"European Symposium on Programming" 1996. Sweden. Springer-Verlag

Global Analysis of Standard Prolog Programs

F. Bueno D. Cabeza M. Hermenegildo G. Puebla
{bueno,dcabeza,herme,german}@fi.upm.es

Computer Science Department
Technical University of Madrid (UPM)
Campus de Montegancedo, 28660, Boadilla del Monte, Spain

Abstract. Abstract interpretation-based data-flow analysis of logic programs is, at this point, relatively well understood from the point of view of general frameworks and abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical dialect of the Prolog language is attempted, and only few solutions to these problems have been proposed to date. Existing proposals generally restrict in one way or another the classes of programs which can be analyzed. This paper attempts to fill this gap by considering a full dialect of Prolog, essentially the recent ISO standard, pointing out the problems that may arise in the analysis of such a dialect, and proposing a combination of known and novel solutions that together allow the correct analysis of arbitrary programs which use the full power of the language.

Keywords: Logic Programming, Abstract Interpretation, Optimization

1 Introduction

Global program analysis, generally based on abstract interpretation [11], is becoming a practical tool in logic program compilation, in which information about calls, answers, and substitutions at different program points is computed statically [18, 26, 23, 27, 4, 13, 1, 12, 22, 6]. Most proposals to date have concentrated on general frameworks and suitable abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical language is attempted. Such problems relate to dealing correctly with all builtins, including meta-logical, extra-logical, and dynamic predicates (where the program is modified during execution). Often, problems also arise because not all the program code is accessible to the analysis, as is the case for some builtins (meta-calls), some predicates (multifile and/or dynamic), and some programs (multifile or modular).

Implementors of the analyses obviously have to somehow deal with such problems, and some of the implemented analyses provide solutions for some problems. However, the few solutions which have been published to date [26, 14, 18, 23, 7] generally restrict the use of builtin predicates in one way or another (and thus the class of programs which can be analyzed).

This paper attempts to fill this gap. We consider the correct analysis of a *full* dialect of Prolog. For concreteness, we essentially follow the recently accepted ISO standard [19]. Our purpose is to review the features of the language which pose problems to global analysis and propose alternative solutions for dealing with these features. The most important objective is obviously to achieve correctness, but also as much accuracy as possible. Since arguably the main problem in static analysis is having dynamic code, which is not available at compile-time, we first propose a general solution for solving the problems associated with features such as dynamic predicates and meta-predicates, and consider other alternative solutions. The proposed alternatives are a combination of known solutions when they are useful, and novel solutions when the known ones are found lacking. The former are identified by giving references.

One of the motivations of our approach is that we would like to accommodate at the same time two types of users. First, the naive user, which would like analysis to be as transparent as possible. Second, we would also like to cater for the advanced user, which may like to guide the analysis in difficult places in order to obtain better optimizations. Thus, for each feature, we will propose solutions that require no user input, but we will also propose solutions that allow the user to provide input to the analysis process. This requires a clear interface to the analyzer at the program text level. Clearly, this need also arises when expressing the information gathered by the different analyses supported. We solve this by proposing an interface, in the form of *annotations*, which is useful not only for two-way communication between the user and the compiler, but also for the cooperation among different analysis tools and for connecting analyses with other modules of the compiler.

After neccessary preliminaries in Section 2, we propose several novel general solutions to deal with the analysis of dynamic programs in Section 3. A set of program annotations which can help in this task is then proposed in Section 4. We then revise our and previous solutions to deal with each of the language features in Section 5, except for modules and multifile programs, which are discussed in Section 6. There we propose a solution based on incremental analysis, and another one based on our program annotations. We conclude with Section 7.

We argue that the proposed set of solutions is the first one to allow the correct analysis of arbitrary programs which use the full power of the language without input from the user (while at the same time allowing such input if so desired). Given the length limitations and the objective of addressing the full language the presentation will be informal. Details can be found in [2].

2 Preliminaries and Notation

For simplicity we will assume that the abstract interpretation based analysis is constructed using the "Galois insertion" approach [11], in which an abstract domain is used which has a lattice structure, with a partial order denoted by $\sqsubseteq$, and whose top value we will refer to by $\top$, and its bottom value by $\perp$. We will refer to the least upper bound (lub) and greatest lower bound (glb) operators in the lattice by $\sqcup$ and $\sqcap$, respectively. The abstract computation proceeds using abstract counterparts of the concrete operations, the most relevant ones being unification (mgu^{α}) and composition ($\circ^{\alpha}$), which operate over abstract substitutions (α). Abstract unification is however often also expressed as a function $unify^{\alpha}$ which computes the abstract mgu of two concrete terms in the presence of a given abstract substitution.

Usually, a *collecting* semantics is used which attaches one or more (abstract) substitutions to program points (such as, for example, the point just before or just after the call of a given literal — the call and success substitutions for that literal). A goal dependent analysis associates abstract success substitutions to specific goals, in particular to call patterns, i.e. pairs of a goal and an abstract call substitution which expresses how the goal is called. Depending on the granularity of the analysis, one or more success substitutions can be computed for different call patterns at the same program point. Goal independent analyses compute abstract success substitutions for generic goals, regardless of the call substitution.

In general we will concentrate on top-down analyses, since they are at present the ones most frequently used in optimizing compilers. However, we believe the techniques proposed are equally applicable to bottom-up analyses. In the text, we consider in general goal dependent analyses, but point out solutions for goal independent analyses where appropriate (see, e.g., [16, 15, 8]).

The pairs of call and success patterns computed by the analysis, be it top-down or bottom-up, goal dependent or independent, will be denoted by $AOT^\alpha(P)$ for a given program P. A *most general goal pattern* (or simply "goal pattern," hereafter) of a predicate is a *normalized* goal for that predicate, i.e. a goal whose predicate symbol and arity are those of the predicate and where all arguments are distinct variables. In goal dependent analyses, for every call pattern of the form $(goal\_pattern, call\_substitution)$ of a program P there are one or more associated success substitutions which will be denoted hereafter by $AOT^\alpha(P, call\_pattern)$. The same holds for goal independent analysis, where the call pattern is simply reduced to the goal pattern. By *program* we refer to the entire program text that the compiler has access to, including any directives and annotations.

3 Static Analysis of Dynamic Program Text

A main problem in statically analyzing logic programs is that not all of the code that would actually be run is statically accesible to the analysis. This can occur either because the particular calls occuring at some places are dynamically constructed, or because the code defining some predicates is dynamically modified. The following problems appear: (1) How to compute success substitutions for the calls which are not known; we call this the success substitution problem, and (2) How to determine calls and call substitutions which may appear from the code which is not known; we call this the extra call pattern problem.

Consider the following program, to be analyzed with entry point `goal`. The predicate `p/2` is known to be dynamic, and may thus be modified at run-time.

```
goal:- ..., X=a, ..., p(X,Y), ...

:- dynamic p/2.
p(X,Y):- q(X,Y).

q(X,Y).
l(a,b).
```

Assume that the call pattern of the goal `p(X,Y)` in the analysis indicates that `X` is ground and `Y` free. If we do not consider the possibility of run-time modifications of the code, the success pattern for `p(X,Y)` is the same as the call pattern. Also, since no calls exist to `l/2`, its definition is dead code. Assume now that a clause "`p(X,Y):- l(X,Y).`" is asserted at run-time. The previous analysis information is not correct for two reasons. First, the success pattern of `p(X,Y)` should now indicate that `Y` is ground (success substitution problem). Second, a call for `l/2` now occurs which has not been considered in the previous analysis (extra call pattern problem).

The first problem is easier to solve: using appropriate topmost substitutions. We call an abstract substitution α *topmost* w.r.t. a tuple (set) of variables $\mathbf{x}$ iff $vars(\alpha) = \mathbf{x}$ and for all other substitution α' such that $vars(\alpha') = \mathbf{x}$, $\alpha' \sqsubseteq \alpha$. An abstract substitution α referring to variables $\mathbf{x}$ is said to be *topmost of* another substitution α', referring to the same variables, iff $\alpha \equiv \alpha' \circ^\alpha \alpha''$, where α'' is the topmost substitution w.r.t. $\mathbf{x}$. Therefore, for a given call substitution, the topmost abstract substitution w.r.t. it is the most accurate approximation which solves the success substitution problem. This is in contrast to roughly considering $\top$ or just giving up in the analysis. Topmost substitutions are preferred, since they are usually more accurate for some domains. For example, if a variable is known to be ground in the call substitution, it will continue being ground in the success substitution.

Note that this is in fact enough for goal independent analyses, for which the second problem does not apply. However, for goal dependent analyses the second problem needs to be solved in some way. This problem is caused by the impossibility of statically computing the subtree underlying a given call, either because this call is not known (it is statically undetermined), or because not all of the code defining the predicate for that call is available. Therefore, since from these subtrees new calls (and new call patterns) can appear, which affect other parts of the program, the whole analysis may not be correct.

There is a first straightforward solution to the extra call pattern problem. It can be tackled by simply assuming that there are unknown call patterns, and thus any of the predicates in the program may be called (either from the undetermined call or from within its subtree). This means that analysis may still proceed but topmost call patterns must be assumed for all predicates. This is similar to performing a goal independent analysis and it may allow some optimizations, but it will probably preclude others. However, if program multiple specialization [29, 25, 26] is done, a non-optimized version of the program should exist (since all the predicates in the program must be prepared to receive any input value), but other optimized versions could be inferred.

Consider the previous example. To solve the success substitution problem we can (a) assume a topmost substitution w.r.t. `X` and `Y`, which will indicate that nothing is known of these two variables; or (b) assume the topmost substitution w.r.t. the call substitution, which will indicate that nothing is known of `Y`, but still `X` is known to be ground. To solve the extra call pattern problem we can (a) assume new call patterns with topmost substitutions for all predicates in the program, since the asserted clause is not known during analysis; or (b) perform the transformation proposed below, which will isolate the problem to predicate `l/2`, which is the only one affected.

We propose a second complete solution which is general enough and very elegant, with the only penalty of some cost in code size. The key idea is to compile essentially two versions of the program — one that is a straightforward compilation of the original program, and another that is analyzed assuming that the only possible calls to each predicate are those that appear explicitly in the program. This version will contain all the optimizations, which will be performed ignoring the effect of the undetermined calls. Still, in the other version, any optimizations possible with a goal independent analysis, or a topmost call pattern goal dependent analysis, may be introduced. Calling from undetermined calls into the more optimized version of the program (which will possibly be unprepared for the call patterns created by such calls) is avoided by making such calls call the less optimized version of the program. This will take place automatically because the terms that will be built at run-time will use the names of the original predicates. When a predicate in the original program is called, it will also call predicates in the original program. Therefore, the original predicate names are used for the less optimized version, and predicates in the more optimized version are renamed in an appropriate way (we will assume for simplicity that it is by using the prefix "`opt_`"). Thus, correctness of a transformation such as the following is guaranteed. Assume that `call(X)` is an undetermined call. If a clause such as the first one appears in the program, the second one is added:

```
p(...) :- q(...), call(X), r(...).
opt_p(...) :- opt_q(...), call(X), opt_r(...).
```

The top-level rewrites calls which have been declared as entry points to the program so that the optimized version is accessed. Note that this also solves (if needed) the general problem of answering queries that have not been declared as entry points: they simply access the less optimized version of the program. If the top-level does also check the call patterns, then it guarantees that only the entry patterns used in the analysis will be executed. For the declared entry patterns, execution will start in the optimized program and will move to the original program to compute a resolution subtree each time an undetermined call is executed. Upon return from the undetermined call, execution will go back to the optimized program.

We shall see how this solution can be applied both to the case of meta-predicates and to that of dynamic predicates, allowing full optimizations to be performed in general to "dynamic" programs. The impact of the optimizations performed in the renamed copy of the program will depend on the time that execution stays in each of the versions. Therefore, the relative computational load of undetermined calls w.r.t. the whole program will condition the benefits of the optimizations achieved. The only drawback with this solution is that it implies keeping two full copies of the program, although only in case there are undetermined calls. In cases where code space is a pressing issue, the user should be given the choice of turning this copying on and off.

4 Program Annotations

Annotations are assertions regarding a program that are introduced as part of its code. Annotations refer to a given program point. We consider two general classes of program points: points inside a clause (such as, for example, before or after the execution of a given goal — the "goal level") and points that refer to a whole predicate (such as, for example, before entering or after exiting a predicate — the "predicate level"). At all levels annotations describe properties of the variables that appear in the program. We will call the descriptions of such properties *declarations*. There are at least two ways of representing declarations which we will call "property oriented" and "abstract domain oriented". In a property oriented annotation framework, there are declarations for each property a given variable or set of variables may have. Examples of such declarations are:

`mode(X,+)`	`X` is bound to a non-variable term
`term(X,r(Y))`	`X` is bound to term `r(Y)`
`depth(X,r/1)`	`X` is bound to a term `r(_)`

The property oriented approach presents two advantages. On one hand, it is easily extensible, provided one defines the semantics for the new properties one wants to add. On the other hand, it is also independent from any abstract domain for analysis. One only needs to define the semantics of each declaration, and, for each abstract domain, a translation into the corresponding abstract substitutions. For concreteness, and in order to avoid referring to any abstract domain in particular, we propose to use such a framework.

An alternative solution is to define declarations in an abstract domain oriented way. For example, for the sharing domain [21]:

`sharing([[X],[Y,Z]])` shows the sharing pattern among variables X, Y, Z

This is a simple enough solution but has the disadvantage that the meaning of such domains is often difficult for users to understand. Also, the interface is bound to change any time the domain changes. It has two other disadvantages. The semantics and the translation functions mentioned above have to be defined pairwise, i.e. one for each two different domains to be communicated. And, secondly, there can exist several (possibly overlapping) properties declared, one for each different domain. In the property oriented approach, additional properties that several domains might take advantage of are declared only once. In any case, both approaches are compatible via the *syntactic* scheme we propose.

Predicate Level: Entry Annotations One class of predicate level annotations are `entry` annotations. They are specified using a directive style syntax, as follows:

`:- entry(`*goal_pattern*`,`*declaration*`).`

These annotations state that calls to that predicate with the given abstract call substitution may exist at execution time. For example, the following annotation states that there can be a call to predicate `p/2` in which its two arguments are ground:

```
:- entry(p(X,Y), (ground(X),ground(Y)) ).
```

Entry annotations and goal dependent analysis. A crucial property of `entry` annotations, which makes them useful in goal dependent analyses, is that they must be *closed with respect to outside calls*. No call patterns other than those specified by the annotations in the program may occur from outside the program text. I.e., the list of `entry` annotations includes all calls that may occur to a program, apart from those which arise from the literals explicitly present in the program text. Obviously this is not an issue in goal independent analyses.

Entry annotations and multiple program specialization. If analysis is multivariant it is often convenient to create different versions of a predicate (multiple specialization). This allows implementing different optimizations in each version. Each one of these versions generally receives an automatically generated unique name in the multiply specialized program. However, in order to keep the multiple specialization process transparent to the user, whenever more than one version is generated for a predicate which is a declared entry point of the program (and, thus, appears in an `entry` directive), the original name of the predicate is reserved for the version that will be called upon program query. If more than one `entry` annotation appears for a predicate and different versions are used for different annotations, it is obviously not possible to assign to all of them the original name of the predicate. There are two solutions to this. The first one is to add a front end with the exported name and run-time tests to determine the version to use. However, this implies run-time overhead. As an alternative we allow the `entry` directive to have one more argument, which indicates the name to be used for the version corresponding to this entry point. For example, given:

```
:- entry(mmultiply(A,B,C),ground([A,B]),mmultiply_ground).
:- entry(mmultiply(A,B,C),true,mmultiply_any).
```

if these two entries originate different versions, they would be given different names. If two or more versions such as those above are collapsed into one, this one will get the name of any of the entry points and, in order to allow calls to all the names given in the annotations, binary clauses will be added to provide the other entry points to that same version.

Predicate Level: Trust Annotations In addition to the more standard `entry` annotations we propose a different kind of annotations at the predicate level, which take the following form:

:- `trust(`*goal_pattern*`,`*call_declaration*`,`*success_declaration*`).`

Declarations in `trust` annotations put in relation the call and the success patterns of calls to the given predicate. These annotations can be read as follows: if a literal that corresponds to *goal_pattern* is executed and *call_declaration* holds for the associated call substitution, then *success_declaration* holds for the associated success substitution. Thus, these annotations relate abstract call and success substitutions. Note that *call_declaration* can be empty (i.e., `true`). In this way, properties can be stated that must always hold for the success substitution, no matter what the call substitution is. This is useful also in goal independent analyses (and in this case it is equivalent to the "omode" declaration of [18]).

Let $(p(\mathbf{x}), \alpha)$ denote the call pattern and α' the success substitution of a given `trust` annotation of a program P. The semantics of `trust` implies that $\forall \alpha_c\ (\alpha_c \sqsubseteq \alpha \Rightarrow AOT^{\alpha}(P, (p(\mathbf{x}), \alpha_c)) \sqsubseteq \alpha')$. I.e., for all call substitutions approximated by that of the given call pattern, their success substitutions are approximated by that of the annotation. For this reason, the compiler will "trust" them. This justifies their consideration of "extra" information, and thus and in contrast to `entry` annotations, the list of `trust` annotations of a program does *not* have to be closed w.r.t. all possible call patterns occurring in the program.

One of the main uses of `trust` annotations is in describing predicates that are not present in the program text. For example, the following annotations describe the behavior of the predicate `p/2` for two possible call patterns:

```
:- trust(p(X,Y), (ground(X),free(Y)) , (ground(X),ground(Y)) ).
:- trust(p(X,Y), (free(X),ground(Y)) , (free(X),ground(Y))   ).
```

This would allow performing the analysis even if the code for `p/2` is not present. In that case the corresponding success information in the annotation can be used ("trusted") as success substitution.

In addition, `trust` annotations can be used to improve the analysis when the results of the analysis are imprecise. However, note that this does not save analyzing the predicate for the corresponding call pattern, since the abstract underlying subtree may contain call patterns that do not occur elsewhere in the program.

If we analyze a call pattern for which a `trust` annotation exists, two abstract success patterns will be available for it: that computed by the analysis (say α_s) and that given by the `trust` annotation (say α', for a call substitution α). As both must be correct, the intersection of them (which may be more accurate than any of them) must also be correct. The intersection among abstract substitutions (whose domain we have assumed has a lattice structure) is computed with the glb operator, $\sqcap$. Therefore, $AOT^{\alpha}(P, (p(\mathbf{x}), \alpha_c)) = \alpha_s \sqcap \alpha'$, provided that $\alpha_c \sqsubseteq \alpha$. Since $\forall \alpha_s \forall \alpha'\ (\alpha_s \sqcap \alpha' \sqsubseteq \alpha_s \wedge \alpha_s \sqcap \alpha' \sqsubseteq \alpha')$ correctness of the analysis within the `trust` semantics is guaranteed, i.e. $AOT^{\alpha}(P, (p(\mathbf{x}), \alpha_c)) \sqsubseteq \alpha'$ and $AOT^{\alpha}(P, (p(\mathbf{x}), \alpha_c)) \sqsubseteq \alpha_s$. However, if their informations are incompatible, their intersection is empty, and $\alpha_s \sqcap \alpha' = \perp$. This is an error (if $\alpha_s \neq \perp$ and also $\alpha' \neq \perp$), because the analysis information must be correct, and the same thing is assumed for the `trust` information. The analysis should give up and warn the user.

A similar scheme can be used to check the mutual consistency of annotations provided by the user. The result of the glb operation between inconsistent annotations will be $\perp$. Also, note that, in addition to improving the substitution at the given point, the trusted information can be used to improve previous patterns computed in the analysis. This might be done by "propagating" the information backwards in the analysis process.

Goal Level: Pragma Annotations Annotations at the goal level refer to the state of the variables of the clause just at the point where the annotation appears: between two literals, after the head of a clause or after the last literal of a clause.[1] We propose reserving the literal `pragma` (as in [24]) to enclose all necessary information referring to a given program point in a clause. It takes the form:

..., $goal_1$, pragma($declaration$), $goal_2$, ...

where the `pragma` information is valid before calling $goal_2$ and also after calling $goal_1$, that is, at the success point for $goal_1$ and at the call point of $goal_2$.

The information given by `pragma` can refer to any of the variables in the clause. The information is expressed using the same kind of declarations as in the predicate level annotations. This allows a uniform format for the declarations of properties in annotations at both the predicate and the goal level. These annotations are related to `trust` annotations in the sense that they give information that should be trusted by the compiler. Therefore, they have similar uses and a similar treatment that them.

5 Dealing with Standard Prolog

In this section we discuss different solutions for analyzing the full standard Prolog language. In order to do so we have divided the complete set of builtins offered by the language in several classes.

Builtins as Abstract Functions Many Prolog builtins can be dealt with efficiently and accurately during analysis by means of functions which capture their semantics. Such functions provide an (as accurate as possible) abstraction of every success substitution for any call to the corresponding builtin. This applies also to goal independent analyses, with minor modifications. It is interesting to note that the functions that describe builtin predicates are very similar in spirit to `trust` annotations. This is not surprising, if builtins are seen as Prolog predicates for which the code is not available. Since most of the treatment of builtins is rather straightforward, the presentation is very brief, concentrating on the more interesting cases of meta-predicates and dynamic predicates. In order to avoid reference to any particular abstract domain any functions described will be given in terms of simple minded `trust` annotations. For the reader interested in the details, the source code for the PLAI analyzer (available by ftp from `clip.dia.fi.upm.es`) contains detailed functions for all Prolog builtins and for a large collection of well known abstract domains. For a description of such functions for some builtins in a different domain see e.g. [10].

Control flow predicates include `true` and `repeat`, which have a simple treatment: identity can be used (i.e., they can be simply ignored). The abstraction of `fail` and `halt` is $\perp$. For cut (`!`) it is also possible to use the identity function (i.e.,

[1] Similar annotations can be used at other levels of granularity, from between head unifications to even between low level instructions, but we will limit the discussion for concreteness to goal-level program points.

ignore it). This is certainly correct in that it only implies that more cases than necessary will be computed in the analysis upon predicate exit, but may result in some cases (specially if *red cuts* –those which modify the meaning of a program– are used) in a certain loss of accuracy. This can be addressed by using a semantics which keeps track of sequences, rather than sets, of substitutions, as shown in [7]. Finally, exception handling can also be included in this class. The methods used by the different Prolog dialects for this purpose have been unified in the Prolog standard into two builtins: `catch` and `throw`. We propose a method for dealing with this new mechanism: note that, since analysis in general assumes that execution can fail at any point, literals of the form `catch(Goal,Catcher,Recovery)` (where execution starts in `Goal` and backtracks to `Recovery` if the exception described by `Catcher` occurs) can be safely approximated by the disjunction `(Goal;Recovery)`, and simply analyzed as a meta-call. The correctness of this transformation is based on the fact that no new control paths can appear due to an exception, since those paths are a subset of the ones considered by the analysis when it assumes that any goal may fail. The builtin `throw`, which explicitly raises an exception, can then be approximated by directly mapping it to failure, i.e. $\perp$.

The function corresponding to `=` is simply abstract unification. Specialized versions of the full abstract unification function can be used for other builtins such as `\=`, `functor`, `arg`, `univ` (`=..`), and `copy_term`. Other term and string manipulation builtins are relatively straightforward to implement. Arithmetic builtins and base type tests such as `is`, `>`, `@>`, `integer`, `var`, `number`, etc., usually also have a natural mapping in the abstract domain considered. In fact, their incomplete implementation in Prolog is an invaluable source of information for the analyzer upon their exit (which assumes that the predicate did not fail — failure is of course always considered as an alternative). For example, their mappings will include relations such as "`:- trust(is(X,Y),true,ground([X,Y])).`" or "`:- trust(var(X),true,free(X)).`" On the contrary, `==`, `\==`, and their arithmetic counterparts, are somewhat more involved, and are implemented (in the same way as with the term manipulation builtins above) by using specialized versions of the abstract unification function.

Output from the program does not directly pose any problem since the related predicates do not instantiate any variables or produce any other side effects beyond modifying external streams, whose effect can only be seen during input to the program. Thus, identity can again be used in this case. On the other hand, the external input cannot be determined beforehand. The main problem happens to be the success substitution problem. In the general case, analysis can always proceed by simply assuming topmost success substitutions in the domain.

The treatment of *directives* is somewhat peculiar. The directive `dynamic` is used to declare predicates which can be modified at run-time. Dynamic predicates will be considered in detail below. The directive `multifile` specifies that the definition of a predicate is not complete in the program. Multifile predicates can therefore be treated as either dynamic or imported predicates — see Section 6. The directives `include` and `ensure_loaded` must specify an accessible file, which can be read in and analyzed together with the current program. The directive `initialization` specifies new (concrete) entry points to the program.

Meta-Predicates Meta-predicates are predicates which use other predicates as arguments. All user defined meta-predicates are in this class but their treatment can be reduced to the treatment of the meta-call builtins they use. Such meta-calls are literals which call one of their arguments at run-time, converting at the time of the call a term into a goal. Builtins in this class are not only `call`, but also `bagof`, `findall`, `setof`, negation by failure, and `once` (single solution). Calls to the solution gathering builtins can be treated as normal (meta-)calls since most analyzers are "collecting" in the sense that they always consider all solutions to predicates. Negation by failure (`\+`) can be defined in terms of `call` and cut, and can be dealt with by combining the treatment of cut with the treatment of meta-calls. Single solution (`once`) can be dealt with in a similar way since it is equivalent to "`once(X) :- call(X), !.`".

Since meta-call builtins convert a term into a goal, they can be difficult to deal with if it is not possible to know at compile-time the exact nature of those terms [14, 18]. In particular, the success substitution problem for the meta-call appears, as well as the extra call pattern problem (within the code defining the corresponding predicate, and for the possible calls which can occur from such code). Both problems can be dealt with using the techniques in Section 3. First, topmost call patterns can be used for all predicates in the program, second, and alternatively, the renaming transformation can also be applied. In this case meta-calls that are fully determined either by declaration or as a result of analysis, and incorporated into the program text will call the more optimized version. Analysis will have taken into account the call patterns produced by such calls since they they would have been entered and analyzed as normal calls. I.e., the following transformation will take place:

```
..., pragma(term(X,p(Y))), call(X),... ⟹ ..., opt_p(Y),...
```

Meta-calls that are partially determined, such as, for example,

```
..., pragma(depth(X,p/1)), call(X), ...
```

are a special case. One solution is not to rename them. In that case they will be treated as undetermined meta-calls. Alternatively, the solution in the second item above can be used. It is necessary in this case to ensure that the optimized program will be entered upon reaching a partially determined meta-call. This can be done dynamically, using a special version of `call/1` or by providing binary predicates which transform the calls into new predicates which perform a mapping of the original terms (known from the analysis) into the renamed ones. Using this idea the example above may be transformed into a new literal and a new clause, as follows:

```
..., opt_call(X),...            opt_call(p(X)) :- opt_p(X).
```

Undetermined meta-calls will not be renamed, and thus will call the original (less optimized) code. This fulfills the correctness requirement, since these calls would not have been analyzed, and therefore can not be allowed to call the optimized code.

More precise solutions to both problems are possible if knowledge regarding the terms to be converted is available at compile-time. Thus, following [14], we can distinguish between:

- *Completely determined* meta-calls. These are calls in which the term (functor and arguments) is given in the program text (this is often the case for example in many uses of `bagof`, `findall`, `setof`, `\+`, and `once`), or can be inferred via some kind of analysis, as proposed in [14]. In the latter case they can even be incoporated into the program text before analysis. These calls can be analyzed in a straightforward way.
- *Partially determined* meta-calls. The exact term cannot be statically found, but at least its main functor can be determined by program analysis. Then, since the predicate that will be called at run-time is known, it is sufficient for analysis to enter only this predicate using the appropriate projection of the current abstract call substitution on the variables involved in the call.

– *Undetermined* meta-calls.

The first two classes distinguish subclasses of the *fully determined* predicates of [14], where certain interesting types of programs are characterized which allow the static determination of this generally undecidable property. Relying exclusively on program analysis, as in [14], has the disadvantage that it restricts the class of programs which can be optimized to those which are fully determined. Our previous solution solves the general case.

There are other possible solutions to the general case. The first and simplest one is to issue a warning if an undetermined meta-call is found and ask the user to provide information regarding the meta-terms. This can be easily done via `pragma` annotations. For example, the following annotation:

```
..., pragma(( term(X,p(Y)) ; term(X,q(Z)) )), call(X), ...
```

states that the term called in the meta-call is either `p(Y)` or `q(Z)`. Note also that this is in some way similar to giving entry mode information for the `p/1` and `q/1` predicates. This suggests another solution to the problem, which has been used before in Aquarius [26], in MA3 [28], and in previous versions of the PLAI analyzer [3]. The idea (cast in the terms of our discussion) is to take the position that meta-calls are *external calls*. Then, since `entry` annotations have to be closed with respect to external calls it is the user's responsibility to declare any entry points and patterns to predicates which can be "meta-called" via `entry` annotations. Accuracy of the analysis will depend on that of the information supplied by the user. These solutions have the disadvantage of putting the burden on the user — something that we would like to avoid at least for naive users. Our alternative solutions are completely transparent to the user.

Database Manipulation and Dynamic Predicates Database manipulation builtins include `assert`, `retract`, `abolish`, and `clause`. These builtins (with the exception of `clause`) affect the program itself by adding to or removing clauses from it. Predicates that can be affected by such builtins are called dynamic predicates and must usually be declared as such in modern Prolog implementations (and this is also the case in the ISO standard).

The potential problems created by the use of the database manipulation builtins are threefold. On the one hand, the extra call pattern problem appears again since the literals in the body of the new clauses that are added dynamically can produce new and different call patterns not considered during analysis. The success substitution problem also appears for literals which call dynamic predicates ("dynamic literals"). Even if abstract success substitutions can be computed from any static definition of the predicate which may be available at compile-time, it may change during program execution. On the other hand, there exists the additional problem of computing success substitutions for the calls to the database manipulation builtins themselves. We call this the "database builtin" success substitution problem. Note that `clause` —which can be viewed as a special case of retract— does not modify the database and thus clearly only has the third problem.

Solving the extra call pattern problem. From the correctness point of view, the extra call pattern problem only arises from the use of `assert`, but not from the use of `abolish` or `retract`. These predicates do not introduce new clauses in the program, and thus they do not introduce any new call patterns. This is true even for "intelligent" analyses which can infer definite success or failure of some goals, because these analyses must take `retract` into account to do so, or otherwise would themselves not be correct in general. Therefore, retraction is not a problem in our case. On the other hand, it is conceivable that more accuracy could be obtained if these predicates were analyzed more precisely since removing clauses may remove call patterns which in turn could make the analysis more precise. We discuss this in the context of incremental analysis at the end of the section. The discussion is general enough to subsume the above mentioned intelligent analyses.

The `assert` predicate is much more problematic, since it can introduce new clauses and through them new call patterns. The problem is compounded by the fact that asserted clauses can call predicates which are not declared as dynamic, and thus the effect is not confined to dynamic predicates. In any case, and as pointed out in [14], not all uses of `assert` are equally damaging. To distinguish these uses, we propose to divide dynamic predicates into the following types:

- `memo` only facts which are logical consequences of the program itself are asserted;
- `data` only facts are asserted, or, if clauses are asserted, they are never called (i.e., only read with `clause` or `retract`);
- `local_call` the dynamic predicate only calls other dynamic predicates;
- `global_call`.

The first two classes correspond to the *unit-assertive* and *green-assertive* predicates of [14], except that we have slightly extended the unit-assertive type by also considering in this type arbitrary predicates which are asserted/retracted but never called. Clauses used in this way can be seen as just recorded terms: simply a set of facts for the predicate symbol `:-/2`.

`data` predicates are guaranteed to produce no new call patterns and therefore they are safe with respect to the extra call pattern problem.[2] This is also the case for `memo` predicates since they only assert facts.[3] If all dynamic predicates are of the `local_call` type, then the analysis of the static program is correct except for the clauses defining the dynamic predicates themselves. Analysis can even ignore the clauses defining such predicates. Optimizations can then be performed over the program text except for those clauses, which in any case may not be such a big loss since in some systems such clauses are not compiled, but rather interpreted.

While the classification mentioned above is useful, two problems remain. The first one is how to detect that dynamic procedures are in the classes that are easy to analyze (dynamic predicates in principle need to be assumed in the `global_call` class). This can be done through analysis for certain programs, as shown in [14], but, as in the case of meta-calls, this does not offer a solution in all cases.

The general case in which `global_call` dynamic predicates appear in the program is similar to that which appeared with undetermined meta-calls. In fact, the calls that appear in the bodies of asserted clauses can be seen as undetermined meta-calls, and similar solutions apply. Additionally, the static clauses of the dynamic predicates themselves are subject to the same treatment as the rest of the program, and therefore subject to full optimization. Clearly, this solution can be combined with the previous ones when particular cases can be identified.

Solving the dynamic literal success substitution problem. If only `abolish` and `retract` are used in the program, the abstract success substitutions of the static

[2] In fact, the builtins `record` and `recorded` provide the functionality of `data` predicates but without the need for dynamic declarations and without affecting global analysis. However, those builtins are now absent from the Prolog standard.

[3] Note however that certain analyses, and especially cost analyses which are affected by program execution time, need to treat these predicates specially.

clauses of the dynamic predicates are a safe approximation of the run-time success substitutions. However, a loss of accuracy can occur, as the abstract success substitution for the remaining clauses (if any) may be more particular. In the presence of `assert`, a correct (but possibly inaccurate) analysis is obtained by using appropriate topmost abstract substitutions. Finally, note that in the case of `memo` predicates (and for certain properties) this problem is avoided since the success substitutions computed from the static program are correct.

Solving the database builtin success substitution problem. This problem does not affect `assert` and `abolish` since the success substitution for calls to these builtins is the same as the call substitution. On the other hand, success substitutions for `retract` (and `clause`) are more difficult to obtain. However, appropriate topmost substitutions can always be safely used. In the special case of dynamic predicates of the `memo` class, and if the term used as argument in the call to `retract` or `clause` is at least partially determined, abstract counterparts of the *static* clauses of the program can be used as approximations in order to compute a more precise success substitution (see [2] for more details).

Dynamic analysis and optimization. There is still another, quite different and interesting solution to the problem of dynamic predicates, which is based on incremental global analysis [17]. Note that in order to implement `assert` some systems include a copy of the full compiler at run-time. The idea would be to also include the (incremental) global analyzer and the analysis information for the program, computed for the static part of the program. The program is in principle optimized using this information but the optimizer is also assumed to be incremental. After each non-trivial assertion or retraction (some cases may be treated specially) the incremental global analysis and optimizer are rerun and any affected parts of the program reanalyzed (and reoptimized). This has the advantage of having fully optimized code at all times, at the cost of increasing the cost of calls to database manipulation predicates and of executable size. A system along these lines has been built by us for a parallelizing compiler. The results presented in [17] show that such a reanalysis can be made in a very small fraction of the normal compilation time.

6 Program Modules

The main problem with studying the impact of modularity in analysis (and the reason we have left the issue until this section) is the lack of even a de-facto standard. There have been many proposals for module systems in logic programming languages (see [5]). For concreteness, we will focus on that proposed in the new draft ISO standard [20]. In this standard, the module interface is *static*, i.e. each module in the program must declare the procedures it exports,[4] and imports. The `module` directive is used for this.

As already pointed out in [18] `module` directives provide the entry points for the analysis of a module for free. Thus, as far as entry points are concerned, only exported predicates need be considered. They can be analyzed using the substitutions declared in the `entry` annotations if available, and topmost otherwise. The analysis of literals which call imported predicates requires new approaches,

[4] This is in contrast with other module systems used in some Prolog implementations that allow entering the code in modules at arbitrary points other than those declared as exported. This defeats the purpose of modules. We will not discuss such module systems since the corresponding programs in general need to be treated as non modular programs from the point of view of analysis.

some of which are discussed in the following paragraphs. One advantage of modules is that they help encapsulate the propagation of complex situations such as with `global_call` dynamic predicates.

Compositional Analysis. Modular analyses based on compositional semantics (such as, for example, that of [9]) can be used to analyze programs split in modules. Such analyses leave the abstract substitutions for the predicates whose definitions are not available *open*, in the sense that some representation of the literals and their interaction with the abstract substitution is incorporated as a handle into the substitutions themselves. Once the corresponding module is analyzed and the (abstract) semantics of such open predicates known, substitutions can be composed via these handles. The main drawback of this interesting approach is that the result of the analysis is not definite if there are open predicates. In principle, this would force some optimizations to be delayed until the final composed semantics is known, which in general can only be done when the code for all modules is available. Therefore, although analysis can be performed for each module separately, optimizations (and thus, compilation) cannot in principle use the global information.

Incremental Analysis. When analyzing a module, each call to a predicate not declared in it is mapped to $\perp$. Each time analysis information is updated, it is applied directly to the parts of the analysis where this information may be relevant. Incremental analysis [17] is conservative: it is correct and optimal. By optimal we mean that if we put together in a single module the code for all modules (with the necessary renaming to avoid name clashes) and analyze it in the traditional way, we will obtain the same information. However, incremental analysis, in a very similar way to the previous solution, is only useful for optimization if the code for all modules is available, since the information obtained for one isolated module is only partial. On the other hand, if optimization is also made incremental, then this does present a solution to the general problem: modules are optimized as much as possible assuming no knowledge of the other modules. Optimizations will be correct with respect to the partial information available at that time. Upon module composition incremental reanalysis and reoptimization will make the composed optimized program always correct.

Note that Prolog compilers are incremental in the sense that at any point in time new clauses can be compiled into the program. Incremental analysis (aided by incremental optimization) allows the combination of full interactive program development with full global analysis based optimization.

Trust-Enhanced Module Interface. In [20] imported predicates have to be declared in the module importing them and such a module can only be compiled if all the module interfaces for the predicates it imports are defined, even if the actual code is not yet available. Note that the same happens for most languages with modules (e.g., Modula). When such languages have some kind of global analysis (e.g., type checking) the module interface also includes suitable declarations. We propose to augment the module interface definition so that it may include `trust` annotations for the exported predicates. Each call to a predicate not defined in the module being analyzed but exported by some module interface is in principle mapped to appropriate topmost substitutions. But if in the module interface there are one or more `trust` annotations applicable to the call pattern, such annotations will be used instead. Any call to a predicate not defined in that module and not present in any of the module interfaces can be safely mapped to $\perp$ during analysis (this corresponds to mapping program errors to failure – note that error can also be treated alternatively as a first class element in the analysis). The advantages are that we do not need the code for other modules and also that we can perform optimizations using the (inaccurate) analysis information obtained in this way.

Analysis using the trust-enhanced interface is correct, but it may be suboptimal. This can only be avoided if the programmer provides the most accurate `trust` annotations. The disadvantage of this method is that it requires the trust-enhanced interface for each module. However, the process of generating these `trust` annotations can be automated. Whenever the module is analyzed, the call/success patterns for each exported predicate in the module which are obtained by the analysis are written out in the module interface as `trust` annotations. From there, they will be seen by other modules during their analysis and will improve their exported information. A global fixpoint can be reached in a distributed way even if different modules are being developed by different programmers at different times and running the analysis only locally, provided that, as required by the module system, the module interfaces (but not necessarily the code) are always made visible to other modules.

Summary. In practice it may be useful to use a combination of incremental analysis and the trust-enhanced module interface The trust-enhanced interface can be used during the development phase to compile modules independently. Once the actual code for all modules is present incremental analysis can be used to analyze modules loading them one after the other. In this way we obtain the best accuracy.

Multifile predicates (those defined over more than one file or module) also need to be treated in a special way. They can be easily identified due to the `multifile` declaration. They are similar to `dynamic` predicates (and also imported predicates) in that if we analyze a file independently of others, some of the code of a predicate is missing. We can treat such predicates as dynamic predicates and assume topmost substitutions as their abstract success substitutions unless there is a `trust` annotation for them. When the whole program composed of several files is compiled, we can again use incremental analysis. At that point, clauses for predicates are added to the analysis using *incremental addition* [17] (regardless of whether these clauses belong to different files and/or modules).

A case also worth discussing is that of libraries. Usually utility libraries provide predicates with an intended use. The automatic generation of `trust` annotations after analysis can be used for each library to provide information regarding the exported predicates. This is done for different uses and the generated annotations are stored in the library interface. With this scheme it is not necessary to analyze a library predicate when it is used in different programs. Instead, it is only analyzed once, and the information stored in the `trust` annotation is used from then on. If new uses of the library predicates arise for a given program, the library code can be reanalyzed and recompiled, keeping track of this new use for future compilations. An alternative approach is to perform a goal independent analysis of the library, coupled with a goal dependent analysis for the particular call patterns used thereafter [8].

7 Conclusions

We have studied several ways in which optimizations based on static analysis can be guaranteed correct for programs which use the full power of Prolog, including modules. We have also introduced several types of program annotations that can be used to both increase the accuracy and efficiency of the analysis and to express its results. The proposed techniques offer different trade-offs between accuracy, analysis cost, and user involvement. We argue that the presented combination of known and novel techniques offers a comprehensive solution for the correct analysis of arbitrary programs using the full power of the language.

Acknowledgements

The work reported herein was partially supported by ESPRIT Project #6707 ParForce, and CICYT Project IPL-D. The authors would also like to thank M. García de la Banda, S. Debray, F. Ballesteros, M. Carro, S. Prestwich, S. Yan, and the anonymous referees for useful comments.

References

1. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
2. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data-Flow Analysis of Prolog Programs with Extra-Logical Features. Technical Report CLIP2/95.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, March 1995.
3. F. Bueno, M. García de la Banda, D. Cabeza, and M. Hermenegildo. The &-Prolog Compiler System – Automatic Parallelization Tools for LP. Technical Report CLIP5/93.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1993.
4. 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.
5. M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19–20:443–502, July 1994.
6. 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.
7. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search-Rule and the Cut. In *International Symposium on Logic Programming*, pages 157–171. MIT Press, November 1994.
8. M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal Dependent vs Goal Independent Analysis of Logic Programs. In F. Pfenning, editor, *Fifth International Conference on Logic Programming and Automated Reasoning*, number 822 in LNAI, pages 305–320, Kiev, Ukraine, July 1994. Springer-Verlag.
9. M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 451–464, ACM, 1993.
10. A. Cortesi and G. File. Abstract interpretation of prolog: the treatment of the built-ins. In *Proc. of the 1992 GULP Conference on Logic Programming*, pages 87–104. Italian Association for Logic Programming, June 1992.
11. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

12. S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.
13. S. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.
14. S.K. Debray. Flow analysis of dynamic logic programs. *Journal of Logic Programming*, 7(2):149–176, September 1989.
15. M. Gabbrielli, R. Giacobazzi, and G. Levi. Goal independency and call patterns in the analysis of logic programs. In *ACM Symposium on Applied Computing*. ACM, 1994.
16. Roberto Giacobazzi, Saumya Debray, and Giorgio Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, 1992. Association for Computing Machinery.
17. M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*. MIT Press, 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. International Organization for Standardization, National Physical Laboratory, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 – Part 1: General Core*, 1994.
20. International Organization for Standardization, National Physical Laboratory, Middlesex, England. *PROLOG. Working Draft 7.0 X3J17/95/1 – Part 2: Modules*, 1995.
21. D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.
22. 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.
23. 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.
24. University of Bristol, Katholieke Universiteit Leuven, and Universidad Politécnica de Madrid. Interface between the prince prolog analysers and the compiler. Technical Report KUL/PRINCE/92.1, Katholieke Universiteit Leuven, October 1992.
25. 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.
26. P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.
27. 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.
28. 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.
29. W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.

This article was processed using the LaTeX macro package with LLNCS style