those benchmarks which benefit from multiple specialization are considered. The results can be interpreted as indicating that, in our system, multiple specialization increases program size by a ratio of 1.14 or 1.18 (relative). This increase is very similar to that introduced by parallelization (1.17 – 1.18) in the set of benchmarks considered. Finally, when multiple specialization and parallelization are composed, the overall increase in program size is around 1/3 even in the unfavorable case studied of not giving any information to the analyzer regarding the instantiations of the input arguments of exported predicates.

Note that the cost in program size for multiple specialization presented in Table 7.3 is better than that presented in Table 7.2. There are several reasons for this. First, the specializer performs some degree of dead-code elimination. Second, abstract executability allows in many cases performing source to source transformations which shorten the program, e.g., by simplifying a conditional, eliminating one of the branches in an if–then–else, etc. Third, because the number of additional versions is not necessarily a good estimate of program size as this will greatly depend on the size of the predicates which are being replicated.

### *7.2. Benefits of Multiple Specialization*

Having discussed the cost of multiple specialization in automatic parallelization both in terms of time and space, we now measure experimentally the benefits introduced by multiple specialization.

The addition of run-time tests and conditionals in parallelized programs will introduce some overhead which can be seen as extra work to be performed at run-time. A pragmatical approach to avoid this overhead is to simply annotate sequential execution when the tests cannot be proved statically to succeed. However, it has been proved that performing run-time independence tests can produce speedups [7]. Table 7.4 shows the slow-downs with respect to the original program of the parallelized (**Par**) and specialized (**Spec**) programs. The main contribution of multiple specialization in program parallelization will be in reducing the overhead of run-time tests and conditionals further, i.e., by getting a high value for **Orig/Spec**. This value will be 1 when the overhead of run-time tests has been completely eliminated and will not be much higher than 1 if the original program was optimally written, i.e., by an experienced programmer. Note that programs which do not benefit from multiple specialization are not considered in Table 7.4 as they contain no test which can be eliminated by multiple specialization. The **Par/Spec** column provides the sequential speedup achieved due to multiple specialization. It is always greater than 1, i.e., no slow-downs are introduced. Speedups range from a small 1.01 for ann to 2.02 for mmatrix.

In the case of browse, the original benchmark contains the clause:

```
p_match([P|Patterns],D) :-
        (match(D,P), fail; true),
        p_match(Patterns, D).
```

where `match(D,P)` produces no side-effects. The specializer transforms this clause into:

```
p_match([P|Patterns],D) :-
        p_match(Patterns, D).
```

and all the work performed in the calls to `match/2` is eliminated from the execution. In order to isolate the effects of multiple specialization from these optimizations (which can be performed without generating different versions of the predicate)

| Bench | $\mathbf{P}_5$ | $\mathbf{S}_5$ | $\mathbf{I}_5$ | $\mathbf{P}_{10}$ | $\mathbf{S}_{10}$ | $\mathbf{I}_{10}$ | $\mathbf{P}_{\#p}$ | $\mathbf{S}_{\#p}$ | $\mathbf{I}_\infty$ |
|---|---|---|---|---|---|---|---|---|---|
| ann | 2.40 | 2.39 | 1.00 | 3.34 | 3.59 | 1.07 | $4.30_{53}$ | $4.33_{52}$ | 1.01 |
| bid | 1.13 | 1.27 | 1.12 | 1.13 | 1.27 | 1.12 | $1.13_9$ | $1.27_9$ | 1.13 |
| boyer | 0.82 | 0.85 | 1.04 | 0.82 | 0.85 | 1.04 | $0.82_7$ | $0.85_7$ | 1.03 |
| brow_nf | 1.85 | 1.89 | 1.02 | 2.03 | 2.07 | 1.02 | $2.12_{124}$ | $2.17_{130}$ | 1.02 |
| deriv | 0.79 | 0.86 | 1.09 | 1.20 | 1.24 | 1.03 | $1.36_{175}$ | $1.38_{166}$ | 1.02 |
| hanoi | 0.89 | 1.18 | 1.33 | 0.89 | 1.18 | 1.33 | $0.82_{34}$ | $1.10_6$ | 1.33 |
| mmat | 1.94 | 3.94 | 2.03 | 3.56 | 7.33 | 2.06 | $5.03_{47}$ | $15.01_{56}$ | 2.98 |
| occur | 3.96 | 4.75 | 1.20 | 6.34 | 8.84 | 1.39 | $9.85_{34}$ | $28.29_{108}$ | 2.87 |
| qplan | 1.31 | 1.35 | 1.03 | 1.31 | 1.35 | 1.03 | $1.31_4$ | $1.35_3$ | 1.03 |
| warplan | 1.07 | 1.07 | 1.00 | 1.07 | 1.07 | 1.00 | $1.07_5$ | $1.07_5$ | 1.00 |

**TABLE 7.5.** Parallel Performance

we have studied instead a modified version of the benchmark, *brow_nf*, which is obtained by removing the call to fail after `match(D,P)` in the original benchmark and eliminating the clause

```
property([],_X,_Y) :- fail.
```

which is also eliminated automatically by the specializer.

Finally, column **Improv(%)** is computed for each benchmark as $\frac{Spec-Par}{1-Par} \times 100$ and gives an idea of the degree to which multiple specialization in the &-Prolog system has accomplished its primary task, i.e., eliminating the overhead introduced by the run-time tests and conditionals as much as possible. Note that this figure makes no sense for *browse.pl* as the improvement is much beyond the overhead of run-time tests, and is thus not presented. This figure is not given for *warplan* either since the overhead introduced by the run-time tests is insignificant.

Another interesting question is how the improvement in sequential execution time, i.e., the reduction of total work to be performed, affects performance in parallel execution, which is of course the ultimate objective of the parallelizing compiler. Due to the simulation approach used (described below) the programs have to be executed on quite small data, which results in small speedups. However, note that we are not interested really in the absolute speedups, but rather in the relative improvement in such speedups due to multiple specialization. To this end we compare the execution speed of the original program with the parallelized (**P**) and specialized (**S**) programs running on several processors and show the results in Table 7.5. The improvement in parallel execution speed due to specialization, **P/S**, is given by column **I**. This is done for three different cases. In the first one five processors are available and dedicated to the execution of the program. In the second, ten processors are used, and in the third an unlimited number of processor can be used, i.e., it gives an estimate of the best possible parallel performance. These three cases are distinguished by the subindex $_5$, $_{10}$ and $_\infty$ respectively. Additionally, in the columns $\mathbf{P}_{\#p}$ and $\mathbf{S}_{\#p}$, an upper bound on the number of processor required to achieve such optimal speed for each benchmark is given as a subindex. These speedup figures have been obtained with the IDRA simulation tool [19]. This tool allows obtaining speedup results which have been shown to match closely the actual speedups obtained in the &-Prolog system for the number of processors available for comparison. It is also believed that the results obtained are good approximations of the best possible parallel execution for larger numbers of processors [19]. This approach allows concentrating on the available parallelism, without the limitations imposed by a fixed number of physical processors, a particular scheduling, bus bandwidth, etc. IDRA takes as input an execution trace file generated from the execution of a parallelized program on one or more processors and the time taken by the sequential program, and computes the achievable speedup for any number of processors. The trace files list the events occurred during the execution of the parallel program, such as a parallel goal being started or finished, and the times at which the events occurred. Since &-Prolog normally generates all possible parallel tasks in a parallel program, regardless of the number of processors in the system, information is gathered for all possible goals that would be executed in parallel. Using this data, IDRA builds a task dependency graph whose edges are annotated with the exact execution times. The possible actual execution graphs (which could be obtained if more processors were available) are constructed from this data and their total execution times compared to the sequential time, thus making quite accurate estimations of (ideal — in the sense that some low level overheads are not taken into account) speedups.

## 8. DISCUSSION

The experimental results presented in Section 7 allow us to conclude that, at least in the application considered, abstract multiple specialization is a useful technique: its costs are reasonable and the benefits of sufficient significance. Summarizing the results in terms of compilation time, the additional time required for specialization is about 1/4 of the parallelization time. Regarding the size of the specialized program, it is about 1/6 larger than the parallelized one and about 1/3 larger than the original one. Regarding the actual benefits of multiple specialization in terms of speedup, it varies greatly from one benchmark to another. Thus, it is not easy to give a factor which summarizes the achievable speedup, but many programs do obtain useful speedups. Note also that if our primary aim when performing multiple specialization is, as in the experiments, to reduce the overhead introduced by independence run-time tests, in the relatively frequent case in which automatic parallelization does not require the introduction of any run-time test, specialization can be easily turned off and only applied to those cases which are problematic to automatic parallelization.

If the particular optimizations being considered are appropriate, multiple specialization always generates programs which are, at least theoretically, more optimized than the original. This is confirmed by column **Par/Spec** of Table 7.4, which for all benchmarks presents values greater than 1. Leaving the atypical case of browse.pl aside, the results show that the sequential improvement is low for some benchmarks (ann, qplan, warplan), significant in others (bid, hanoi, occur), and very important in others (mmatrix). This program (Figure 5.1), is a reasonable candidate for parallelization and its execution time decreases nearly linearly with the number of processors. Note, however, that if the user provides enough information regarding the input, this program would be parallelized in the &-Prolog compiler without any run-time tests. However, if no information is provided by the user (the case studied) many such tests are generated and performance decreases. The reason for obtaining such improved speedups for mmatrix when multiple specialization is used is that it is a recursive program in which specialization automatically detects and extracts an invariant, as explained in Example 6.2.

Another important conclusion which the experiments seem to bear is that the speedup achieved by multiple specialization generally increases with the number of processors, thus making multiple specialization quite relevant in the context of a parallelizing compiler. The reason for this is that, in general, specialization reduces the overhead of parallelization but does not deeply transform the structure of tasks to be performed: the length of some tasks will be shortened due to the elimination of run-time tests. This is the case for most benchmarks studied. The main exception is `deriv.pl`, which is a program for symbolic differentiation and also a good candidate for parallelization. However, the improvement obtained with specialization is 1.11 for one processor and it decreases to a low 1.02 with 130 processors. This shows that not all programs with significant parallelism are good candidates for specialization.

Another interesting case is `occur.pl`. It counts the number of occurrences of an element in a list. Improvement in the sequential execution is 1.21. This improvement increases with the number of processors. Additionally, the specialized program keeps on accelerating up to 108 processors while the non specialized does not speed up after 34 processors.

## 9. RELATED WORK

The possibility of generating different specialized versions for a given predicate in the original program has long been used in partial evaluation [14, 33, 16]. There, program predicates are specialized w.r.t. concrete values (bindings).

Unfortunately, in many cases it is not possible to determine such concrete values at compile-time. However, program analysis can still be used in order to obtain useful information about the behaviour of the program which can then be used to optimize the program. Most analysis-based optimizing compilers only generate monovariant specializations. Clearly, this may miss important specialization opportunities and is in contrast with the fact that most practical analyzers are inherently multivariant. The fact that multivariant analysis can be used in order to generate multiple specialization is already mentioned in Bruynooghe's analysis framework [2]. However, no method is provided to perform such multiple specialization. The relevance of multiple specialization is also foreseen in [39, 59] where some improvements for a few small, hand-coded examples are reported.

Winsborough in [61] presents a powerful multiple specialization framework based on the notion of minimal function graphs [32]. A new abstract interpretation framework is introduced which is tightly coupled with the specialization algorithm. The combination is proved to produce a program with multiple versions of predicates that allow the maximum optimizations possible while having the minimal number of versions for each predicate. However, such multiple specialization was not implemented and no empirical evaluation was performed.

While the analysis framework used by Winsborough is interesting in itself, several generic analysis engines, such as PLAI [48, 45] and GAIA [10], which greatly facilitate construction of abstract interpretation analyzers, are available, well understood, and in comparatively wide use. We believe that it is of practical interest to specify a method for multiple specialization which can be incorporated in a compiler using a minimally modified existing generic analyzer. This was previously attempted in [22], where a simple program transformation technique which has no direct communication with the abstract interpreter is proposed, as well as a simple mechanism for detecting cases in which multiple specialization is profitable. However, this technique is not capable of detecting all the possibilities for specialization or producing a minimally specialized program. It also requires running the interpreter several times after specialization, repeating the analysis–program transformation cycle until a fixpoint is reached.

The specialization framework presented in this paper (and first published in [51]) achieves the same results as those of Winsborough's but with only a slight modification of a standard abstract interpreter and by assuming minimal communication with such interpreter (namely, access to the memoization tables). Our algorithm can be seen as an implementation technique for Winsborough's method in the context of standard analyzers. Also, to the best of our knowledge, the first integration of multiple specialization in a compiler and its experimental evaluation was reported by us in [51].

More recently, an implementation of multiple specialization has also been reported in [34, 35], applied to CLP($\mathcal{R}$). Such multiple specialization framework is simpler than the one we present or that of [61] in that the resulting programs may not be of maximal optimization, i.e., given the existing analysis information and a class of optimizations to be performed, it is possible to build programs more optimized that those achievable by [34, 35]. In spite of this, the results are interesting in that they provide experimental evidence on the relevance of multiple specialization even using a simple strategy.

Another interesting application of multiple specialization has been reported in [37]. There, multiple specialization can be applied in a simple way while obtaining seemingly important improvements over monovariant specialization. The kind of optimizations considered are based on *uninitialized variables*. For them, the expensive general implementation of unification can be replaced by a specialized one which is more efficient. Detection of uninitialized variables is performed by an ad-hoc analysis which does not require fixpoint computation. This allows performing program analysis and transformation simultaneously without losing efficiency. The analysis computes a particular kind of *call modes* for predicates and the transformation generates a different implementation for each call mode. As usual, literals in the final program are renamed to call the correct version. In this application, no minimization step is performed because different call modes always give rise to different optimizations. As a result, implementation of the method is simple, it is efficient, and important improvements are once again obtained w.r.t. monovariant specialization. However, the method is not general and does not seem directly applicable to other kinds of analyses and/or optimizations.

## 10. CONCLUSIONS AND FUTURE WORK

The topic of multiple specialization of logic programs has received considerable theoretical attention and also many of the existing abstract interpreters implement different degrees of multivariance for improving the accuracy of the analysis. This is in contrast with the fact that most existing optimization systems which use analysis information are monovariant. We have proposed a simple framework capable of exploiting the multivariance of analysis in order to obtain multiple specialization without the need for run-time tests for selecting among different versions of a predicate. This framework is potentially capable of generating an expanded version of the program which contains as many versions of a predicate as calling patterns the analysis has considered for it. However, the program is only expanded if such expansion allows further optimizations, thanks to the use of a minimizing algorithm. As in the case of [61], the framework we propose has the two important features of being minimal, i.e., eliminating any of the versions implemented (by collapsing them into other versions) would imply losing some of the optimizations allowed in the expanded program, and of maximal optimization, i.e., no more optimizations are possible by implementing more of the versions generated by analysis. The multiple specialization framework we propose is efficient, as shown by the experimental results, because the core of the process, i.e., the minimization algorithm, does not require the extended program to be materialized. Instead it works with a synthetic representation of the program. It is only after minimization that the program is materialized.

Another important feature of the framework we propose is that there is no restriction on the nature of the optimizations considered and the multiple specialization algorithm is independent from it. However, we have also discussed a relevant class of optimizations: those based on abstract executability. We refer to this combination of multiple specialization and abstract executability as *abstract multiple specialization*.

We argue that our experimental results in the context of a parallelizing compiler are encouraging and show that multiple specialization has a reasonable cost both in compilation time and final program size. Also, the results provide some evidence that the resulting programs can show useful speedups in actual execution time and that thus multiple specialization is indeed a relevant technique in practice.

It remains as future work to extend the presented multiple specialization system in several directions. One of them would be to perform other kinds of optimizations both within program parallelization and beyond this application, including those based on concrete (as opposed to abstract) values, as in traditional partial evaluation. Obviously, the specialization system should be augmented in order to be able to detect and materialize the new optimizations. On-going work in this direction can be found in [54, 49]. Another direction would be to devise and experiment with different minimization criteria: even though the programs generated by the specializer are minimal to allow *all* possible optimizations, it would sometimes be useful to obtain smaller programs even if some of the optimizations are lost.

## ACKNOWLEDGMENTS

This work has been funded in part by ESPRIT project "*DiSCiPl*" 22532, by CICYT project "*ELLA*" TIC96-1012-C02-01, and by M.E.C. grant AP96 50831686. The authors would like to thank Will Winsborough, John Gallagher, and Saumya Debray for useful discussions on multiple specialization, and Francisco Bueno and María García de la Banda for their help during the implementation and experimentation with the tools herein presented. Also, thanks are due to the anonymous referees for useful comments on previous versions of the paper.

## REFERENCES


1. A. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
2. M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
4. M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.
5. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
6. F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *First International Symposium on Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.
7. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.
8. F. Bueno, M. García de la Banda, D. Cabeza, and M. Hermenegildo. The &–Prolog Compiler System — Automatic Parallelization Tools for LP. Technical Report CLIP5/93.0, Computer Science Dept., Technical U. of Madrid (UPM), 1993.
9. M.A. Bulyonkov. Polivariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.
10. 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.
11. J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.
12. M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming*, pages 79–96, Paris, France, June 1991. MIT Press.
13. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.
14. C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 493–501, Charleston, South Carolina, 1993. ACM.
15. 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.

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

17. S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

18. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

19. M. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 724–734. Springer-Verlag, August 1996.

20. J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *1990 International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

21. 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.

22. 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.

23. R. Glueck and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *LNCS*, pages 165–182, Madrid, Spain, 1994. Springer Verlag.

24. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

25. 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.

26. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

27. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

28. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

29. D. Jacobs, A. Langen, and W. Winsborough. Multiple specialization of logic programs with run-time tests. In *1990 International Conference on Logic Programming*, pages 718–731. MIT Press, June 1990.

30. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

31. N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

32. N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Thirteenth Ann. ACM Symp. Principles of Programming Languages*, pages 296–306. St. Petersburg, Florida, ACM, 1986.

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

34. A. Kelly, A. Macdonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An optimizing compiler for CLP(R). In *Proceedings of the Conference on Constraint Programming CP'95*. LNCS, Springer-Verlag, 1995.

35. A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

36. J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

37. T. Lindgren. Polyvariant detection of uninitialized arguments of Prolog predicates. *Journal of Logic Programming*, 28(3):217–229, September 1996.

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

39. A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

40. K. Marriott and H. Sondergaard. Abstract interpretation, 1989. 1989 SLP Tutorial Notes.

41. K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

42. C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

43. A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, Israel, June 1990. MIT Press.

44. K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

45. 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.

46. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

47. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

48. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, 1992.

49. 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.

50. G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.

51. 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.

52. 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.

53. 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.

54. G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*. BRISC, University of Aarhus, Denmark, January 1999.

55. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

56. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

57. 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.

58. P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

59. P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

60. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

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