Abstract Specialization and its Application to Program Parallelization

Germán Puebla and *Manuel Hermenegildo*

{german,herme}@fi.upm.es
Department of Computer Science
Technical University of Madrid (UPM)

Abstract. 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. This paper reports on the application of abstract multiple specialization to automatic program parallelization in the &-Prolog compiler. Abstract executability, the main concept underlying abstract specialization, is formalized, the design of the specialization system presented, and a non-trivial example of specialization in automatic parallelization is given.

1 Introduction

A good number of compiler optimizations can be seen as special cases of partial evaluation [CD93,JGS93]. 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. Much work has been done in partial evaluation (deduction) and specialization of logic programs (see e.g. [LS91,Kom92,GB90,GCS88,JLW90]). 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 [CC77], specialization then being with respect to abstract values (substitutions), rather than concrete ones.

A procedure may have 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 program *multiple specialization* [JLW90,GH91,Bru91,Win92] (the counterpart of polyvariant specialization [Bul84]), where a different version of a predicate is 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.

In [PH95] we presented a framework for abstract multiple specialization. In order to reduce the size of the resulting program as much as possible, the framework incorporates a minimization algorithm that associates a set of optimizations to each of the multiple versions of a procedure generated during multi-variant abstract interpretation. This framework achieves the same results as those of Winsborough's [Win92] but with only a slight modification of a standard abstract interpreter. We argue that the experimental results given in [PH95] showed that multiple specialization is indeed practical and useful in automatic parallelization, and also that such results shed some light on its possible practicality in other applications. However, due to space limitations, the nature of the optimizations, the procedure used to detect them, and the actual source to source transformations by which optimizations are materialized, were not presented. This work tries to fill this gap by describing the application of abstract multiple specialization to automatic program parallelization in the &-Prolog compiler. *Abstract executability*, the main concept underlying abstract specialization, is formalized, the design of the specialization system presented, and a non-trivial example of specialization in automatic parallelization is given.

The structure of the paper is as follows. Section 2 briefly recalls abstract interpretation. In Section 3 the notion of abstract executability is formalized. Then Section 4 presents a particular application of abstract specialization. Section 5 shows the design of the abstract specializer and an example of specialized program. Finally, Section 6 concludes.

2 Abstract Interpretation of Logic Programs

Abstract interpretation [CC77] is a useful technique for performing global analysis of a program in order to compute at compile-time characteristics of the terms to which the variables in that program will be bound at run-time. 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.

In abstract interpretation a program is executed using *abstract substitutions* (λ) instead of actual substitutions (θ). An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain (D). The set of all possible terms that a variable can be bound to in an abstract substitution represents an *abstract domain* (D_α) which is usually a complete lattice or cpo which is ascending chain finite.

Abstract substitutions and sets of concrete substitutions are related via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. The usual definition for partial order ($\sqsubseteq$) over abstract domains, is $\forall \lambda, \lambda' \in D_\alpha \quad \lambda \sqsubseteq \lambda'$ 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 its corresponding abstract operation u' as follows: for every x in the concrete computational domain, $u(x) \subseteq \gamma(u'(\alpha(x)))$.

We now introduce some notation. A *program* is a sequence of *clauses.* Clauses are of the form $h \leftarrow b$, where h is an atom and b is a possibly empty conjunction of literals. Clauses in the program are written with a unique subscript attached to the head atom (the clause number), and dual subscript (clause number, body position) attached to each literal in the body atom e.g. $H_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$ where $B_{k,i}$ is a subscripted literal. The clause may also be referred to as clause k, the subscript of the head atom, and each literal in the program is uniquely identified by its subscript k, i.

The goal of the abstract interpreter is, for a given abstract domain, to annotate the program with information about the current environment (i.e. the values of variables), at each program point. Correctness of the analysis requires that annotations be valid for any call (program execution). Different names distinguish abstract substitutions depending on the point in a clause to which they correspond. In particular, we will be interested in the *abstract call substitution* $\lambda_{k,i}$ for each literal $L_{k,i}$ which is the abstract substitution just before calling the literal $L_{k,i}$.

If the analysis is 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. 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 [BCHP96]. Information inferred by goal oriented analysis may be more accurate as each $\lambda_{k,i}$ "only" has to be valid when executing calls described by the calling patterns.

3 Abstract Execution

The concept of *abstract executability* was, to our knowledge, first introduced informally in [GH91]. 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 introduce in the following an improved formalization of abstract executability. The set of variables in a literal L is represented as $var(L)$. The restriction of the substitution θ to the variables in 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 θ is a substitution such that the value of each variable in $var(L)$ is the same in the environment e and the substitution θ.

Definition 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*

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

$RT(L,P)$ is not computable in general. The set of run-time environments for a literal is not known at compile-time. However, it is sometimes possible to find a set of bindings which will appear in *any* environment for L. These "invariants" can be synthesized in a substitution θ_s such that $\forall\theta \in RT(L,P)\ \exists\theta_d : L\theta = L\theta_s\theta_d$. Note that it is always possible to find a trivial $\theta_s = \epsilon$, the empty substitution, which corresponds to having no static knowledge of the run-time environment. In this case, we can simply take $\theta_d = \theta$ for any θ.

The substitutions θ_s and θ_d correspond to the so-called *static* and *dynamic* values respectively in partial evaluation [JGS93]. As a result, we can specialize L for the statically known data θ_s. Specialization is then usually performed by *unfolding* $L\theta_s$. If all the leaves in the SLD tree for $L\theta_s$ are failing nodes and $L\theta_s$ is pure (i.e., its execution does not produce side-effects), then the literal L can be replaced by $false$. If all the leaves are failing nodes except for one which is a success node and $L\theta_s$ is pure then L can be replaced by a set of unifications on $var(L\theta_s)$ which have the same effect as actually executing $L\theta_s\theta_d$ in P. If such set of unifications is empty, L can be replaced by $true$.

The goal of abstract specialization is also to replace a literal by $false$, $true$ or a set of unifications, but rather than starting from $RT(L,P)$ it will use information on $RT(L,P)$ provided by abstract interpretation, i.e., the abstract call substitution for L. For simplicity, we will restrict our discussion to replacing L with $false$ or $true$.

Definition 2 (Trivial Success Set). *Given a literal L from a program P we define the* trivial success set *of L in P as*

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

Definition 3 (Finite Failure Set). *Given a literal L from a program P we define the* finite failure set *of L in P as*

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

Note that if two distinct literals $L_{k,i}$ and $L_{l,j}$ are equal up to renaming then the sets $TS(L_{k,i},P)$ (resp. $FF(L_{k,i},P)$) and $TS(L_{l,j},P)$ (resp. $FF(L_{l,j},P)$) will also be equal up to renaming. However, there is no a priori relation between $RT(L_{k,i},P)$ and $RT(L_{l,j},P)$.

Definition 4 (Elementary Literal Replacement). Elementary Literal Replacement *(ER) of a literal L in a program P is defined as:*

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

Note that given the definitions of $TS(L,P)$ and $FF(L,P)$, any literal L which is not dead code and produces some side-effect (i.e., L is not pure) will not be affected by elementary literal replacement, i.e., $ER(L,P) = L$.

Theorem 1 (Elementary Replacement). *Let P_{ER} be the program obtained by replacing each literal $L_{k,i}$ in P by $ER(L_{k,i},P)$. P and P_{ER} produce the same computed answers and side-effects.*

The idea is to optimize a program by replacing 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, this optimization is indeed very relevant. However, elementary replacement is not directly applicable because $RT(L,P)$, $TS(L,P)$, and $FF(L,P)$ are generally not known at specialization time.

Definition 5 (Abstract Trivial Success Set). *Given an abstract domain D_α we define the* abstract trivial success set *of L in P as*

$$TS_\alpha(L,P,D_\alpha) = \{\lambda \in D_\alpha : \gamma(\lambda) \subseteq TS(L,P)\}$$

Definition 6 (Abstract Finite Failure Set). *Given an abstract domain D_α we define the* abstract finite failure set *of L in P as*

$$FF_\alpha(L,P,D_\alpha) = \{\lambda \in D_\alpha : \gamma(\lambda) \subseteq FF(L,P)\}$$

Note that by using the least upper bound operator ($\sqcup$) of the abstract domain D_α, $TS_\alpha(L,P,D_\alpha)$ and $FF_\alpha(L,P,D_\alpha)$ could be represented by a single abstract substitution (rather than a set of them), say $\lambda_{TS_\alpha(L,P,D_\alpha)} = \sqcup_{TS_\alpha(L,P,D_\alpha)}\lambda$ and $\lambda_{FF_\alpha(L,P,D_\alpha)} = \sqcup_{FF_\alpha(L,P,D_\alpha)}\lambda$. However, this alternative approximation of the actual sets $TS(L,P)$ and $FF(L,P)$ can introduce an important loss of accuracy for some abstract domains because $\gamma(\lambda_{TS_\alpha(L,P,D_\alpha)}) \supseteq \bigcup_{TS_\alpha(L,P,D_\alpha)} \gamma(\lambda)$, thus reducing the optimizations achievable by abstract executability .

Definition 7 (Abstract Execution). Abstract Execution *(AE) of L in P with abstract call substitution $\lambda \in D_\alpha$ is defined as:*

$$AE(L,P,D_\alpha,\lambda) = \begin{cases} true & \text{if } \lambda \in TS_\alpha(L,P,D_\alpha) \\ false & \text{if } \lambda \in FF_\alpha(L,P,D_\alpha) \\ L & \text{otherwise} \end{cases}$$

If $AE(L,P,D_\alpha,\lambda) = true$ (resp. $false$) we will say that L is abstractly executable to *true* (resp. $false$). If $AE(L,P,D_\alpha,\lambda) = L$ then L is not abstractly executable.

Theorem 2 (Abstract Executability). *Let let P_{AE} be the program obtained by replacing each literal $L_{k,i}$ in P by $AE(L_{k,i},P,D_\alpha,\lambda_{k,i})$. P and P_{AE} produce the same computed answers and side-effects.*

The advantage of abstract executability as given in Definition 7 over elementary replacement is that instead of using $RT(L, P)$ which is not computable in general, such sets are approximated by abstract substitutions which for appropriate abstract domains (and widening mechanisms) will be computable in finite time.

Definition 8 (Optimal $\mathbf{TS}_\alpha$). *An abstract trivial success set $TS_\alpha(L, P, D_\alpha)$ is* optimal *iff*

$$\Big(\bigcup_{\lambda \in TS_\alpha(L,P,D_\alpha)} \gamma(\lambda)\Big) = TS(L, P)$$

Optimal abstract finite failure sets are defined similarly. One first possible disadvantage of abstract execution with respect to elementary replacement is due to the loss of information associated to using an abstract domain instead of the concrete domain. This is related to the expressive power of the abstract domain, i.e. what kind of information it provides. If $TS_\alpha(L, P, D_\alpha)$ and/or $FF_\alpha(L, P, D_\alpha)$ are not optimal then there may exist literals in the program such that $RT(L, P) \subseteq TS(L, P)$ or $RT(L, P) \subseteq FF(L, P)$ and thus elementary replacement could in principle be applied but abstract execution cannot. In general, domains will be optimal for *some* predicates but not all.

Another possible disadvantage is that even if the abstract domain is expressive enough and both $TS_\alpha(L, P, D_\alpha)$ and $FF_\alpha(L, P, D_\alpha)$ are optimal, the computed abstract substitutions may not be accurate enough to allow abstract execution. Therefore, the choice of the domain should be first guided by the predicates whose optimization is of interest so that $TS_\alpha(L, P, D_\alpha)$ and $FF_\alpha(L, P, D_\alpha)$ are as adequate as possible for them, and second by the accuracy of the abstract substitutions it provides and its computational cost.

Definition 9 (Maximal Subset). *Let S be a set and let $\sqsubseteq$ be a partial order over the elements of S. We define the* maximal subset *of S with respect to $\sqsubseteq$ as*

$$M_{\sqsubseteq}(S) = \{s \in S : \nexists s' \in S \ (s \neq s' \wedge s \sqsubseteq s')\}^1$$

Abstract execution as given in Definition 7 is not applicable in general because even though each $\lambda_{k,i}$ is computable by means of abstract interpretation, TS_α and FF_α are not computable in general. Additionally, if D_α is infinite, TS_α and FF_α may also be infinite. However, based on the observation that if $\lambda \in TS_\alpha$ then $\forall \lambda' \sqsubseteq \lambda \ \ \lambda' \in TS_\alpha$, the conditions $\lambda \in TS_\alpha(L, P, D_\alpha)$ and $\lambda \in FF_\alpha(L, P, D_\alpha)$ are equivalent to $\exists \lambda' \in M_{\sqsubseteq}(TS_\alpha(L, P, D_\alpha)) : \lambda \sqsubseteq \lambda'$ and $\exists \lambda' \in M_{\sqsubseteq}(FF_\alpha(L, P, D_\alpha)) : \lambda \sqsubseteq \lambda'$ respectively and thus can be replaced in Definition 7. Unlike TS_α and FF_α, $M_{\sqsubseteq}(TS_\alpha(L, P, D_\alpha))$ and $M_{\sqsubseteq}(FF_\alpha(L, P, D_\alpha))$ are finite for any D_α with finite width. Additionally, they usually have one or just a few elements for most practical domains.

Definition 10 (Base Form). *The* Base Form *of a literal L which calls predicate $Pred$ of arity n (represented as $\overline{L}$) is the literal $Pred(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct free variables.*

1 $s \neq s' \wedge s \sqsubseteq s'$ may also be written as $s \sqsubset s'$.

As the number of literals in a program that call a given predicate is not bounded and in order to reduce the number of TS_α and FF_α sets that need to be computed to optimize a program, in what follows we will only consider one TS_α and FF_α per predicate which refers to its base form.

The function named *call_to_entry*, which is normally defined for each domain in most abstract interpretation frameworks, will be used to relate an abstract substitution over the variables of an arbitrary literal with the base form of the literal. The format of this function is $\mathit{call\_to\_entry}(L1, L2, D_\alpha, \lambda)$. Given a literal $L1$ and an abstract substitution $\lambda \in D_\alpha$ over the variables in $L1$, this function computes an abstract substitution over the variables in $L2$ which is the result of unifying $L1$ and $L2$ both with respect to concrete and abstract substitutions.

Using the base form and *call_to_entry* the conditions $\lambda \in TS_\alpha(L, P, D_\alpha)$ and $\lambda \in FF_\alpha(L, P, D_\alpha)$ in Definition 7 can be replaced by $\mathit{call\_to\_entry}(L, \overline{L}, D_\alpha, \lambda) \in TS_\alpha(\overline{L}, P, D_\alpha)$ and $\mathit{call\_to\_entry}(L, \overline{L}, D_\alpha, \lambda) \in FF_\alpha(\overline{L}, P, D_\alpha)$ respectively. The transformed conditions are not equivalent, but are sufficient. This means that correctness is guaranteed, but possibly some optimizations will be lost.

3.1 Optimization of Calls to 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, i.e., $\forall P, P'\ TS_\alpha(\overline{B}, P, D_\alpha) = TS_\alpha(\overline{B}, P', D_\alpha) = TS_\alpha(\overline{B}, D_\alpha)$. As a result, we can compute $TS_\alpha(\overline{B}, D_\alpha)$ and $FF_\alpha(\overline{B}, D_\alpha)$ once and for all for each builtin predicate B and they will be applicable to all literals that call the builtin predicate in any program.

Definition 11 (Operational Abstract Execution of Builtins). Operational abstract execution *(OAEB) of a literal L with abstract call substitution λ that calls a builtin predicate B is defined as:*

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

$A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ are approximations of $M_\sqsubseteq(TS_\alpha(\overline{B}, D_\alpha))$ and $M_\sqsubseteq(FF_\alpha(\overline{B}, D_\alpha))$ respectively. This is because there is no automated method that we are aware of to compute $M_\sqsubseteq(TS_\alpha(\overline{B}, D_\alpha))$ and $M_\sqsubseteq(FF_\alpha(\overline{B}, D_\alpha))$ for each builtin predicate $\overline{B}$. For soundness it is required that both $A_{TS}(\overline{B}, D_\alpha) \subseteq TS_\alpha(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha) \subseteq FF_\alpha(\overline{B}, D_\alpha)$. We believe that a good knowledge of D_α allows finding safe approximations, and that in many cases it is easy to find the best possible approximations $A_{TS}(\overline{B}, D_\alpha) = M_\sqsubseteq(TS_\alpha(\overline{B}, D_\alpha))$ and $A_{FF}(\overline{B}, D_\alpha) = M_\sqsubseteq(FF_\alpha(\overline{B}, D_\alpha))$.

Additionally, the condition $\mathit{call\_to\_entry}(L, \overline{B}, D_\alpha, \lambda) \sqsubseteq \lambda'$ has been replaced by the equivalent one $\mathit{call\_to\_entry}(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda'$, where $\sqcup$ stands for the *least upper bound*, which can be computed effectively.

Theorem 3 (Operational Abstract Executability of Builtins). *Let P be a program and let P_{OAEB} be the program obtained by replacing each literal $L_{k,i}$ in P by $OAEB(L_{k,i}, \lambda_{k,i})$ where $\lambda_{k,i}$ is the abstract call substitution for $L_{k,i}$. If $\forall B\ A_{TS}(\overline{B}, D_\alpha) \subseteq TS_\alpha(\overline{B}, D_\alpha) \wedge A_{FF}(\overline{B}, D_\alpha) \subseteq FF_\alpha(\overline{B}, D_\alpha)$ then P_{OAEB} is computable in finite time, and both P and P_{OAEB} produce the same computed answers and side-effects.*

Example 1 Consider an abstract domain D_α 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. 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$\}$ and its abstract version $TS_\alpha(ground(X_1), D_\alpha) = \{int, float, bottom\}$, which is clearly not optimal (there are many ground terms which are neither integers nor floating numbers). We can take $A_{TS}(ground(X_1), D_\alpha) = \{int, float\}$ $= M_{\sqsubseteq}(\{int, float, bottom\})$. Consider the following clause containing the literal $ground(X)$:

$$p(X, Y) \leftarrow q(Y), ground(X), r(X, Y).$$

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 $A_{FF}(ground(X_1), D_\alpha) = \{free\} = M_{\sqsubseteq}(FF_\alpha(ground(X_1), D_\alpha))$ which again is not optimal.

4 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 [Con83,CC94]: or-parallelism and and–parallelism. And-parallelism refers to the parallel execution of the goals 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, [CC94] and it 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 [Con83,HR95]. Thus, the dependencies among

```
:-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.
```

Fig. 1. mmatrix.pl

the different goals must be determined, and there is a related parallelization overhead involved. It is vital that such overhead remain reasonable. Herein we follow the approach proposed initially by R. Warren et al [WHD88,HWD92] (see their references for alternative approaches) which combines local analysis and run–time checking with a data-flow analysis based on abstract interpretation [CC77]. This combination of techniques has been shown to be quite useful in practice [WHD88,MJMB89,RD90,Tay90,dMSC93].

4.1 The Annotation Process and Run-time Tests

In the &-Prolog system, the automatic parallelization process is performed as follows [BGH94a]. Firstly, if required by the user, the Prolog program is analyzed using one or more global analyzers, aimed at inferring useful information for detecting independence. Secondly, since side–effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [MH89] 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

```
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)).
```

Fig. 2. Parallel mmatrix

minimum 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 [BGH94a], 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.

4.2 An Example: Matrix Multiplication

We illustrate the process of automatic program parallelization with an example. Figure 1 shows the code of a Prolog program for matrix multiplication. The declaration `:-module(mmatrix,[mmultiply/3]).` is used by the (goal-oriented) analyzer to determine that the only predicate which may appear in top-level queries is *mmatrix/3*. No information is given about the arguments in calls to the predicate *mmatrix/3*. As mentioned before, this could be done using one or more *entry* declarations [BCHP96]. 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.

Figure 2 contains the source program after automatic parallelization. if-then-else's, like in Prolog, are written `(cond -> then ; else)`. Even though programs as defined in Section 2 do not have if-then-else's in the body of clauses, this construct poses no additional theoretical difficulties. The same effect (modulo some run-time overhead) can be achieved using conjunctions of literals and the cut. The `&` signs between goals indicate that they can be executed in parallel. As can be seen, a lot of run-time tests have been introduced. They are used to determine independence at run-time. If the tests hold 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]])`.

Obviously, the tests will cause considerable overhead in run-time performance, to the point of not even knowing at first sight if the parallelized program will run faster than the sequential one. The predicate *vmul/3* does not appear in Figure 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.

5 Abstract Specialization in the &-Prolog Compiler

As stated in Section 4.1, analysis information is used in the &-Prolog system to introduce as few run-time tests as possible in the process of automatic program parallelization. However, analysis information allows more powerful program optimizations than the ones performed during program annotation. First, analysis information can be used to perform abstract specialization to all program points, instead of just those ones in which run-time tests are going to be introduced. Second, the abstract interpretation framework used in &-Prolog (PLAI [MH91,MH92,BGH94b]) is multi-variant. This allows, in principle, the introduction of multiple specialization based on abstract interpretation. To this end, the analysis framework has been augmented in order to distinguish abstract substitutions for the different variants and additional structure information has been added to recover the analysis and-or graph (the *ancestors information* of [PH95]).

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 system uses incremental analysis techniques to efficiently obtain updated analysis information from the one generated for the original program [HMPS95,PH96]. This updated analysis information is then used by the abstract specializer to optimize the program as much as possible.

5.1 Design of the Abstract Multiple Specializer

Conceptually, the process of abstract multiple specialization is composed of five steps, which are shown in Figure 3 (picture on the right), together with the role of abstract specialization in the &-Prolog system (picture on the left).

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 an 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 multi-variant abstract interpretation is used to detect (but not to perform) the optimizations

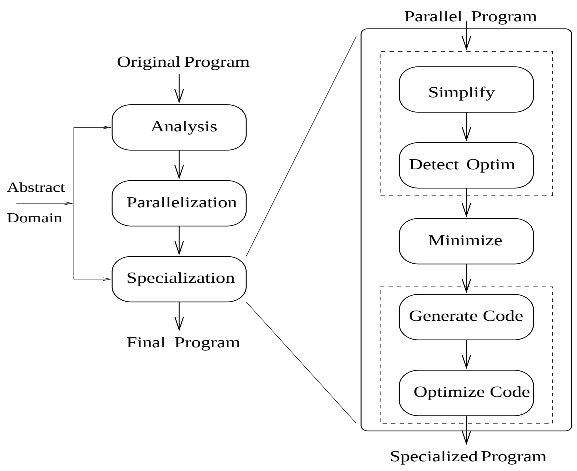

Fig. 3. Program Annotation and Abstract Multiple Specialization

allowed in each of the (possibly) multiple versions generated for each predicate during analysis. 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 lowest upper bound, as is done in the first step. The output of this step is the set of literals that become abstractly executable (and their value) in each version of a predicate due to multiple specialization. Note that these literals are not abstractly executable without multiple specialization, otherwise the optimization would have already been performed in the first step.

The third step (*minimize*) is concerned with reducing the size of the multiply specialized program as much as possible, while maintaining all possible optimizations without the need of introducing run-time tests to select among different versions of a predicate. A detailed presentation of the algorithm used in this step and its evaluation is the subject of [PH95].

In the fourth step (*generate code*) the source code of the minimal multiply specialized program is generated. The result of the minimization algorithm in the previous step indicates the number of implementations needed for each predicate. Each of them 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.

Domain	$TS_\alpha(ground(X_1))$	$FF_\alpha(ground(X_1))$	$TS_\alpha(indep(X_1))$	$FF_\alpha(indep(X_1))$
sharing	O	N	O	N
sh+fr	O	S	O	S
asub	O	N	O	N

Table 1. Optimality of Different Domains

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 3 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*) are independent from the actual optimizations being performed.

5.2 Abstract Domains

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_α), 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 11 with specialized ones because in $\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \mathit{call\_to\_entry}(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda'$ all values are known before specialization time except for λ which will be computed by analysis. I.e., conditions can be partially evaluated with respect to D_α, $\overline{B}$ and a set of λ', as they are known in advance.

5.3 Example

As seen before, in the context of automatic parallelization in the &-Prolog system abstract interpretation is mostly used to eliminate run-time tests necessary to determine independence. These tests are of two types: `ground/1` and `indep/1`. As these builtin predicates are the main target of optimization, the abstract domains used in analysis should be able to provide useful TS_α and FF_α for them. For the three domains these sets are computable and we can take $A_{TS} = M_{\sqsubseteq}(TS_\alpha(\overline{B}, D_\alpha))$ and $A_{FF} = M_{\sqsubseteq}(FF_\alpha(\overline{B}, D_\alpha))$.

Table 1 shows the accuracy of a number of abstract domains (*sharing* [JL92,MH92], *sharing+freeness* (sh+fr) [MH91], and *asub* [Son86,CDY91]) present in the &-Prolog system with respect to the run-time tests. O stands for optimal, S stands for approximate, and N stands for none, i.e. $FF_\alpha(\overline{B}, D_\alpha) = \{\bot\}$. The three of them are optimal for abstractly executing both types of tests to true. However, only sharing+freeness (sh+fr) allows abstractly executing these tests to false, even though not in an optimal way.

```
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)).
```

Fig. 4. Specialized mmatrix

The resulting program after abstract multiple specialization is performed is shown in Figure 4. Two versions have been generated for the predicate *mmultiply/3* and four for the predicate *multiply/3*. They all have unique names and literals have been renamed appropriately to avoid run-time tests. As in Figure 2, the predicate *vmul/3* is not presented in the figure because its code is identical to the one in the original program (and the parallelized program). Only one version has been generated for this predicate even though multi-variant abstract interpretation generated eight different variants for it. As no further optimiza-

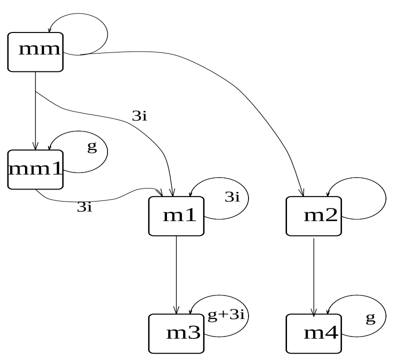

Fig. 5. Call Graph of the Specialized Program

tion 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 [GH91]. Figure 5 shows the call graph of the specialized program. `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.

The specialized version of matrix multiplication obtains speed-ups ranging from 2.75 for one processor to 2.24 with 9 processors with respect to the non-specialized parallel program. The speed-up with respect to the original sequential program is 5.31 for nine processors. The parallelized program without specialization obtains a speed-up of (only) 2.37 with nine processors. Detailed experimental results, including specialization times and size of the resulting specialized programs can be found in [PH95].

6 Conclusions and Future Work

In this paper we have presented the use of abstract (multiple) specialization in the context of a parallelizing compiler and formalized the concept of abstract executability. By means of an example, we have shown the ability of abstract specialization to perform non-trivial optimizations, such as loop-invariant detection, on the parallel code generated.

It remains as future work to improve the abstract specialization system presented in several directions. One of them would be the extension of the abstract specialization framework in order to perform other kinds of optimizations, 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.

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 was funded in part by ESPRIT project 6707 "*ParForCE*" and by CICYT project TIC93-0737-C02-01. Germán Puebla is also funded in part by project C.A.M. AE00157/95. The authors would also like to thank Will Winsborough, Saumya Debray, and John Gallagher 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.

References

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data–flow Analysis of Standard Prolog Programs. In *European Symposium on Programming*, Sweden, April 1996.

[BGH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

[BGH94b] 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.

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

[Bul84] M.A. Bulyonkov. Polivariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

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

[CC94] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[CD93] 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.

[CDY91] 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.

[Con83] 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.

[dMSC93] Vítor Manuel de Morais Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[GB90] 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.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

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

[HMPS95] 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.

[HR95] 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.

[HWD92] 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.

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

[JL92] 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.

[JLW90] 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.

[Kom92] 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.

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

[MH89] 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.

[MH91] 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.

[MH92] 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.

[MJMB89] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *International Conference on Logic Programming*. MIT Press, June 1989.

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

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

[RD90] 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.

[Son86] 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.

[Tay90] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. In *1990 International Conference on Logic Programming*, pages 174–189. MIT Press, June 1990.

[WHD88] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

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