Implementation of Multiple Specialization in Logic Programs

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

Abstract

We study the multiple specialization of logic programs based on abstract interpretation. This involves in general generating several versions of a program predicate for different uses of such predicate, making use of information obtained from global analysis performed by an abstract interpreter, and finally producing a new, "multiply specialized" program. While the topic of multiple specialization of logic programs has received considerable theoretical attention, it has never been actually incorporated in a compiler and its effects quantified. We perform such a study in the context of a parallelizing compiler and show that it is indeed a relevant technique in practice. Also, we propose an implementation technique which has the same power as the strongest of the previously proposed techniques but requires little or no modification of an existing abstract interpreter.

Keywords: Multiple Program Specialization, Abstract Interpretation, Logic Programming, Compile-time Analysis, Optimization.

1 Introduction

Compilers often use static knowledge regarding invariants in the execution state of the program in order to optimize the program for such particular cases [1]. Standard optimizations of this kind include dead-code elimination, constant propagation, conditional reduction, code hoisting, etc. A good number of optimizations can be seen as special cases of partial evaluation [6, 22]. 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 [24, 23], which has recently been found to be closely related to other techniques used in functional languages such as "driving" [14]. Much work has been done in logic program partial deduction and specialization of logic programs(see e.g. [11, 12, 19]).

PEPM'95 La Jolla, CA USA

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 [7], specialization then being 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 (or a subset, depending on the property being abstracted and the optimizations to be performed) of the concrete values that may appear at a certain program point in all possible program executions. Thus, any optimization allowed in the superset (respectively, subset) 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 any two of its arguments are integers:

```
plus(X,Y,Z):-
     int(X),int(Y),!,Z is X + Y.
plus(X,Y,Z):-
     int(Y),int(Z),!,X is Z - Y.
plus(X,Y,Z):-
     int(X),int(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. Abstract interpretation of logic programs and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [10, 2, 31, 9, 27, 5].

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 two 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" [19, 13, 2, 34] (the counterpart of polyvariant specialization [4]), where different versions of the predicate are generated for each use, so that each one of them is optimized for the particular subset of input values with which each version is to be used. 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 [19, 13, 2, 34]. 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. For example, for the situation in the example above this would result in the following calls and additional 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 the specialized predicate, some intermediate predicates may have to also be specialized even if no optimization is performed for them, with a resulting additional increase in code size. Jacobs et al. [19] 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" specializations of the previous solutions (and thus reduces code size), but 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, which will be discussed in more depth later, 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. Also, there is a close interaction between global analysis and specialization in that in order to optimize each one of the specialized versions global analysis needs to provide information not only for one superset of all the different activations of a predicate, but instead for each one of the different versions that would be generated for each predicate, including in the worst case all the different program points in the bodies of all the clauses in all versions. Thus, in some ways, the actual analysis has to incorporate multiple specialization. Winsborough [34] presents an algorithm, based on the notion of minimal function graphs [21], that solves the two problems outlined above. 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.

While the body of work in the area and Winsborough's fundamental results, both briefly summarized above, are encouraging, there has been little or no evidence to date on the practicality of abstract interpretation driven multiple specialization in logic programs, other than the small improvements for a few small, hand-coded examples reported in [25, 32]. This is in contrast with the fact that abstract interpretation is becoming a practical tool in logic program compilation [18, 32, 31, 33, 3]. The first contribution of this paper is to fill this gap. We report on the implementation of multiple specialization in a parallelizing compiler for Prolog which incorporates an abstract interpretation-based global analyzer and present a performance analysis of multiple specialization in this system. 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.

In doing so, 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 [31, 29] and GAIA [5], 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 [13], 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 second contribution of this paper is to propose an algorithm which 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 (access to the memoization tables). Our algorithm can be seen as an implementation technique for Winsborough's method in the context of standard analyzers. Regarding the problem of version selection, our implementation uses predicate renaming to create paths from calls to specialized predicates. However, we argue that our technique is equally valid in the context of run-time test based clause selection.

The structure of the rest of the paper is as follows. In Section 2 we propose a naive implementation method for multiple specialization. In Section 3 we then present an algorithm for minimizing the number of versions and show that it terminates and is indeed minimal by reasoning over the lattice of transformed programs. In Section 4 we then report on the implementation of the algorithm in a parallelizing compiler, and present and discuss our experimental results. Finally, Section 5 concludes and suggests future work.

2 Building Multiply Specialized Programs based on Abstract Interpretation

The aim of the kind of (goal oriented) program analysis performed by the standard analysis engines used in logic programming is, for a particular description domain, to take a program and a set of initial calling patterns (expressed using elements of such domain) and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the calling patterns. Usual relevant program points are entry to the rule, the point between each two literals, and return from the call.

In essence, the analyzer produces a program analysis graph which can be viewed as a finite representation of the (possibly infinite) set of (possibly infinite) and–or trees explored by the concrete execution [2]. Execution and–or trees which are infinite can be represented finitely through a "widening" [8] 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. The graph has two sorts of nodes: those belonging to rules (also called "and–nodes") and those belonging to atoms (also called "or–nodes"). In order to increase accuracy analyzers usually perform multiple program specialization. This results in several nodes in the and–or graph that correspond to a single program point (in the non-specialized version of the program). Actual analyzers differ in the degree of specialization supported and in the way such specialization is represented, but, in general, most analyzers generate all possible versions since this allows the most accurate analysis [2, 31, 29, 5]. Normally, the results of the analysis are simply "folded back" into the program: information from nodes which correspond to the same points in the original program is "lubbed." The main idea that we will exploit is to instead use the implicit multiply specialized program explored by the analyzer not only to improve analysis information, but also to generate a multiply specialized program in which we have more accurate information for each version and specialize each version accordingly.

In order to perform multiple specialization given the information available at the end of the analysis two problems remain: the first one is devising a method for actually materializing the versions generated taking such information as a starting point and creating the paths connecting each version with its corresponding call point(s). The second one is ensuring that not all possible versions are materialized in the specialized program, but rather only the minimal number necessary to perform all the optimizations which are possible. The first problem is addressed in the remainder of this section and the second in Section 3.

2.1 Analyses with Explicit Construction of the And-Or Graph

As mentioned before, some formulations of top-down abstract interpretation for logic program, such as the original one in Bruynooghe's seminal work [2], are based on explicitly building an abstract version of the resolution tree which contains all possible specialized versions [28, 20]. This has the advantage that, while not directly represented in the abstract and–or graph, it is quite straightforward to derive a fully specialized program (i.e. with all possible versions) from such graph and the original program. Essentially, a new version is generated for a predicate for each or–node present for that predicate in the abstract graph. Thus, the fully specialized program includes a different, uniquely named version of a predicate for each or–node corresponding to this predicate. Different descendent and–nodes represent different calls in the bodies of the clauses of the specialized predicates. Each call in each clause body in the specialized program is replaced with a call to the unique predicate name corresponding to the successor or–node in the graph for each predicate.

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

2.2 Tabulation-based Analyses

For efficiency reasons, most practical analyzers [10, 18, 31, 5, 26] do not explicitly build the analysis graph. Instead, a representation of the information corresponding to each program point is kept in a "memo table." Entries in such memo tables typically contain matched pairs of call and success patterns for that program point. In most systems some of the graph structure is lost and the data available after analysis (essentially, the memo table) is not quite sufficient for connecting each version with its call point(s). This requires a (very minor) modification of the analysis algorithm. For concreteness, we consider here the case of PLAI [31, 29]. In the standard implementation of this analyzer, the memo table essentially contains only entries which correspond to or–nodes in the table. And–nodes are also computed and used, but they are not stored. In the following we will refer to the table entries which correspond to or–nodes as *or–records*. Each or–record contains the following information: predicate to which the or–record belongs, call–pattern, abstract success substitution, and a number identifying the or–record itself. The modification proposed involves the simple addition of one more field to the or–record which contains the *ancestor* information i.e., the point(s) in the multiply specialized program where this or–record (version) is used. By a point we mean a literal within an or–record. It should be noted that the fixpoint algorithm also has to be modified slightly so that this information is correctly stored, but this modification is straightforward. The version to use in each call can then be determined by the ancestor information and no run-time tests are needed to choose among versions.

Example

We will try to clarify the ideas presented with an example. Consider the following program, where the predicate *plus/3* is defined as before and *go/2* is known to be always called with both arguments bound to integers

```
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.
```

After analysis the memo table contains the following or–records:

predicate	id	ancestors
go/2	1	{(query,1)}
p/3	2	{(go/2/1/1,1)}
p/3	4	{(go/2/1/2,1)}
plus/3	3	{(p/3/1/1,2)}
plus/3	5	{(p/3/1/1,4)}

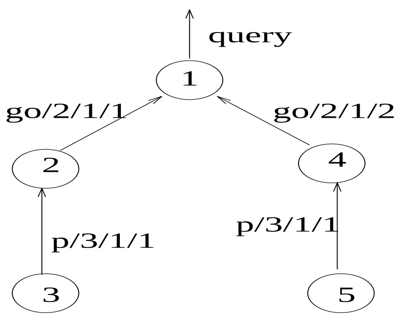

Figure 1: Ancestor information for the example

(only fields relevant to our purposes are shown.) The first field is the predicate to which the or–record belongs, the second is the number that identifies the or–record, and the third is the ancestor information. It is a list of pairs (literal, or–record). A literal is identified with 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*.

In Figure 1 the ancestor information for each or–record is shown graphically. Each or–record is represented by its identifier. It is clear that the ancestor information can be interpreted as backward pointers. These pointers can be followed to determine the version to use in the multiply specialized program. The special literal `query` indicates the starting point of the top-down analysis. PLAI admits any number of starting (entry) points. They are identified by the second number of the pair (query,N).

3 Minimizing the Number of Versions

The number of versions in the multiply specialized program introduced in Section 2 does not depend on the possible optimizations but rather on the number of versions generated during analysis. Even if no benefit is obtained, the program may have several versions of predicates. In this section we address the issue of finding the minimal program that allows the same set of optimizations. In order to do that we collapse into the same version those or–records that are equivalent.[1] This way the set of or–records for each predicate is partitioned into equivalence classes. We now provide an informal description of an algorithm for finding such a program, followed by a more formal description and an algebraic interpretation of the algorithm. The section ends with an example illustrating the execution of the algorithm.

3.1 Informal Description of the algorithm

As mentioned before, the purpose of this algorithm is to minimize the number of versions needed of each predicate while maintaining all the possible optimizations. After analysis and prior to the execution of the algorithm, we compute the optimizations that would be allowed in each version if we implemented the program introduced in Section 2 (i.e., the one which introduces one version of a predicate for each or–record). These optimizations are represented as finite sets which are associated with the corresponding or–record. The algorithm receives as input the set of table entries (or–records) computed during analysis, augmented with the set of optimizations allowed in each or–record. The output of the algorithm is a partition of the or–records for each predicate into equivalence classes. This information together with the original program is enough to build the final program. For each predicate in the original program as many copies are generated as equivalence classes exist for it. Each of these copies (implementations) receives a unique name. Then, the predicate symbols of the calls to predicates with multiple versions are replaced with the predicate symbols of the corresponding version. This is done using the ancestor information. At the same time, some optimizations can take place in each specialized version.

Not all the information in the or–records is necessary for this algorithm. The identifier of the or–record, the ancestor information, and the set of optimizations are enough.

Note that two or–records that allow the same set of optimizations cannot be blindly collapsed since they may use for the same literal (program point) versions of predicates with different optimizations, and thus these two or–records must be kept separate if all possible optimizations are to be maintained. This is why the algorithm consists of two phases. In the first one all the or–records for the same predicate that allow the same set of optimizations are joined in a single version. In the second phase those that use different versions of a predicate for the same literal are split into different versions. Note that each time versions are split it is possible that other versions may also need to be split. This process goes on until no more splitting is needed (a fixpoint is reached). The process always terminates as the number of versions for a predicate is bounded by the number of times the predicate has been analyzed with a different call pattern. Thus, in the worst case we will have as many versions for a predicate as or–records have been generated by the analysis.

3.2 Formalization of the algorithm

In this section some notation is first introduced and then the algorithm and the operations involved are formalized based on such definitions. Termination of the algorithm is discussed in Section 3.3. In the following definitions a program is a set of predicates, a predicate a set of versions, and a version a set of or–records. The algorithm is independent of the kind of optimizations being performed. Thus, no definition of optimization is presented. Instead, it is left open for each particular implementation. However, this algorithm requires sets of optimizations for different or–records to be comparable for equality. As an example, in our implementation an optimization is a pair $(literal, value)$, where value is `true`, `fail` or a list of unifications. The optimization will be materialized in the final program using source to source transformations.

Definition 1 (Or–record) *An* or–record *is a triple* $o = < N, P, S >$ *where* N *is a natural number that identifies the or–record,* P *is a set of pairs (literal, number of or–record) and* S *a set of optimizations.*

Definition 2 (Set of Or–records of a Predicate) *The* set of or–records *of a predicate* Pred*, denoted by* $\mathcal{O}_{Pred}$*, is the set of all the or–records the analyzer has generated for* $Pred$*.*

[1] The equivalence relation will be presented more formally in Section 3.2.

Definition 3 (Version of a Predicate) *Given a predicate* Pred v *is a version of* Pred *if* $v \subseteq \mathcal{O}_{Pred}$.

Definition 4 (Well Defined Set of Versions) *Let* $\mathcal{O}_{Pred}$ *be the set of or–records of* Pred *and let* $\mathcal{V}_{Pred} = \{v_i, i = 1, \ldots, n\}$ *be the set of versions for* Pred. $\mathcal{V}_{Pred}$ *is a* well defined set of versions *if*

$$\bigcup_{i=1}^{n} v_i = \mathcal{O}_{Pred} \text{ and } v_i \bigcap v_j = \emptyset \ i \neq j$$

i.e. $\mathcal{V}_{Pred}$ *is a partition of* $\mathcal{O}_{Pred}$.

Definition 5 (Feasible Version) *A version* $v \in \mathcal{V}_{Pred}$ *is* feasible *if it does not use two different versions for the same literal, i.e. if*

$$\forall o_i, o_j \in v \ \forall literal \in Pred : \exists \mathcal{V}_{Pred'}(\\ (\exists v_k \in \mathcal{V}_{Pred'} \exists o_l = (N_l, P_l, S_l) \in v_k | (literal, N_i) \in P_l) \ \wedge \\ (\exists v_m \in \mathcal{V}_{Pred'} \exists o_n = (N_n, P_n, S_n) \in v_m | (literal, N_j) \in P_n)) \\ \Longrightarrow k = m$$

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. A set of versions is feasible if all the versions in it are feasible.

Definition 6 (Equivalent Or–records) *Two or–records* $o_i = (N_i, P_i, S_i), o_j = (N_j, P_j, S_j) \in \mathcal{O}_{Pred}$, *are* equivalent, *denote by* $o_i \equiv_v o_j$, *if*

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

Definition 7 (Minimal set of Versions) *A set of versions* $\mathcal{V}_{Pred}$ *is minimal if* $\forall o_i, o_j \in \mathcal{O}_{Pred}$

$$o_i \equiv_v o_j \Longrightarrow \exists v_k \in \mathcal{V}_{Pred} \text{ such that } o_i, o_j \in v_k$$

Definition 8 (Version of Maximal Optimization) *A version* v *is of* maximal optimization *if*

$$\forall o_i = (N_i, P_i, S_i), o_j = (N_j, P_j, S_j) \in v \ \ S_i = S_j$$

(all the or–records in the version allow the same optimizations). A set of versions is of maximal optimization *if all the versions in it are of maximal optimization.*

Definition 9 (Optimal Set of versions) *A set of versions* $\mathcal{V}_{Pred}$ *for a predicate* Pred *is* optimal *if it is minimal, of maximal optimization, and feasible, i.e. if*

$$\forall o_i, o_j \in \mathcal{O}_{Pred} : \exists v_k \in \mathcal{V}_{Pred}(o_i, o_j \in v_k) \Leftrightarrow o_i \equiv_v o_j$$

We extend these definitions in the obvious way. For example, we say that a program is minimal if the set of versions for all the predicates in the program are minimal.

According to these definitions, the program before multiple specialization is well defined, feasible, and minimal, but not of maximal optimization in general.

Definition 10 ($Program_0$) *For each predicate* Pred *let* $\mathcal{V}_{Pred} = \{v_i | v_i = \{o_i\}\}$. *We call this program* $Program_0$

Clearly $Program_0$, which assigns a different version to each or–record, is feasible, of maximal optimization, and well defined. This is the program constructed in Section 2.

Definition 11 (Reunion of Versions) *Given two versions* $v_i, v_j \in \mathcal{V}_{Pred}$ *the* reunion *of* v_i *and* v_j, *denoted by* $v_i +_v v_j$, *is*

- $v_i \bigcup v_j$ *if* $\forall o_k, o_l \in (v_i \bigcup v_j) \ S_k = S_l$
- v_i, v_j *otherwise*

The new set of versions $\mathcal{V}_{Pred'}$ *is* $\mathcal{V}_{Pred} - \{v_i, v_j\} \bigcup \{v_i +_v v_j\}$

It is easy to see that if we apply the reunion of versions to programs that are well defined and of maximal optimization the resulting programs are also well defined and of maximal optimization.

Definition 12 ($Program_i$) $Program_i$ *is the program obtained from* $Program_0$ *by reunion of versions when no more reunions are possible (a fixpoint is reached).*

The fixpoint obtained by applying reunion of versions to $Program_0$ is unique and 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.

$Program_i$ is well defined, of maximal optimization (like $Program_0$) and minimal. However, it is not feasible in general. This is the purpose of phase 2 of the algorithm.

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

Definition 13 (Restriction) *Given a set of or–records* $\mathcal{O}_{Pred}$, $\mathcal{R}$ *is a restriction over* $\mathcal{O}_{Pred}$ *if* $\mathcal{R}$ *is a partition of* $\mathcal{N}$ *and* $\mathcal{N} \subseteq \mathcal{O}_{Pred}$.

Definition 14 (Restriction from a Predicate to a Goal) *Let* $\mathcal{V}_{Pred} = \{v_1, v_2, \cdots, v_i, \cdots, v_n\}$ *be a set versions of the predicate* Pred, *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* $\{N | \exists o = (N, P, S) \in v_i \text{ such that } (lit, N) \in P\}$[2]

Definition 15 (A Restriction Holds) *A restriction* $\mathcal{R}$ *holds in a version* v *if*

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

Definition 16 (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* $v \, x_v \, \mathcal{R}$ *and is*

- v *if* v *holds the restriction* $\mathcal{R}$
- v_1, v_2 *where* $v_1 = \{o = (N, P, S) | o \in v \wedge N \in r_k\}$ *and* $v_2 = v - v_1$ *if* v *does not hold* $\mathcal{R}$, *i.e.* $N_i \in r_k \wedge N_j \in r_l \wedge k \neq l$

After splitting a version $v \in \mathcal{V}_{Pred}$ *by a restriction, the new set of versions is*

$$\mathcal{V}_{Pred'} = \mathcal{V}_{Pred} - \{v\} \bigcup \{v \, x_v \, \mathcal{R}\}$$

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

E.g., $\{1,2,3,5\}x_v\{\{1\},\{2,3,4\},\{5\}\} = \{1\},\{2,3,5\}$, but in $\{2,3,5\}$ the restriction does not hold yet. $\{2,3,5\}x_v\{\{1\},\{2,3,4\},\{5\}\} = \{2,3\},\{5\}$. Now the restriction holds. Thus, the initial version is split into 3 versions: $\{1\},\{2,3\},\{5\}$.

The programs obtained by applying splitting of versions by restrictions to programs that are well defined, of maximal optimization, and minimal are also well defined, of maximal optimization and minimal.

Phase 2 of the algorithm terminates. In the worst case we will finish with $Program_0$.

Definition 17 ($Program_f$) *$Program_f$ is the program obtained when all the restrictions hold and no more splitting is needed, i.e., when a fixpoint is reached.*

Theorem 1 (Multiple Specialization Algorithm) *$Program_f$, the result of the multiple specialization algorithm, is optimal.*

By definition of splitting of versions $Program_f$ is well defined, 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.

3.3 Structure of the Set of Programs and Termination

As shown above, given a multiply specialized program generated by the analyzer, several different (but equivalent) programs may be obtained. They may differ in size, optimizations, and even feasibility. In this section we discuss the structure of this set of programs and the relations among its elements.

As $Program_0$ is *well defined* and due to the definitions of reunion and splitting, all the intermediate programs and the final program are well defined. This means that ill-defined programs are not of interest to us and the set of well defined programs is closed under reunion and splitting of versions. This set of well defined programs forms a complete lattice under the $\subseteq$ operation. The $\perp$ element of such a lattice is given by the program in which each or–record is in a different version ($Program_0$). This is the program with the greatest number of versions. The $\top$ element is the program in which all the or–records that correspond to the same predicate are in the same version. This program is the one with the minimum number of versions and is the one obtained when no multiple specialization is done. We move up in the lattice by applying the reunion operation and down with the splitting operation.

Note that not all the programs in the lattice are feasible. A program is not feasible when two or–records in the same version use at the same program point or–records that are in different versions. This is the reason why phase 2 of the algorithm is required. This phase ends as soon as a program is reached that is feasible.

Although not formally stated, the two different operations used during the multiple specialization algorithm (namely reunion and splitting) are operators defined on this lattice since they receive a program as input and produce another program as output. Phase 1 starts with $\perp$ and repeatedly applies $operator_1$ (reunion) moving up in the lattice until we reach a fixpoint. Since the lattice is finite and $operator_1$ is monotonic the termination of phase 1 is guaranteed.

Phase 2 starts with the program that is a fixpoint of $operator_1$ ($Program_i$) and moves down in the lattice. During phase 2 using $operator_2$ (splitting) we move from an infeasible program to (a less) infeasible program, until we reach a feasible program (which will be the fixpoint). $operator_2$ is also monotonic and thus phase 2 also terminates.

3.4 Example

We now apply the minimizing algorithm to the example program in Section 2.2. As was mentioned before, the algorithm also needs to know the set of possible optimizations in each or–record. We will add this information to the or–record registers. The input to the algorithm is as follows:

Pred	id	ancestors	optimizations
go/2	1	{(query,1)}	$\emptyset$
p/3	2	{(go/2/1/1,1)}	$\emptyset$
p/3	4	{(go/2/1/2,1)}	$\emptyset$
plus/3	3	{(p/3/1/1,2)}	{(plus/3/3/2,fail), (plus/3/2/1,true), (plus/3/1/2,true)}
plus/3	5	{(p/3/1/1,4)}	{(plus/3/3/2,true), (plus/3/2/1,fail), (plus/3/1/2,fail)}

We will not go into the details of the set of optimizations, because as mentioned before, the multiple specialization technique presented is independent of the type of optimizations performed. In any case, 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 `int` will take in execution. The only thing to note here is that the set of optimizations is different in these two or–records for plus/3.

Phase 1 starts with each or–record in a different version ($Program_0$). We represent each or–record only by its identifier:

$Program_0$:

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

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 number 2 must be in a different version than or–record number 4. The restriction does not hold and thus $\{2,4\}x_v\{\{2\},\{4\}\} = \{2\},\{4\}$. Now we must check if this splitting has introduced new 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:

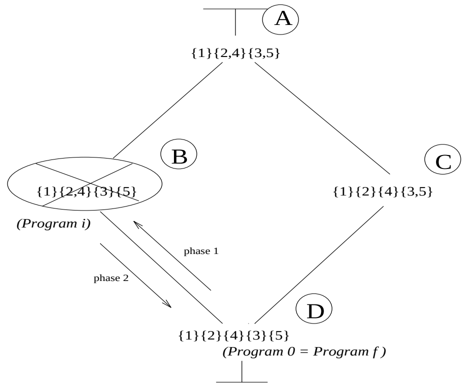

Figure 2: Lattice for the example program

$Program_f$:

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

The final program generated in our implementation of the multiple specializer is the following:

```
go(A,B) :-
        'p/3/$sp/1'(A,B,_), 'p/3/$sp/2'(A,_,B).

'p/3/$sp/1'(X,Y,Z) :-
        'plus/3/$sp/1'(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.
'p/3/$sp/2'(X,Y,Z) :-
        'plus/3/$sp/2'(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.

'plus/3/$sp/1'(X,Y,Z) :-
        Z is X+Y.
'plus/3/$sp/2'(X,Y,Z) :-
        Y is Z-X.
```

Each multiply specialized version receives a unique name (predicate/arity, the string `/$sp/` being used to avoid colliding with user-defined names, and the version number).

Figure 2 shows the lattice for the example program. The node marked with a cross (B) is infeasible. That is why during phase 2 we move down in the lattice and return to $program_0$.

We can use Figure 2 to illustrate the definitions introduced in Section 3.2. 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, thus all the programs are minimal.

4 Experimental Results

In this section we present a series of experimental results. Our aim is to study 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. In particular, we have implemented the specialization method presented in the previous sections in the context of the &-Prolog parallelizing compiler [15, 31, 18, 3], where automatic program parallelization, analysis, optimization, and, now, specialization are completely integrated. This required the addition of a specialization module and a slight modification of the analyzer, as described previously. The analysis time overhead resulting from this modification has been measured at 3% on the average, which we argue is quite tolerable. Furthermore, the same modification is used for other purposes, most notably for incremental global analysis [16], and is therefore now in any case a permanent addition to the analyzer. Only one pass of the analyzer is required to generate both the multiply specialized program and to obtain the information needed to determine the optimizations applicable to each version. These optimizations are of the "abstract executability" type [13], where, as mentioned before, certain builtins or even user defined predicates are reduced to `true`, `fail`, or a set of primitives (typically, unifications) based on the information available from abstract interpretation. Such executability is expressed in a system table (which can be extended through a user–defined predicate). There is one such table for each abstract domain supported since different abstract domains provide different information and allow different optimizations.

The particular application studied is automatic program parallelization. Sequential programs are transformed into equivalent ones in which some parts of the program can be executed in parallel. The parallelism generated by the system is among goals which are "independent," a property which ensures several correctness and efficiency results [17] and which has the additional advantage of not requiring locking during unification. However, while independence can sometimes be determined statically by the analyzer [3], in other cases the resulting parallelized programs contain run-time tests and conditionals (which are used to dynamically ensure independence) and which are targets for optimization through specialization. In a specialized program these tests also provide much information to the analyzer which can be used for subsequent optimizations.

We have used a relatively wide range of programs as benchmarks. They are described in more detail in [3] and can be obtained from `http://clip.dia.fi.upm.es`. These benchmarks have been automatically parallelized using the $sharing + freeness$ abstract domain [30] to eliminate unnecessary run-time tests. We study the very interesting situation in which no information is provided to the analyzer regarding the possible input values – i.e. the analysis has to do its job with only the entry points to the programs given in the module declarations as input data. Since as a result of this the analyzer will sometimes have incomplete information, run-time tests will be included in the resulting programs, which are then amenable to multiple specialization.

In order to assess the cost of specialization at compilation

Benchmark	Analysis	Specializ.	Total	%
aiakl	3806	336	4142	8.11
ann	11442	4306	15748	27.34
bid	1309	723	2032	35.58
boyer	5792	840	6632	12.67
browse	1069	600	1669	35.95
deriv	2032	430	2462	17.47
hanoiapp	806	143	949	15.07
mmatrix	673	233	906	25.72
occur	915	236	1151	20.50
peephole	8399	1173	9572	12.25
progeom	276	143	419	34.13
qplan	2338	1813	4151	43.68
query	239	153	392	39.03
read	37639	2290	39929	5.74
serialize	656	186	842	22.09
warplan	15932	2907	18839	15.43
zebra	4586	543	5129	10.59
			Average	14.84

Table 1: Multiple Specialization Times

time in Table 1 we compare the analysis and specialization time. We argue that it is reasonable to compare these times as the programs that accomplish those tasks are both coded in Prolog and work with the same input program. The specialization time includes computing the possible optimizations in each or–record, minimizing the number of versions, and materializing the new program in which the new versions are optimized (using source to source transformations). It is also important to note that our multiple specialization algorithm requires an analysis. Thus, in principle the total time needed would be the sum of both times. However, as mentioned above, during the automatic parallelization process, an analysis is generally done to optimize the run-time tests. This first analysis can in fact be reused for the multiple specialization with a few modifications [16]. For each benchmark program we present the analysis time, the multiple specialization time, their sum, and the percentage of the total time used in specialization. All the times are in milliseconds and have been measured on a SPARC 10.

We argue that the time required for multiple specialization, at least in this application, is reasonable. However, a potentially greater concern in multiple specialization than compilation time is the increase in program size. Table 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 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 specialized version, i.e. to make the program feasible. 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. **Max(%)** is computed as $\frac{Max}{Preds} \times 100$. **Min(%)** and **Ind(%)** are computed similarly. 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 2 show two different averages. The first is computed considering all the benchmark programs and the second considering only the programs in which the specialization method has obtained some optimization (**Min**> 0).

According to the global average, the specialized program has 43% additional versions with respect to the original program. However, this average greatly depends on the amount of possible optimizations the original program has (in our case run-time tests) and cannot be taken as a general result. Of much more relevance are the ratios between **Max(%)** and **Min(%)**, and between **Ind(%)** and **Min(%)**, which are in some ways independent of the number of possible optimizations in the program. This is supported by the relative independence of the ratios from the benchmarks. 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. We argue that this can in many cases be an acceptable cost in return for no run-time overhead in version selection.

Benchmark	Preds	Max	Min	Ind	Max(%)	Min(%)	Ind(%)	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
peephole	27	31	11	6	114	40	22	1.53
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
Global Average					147	43	24	1.73
Relative Average					208	80	33	1.72

Table 2: Number of Versions

P	**mmatrix.pl**			**deriv.pl**			**occur.pl**		
	std	spec	imp(%)	std	spec	imp(%)	std	spec	imp(%)
1	31800	11549	175.35	759	715	6.15	690	665	3.76
2	16309	6500	150.91	420	399	5.26	458	385	18.96
3	11200	4579	144.59	305	289	5.54	330	283	16.61
4	8819	3555	148.07	250	235	6.38	276	234	17.95
5	7235	2930	146.93	211	202	4.46	225	200	12.50
6	5845	2495	134.27	190	182	4.40	210	179	17.32
7	5069	2200	130.41	172	166	3.61	203	174	16.67
8	4750	1980	139.90	162	156	3.85	203	167	21.56
9	4075	1820	123.90	151	145	4.14	203	158	28.48

Table 3: Run-time performance

Having briefly addressed the cost (in time and size) of multiple specialization, we now study the actual benefits obtained. In order to do so we report on the execution of a representative subset of the parallelized programs, with and without multiple specialization on a 10 processor Sequent Symmetry and compare their performance.[3] The results are shown in Table 3. All times are again in milliseconds.

The first benchmark program, `mmatrix.pl`, is a program for matrix multiplication. It is a good candidate for parallelization and its execution time decreases nearly linearly with the number of processors. If the user provides enough information regarding the input this program can be parallelized without any run-time tests. However, if no information is provided by the user (the case studied) such tests are generated and performance decreases. In this arguably interesting case from the practical point of view the improvement obtained with multiple specialization is quite high, ranging from 175.35% with one processor to 126.32% with ten processors, i.e. the specialized program runs more than twice as fast as the original program. This is because it is a recursive program in which specialization automatically detects and extracts an invariant (see [13]): that once a certain run-time test has succeeded it does not need to be checked in the following recursive calls.

`deriv.pl` is a program for symbolic differentiation and also a good candidate for parallelization. However, the improvement obtained with specialization is not very high (around 5%). This shows that not all programs with significant parallelism are good candidates for specialization.

The last benchmark program we present is `occur.pl`. It counts the number of occurrences of an element in a list. Improvement in the sequential execution is low. However, it increases when more processors are involved. It is also important to note that the program before multiple specialization gets no speedup from 7 to 9 processors, while the multiply specialized program keeps on speeding up in that range.

[3] Note that these times are not comparable with the previous ones since the Sequent is a slower machine sequentially than SPARC 10.

5 Conclusions and Future Work

While the topic of multiple specialization of logic programs has received considerable theoretical attention, it has never been actually incorporated in a compiler and its effects quantified. We perform such a study in the context of a parallelizing compiler and show that it is indeed a relevant technique in practice. Also, we propose an implementation technique which has the same power as the strongest of the previously proposed ones but requires little or no modification of an existing abstract interpreter.

We argue that our experimental results are encouraging and show that multiple specialization has a reasonable cost both in compile-time cost and final program size. Also, the results provide some evidence that the resulting programs can show considerable benefits in actual execution time for the application studied. As future work we plan to investigate reducing program size by using run-time test based selection of specialized predicates. However, it also remains to be studied whether this is more profitable when execution time is also taken into account. We also plan on extending our studies to other forms of optimization in program parallelization and also to optimizations beyond this application.

Acknowledgments

This work was funded in part by ESPRIT project 6707 "*ParForCE*" and by CICYT project TIC93-0737-C02-01. The Sequent Symmetry was acquired thanks to a grant from British Telecom. The authors would also like to thank Will Winsborough, Saumya Debray, John Gallagher, Jonathan Martin, and the members of the CLIP group at UPM for their comments on earlier drafts of this paper and help during the implementation and experimentation with the tools herein presented.

References

[1] A. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.

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

[3] 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.

[4] M.A. Bulyonkov. Polivariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

[13] 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*, pages 323–335. Springer-Verlag, 1991.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] K. Marriott and H. Sondergaard. Abstract interpretation, 1989. 1989 SLP Tutorial Notes.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

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

[33] 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.

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