facultad de informática

universidad politécnica de madrid

Data-Flow Analysis of Prolog Programs with Extra-Logical Features

F. Bueno
D. Cabeza
M. Hermenegildo
G. Puebla

TR Number CLIP2/95.0

Data-Flow Analysis of Prolog Programs with Extra-Logical Features

Technical Report Number: CLIP2/95.0
March 1995

Authors

F. Bueno, D. Cabeza, M. Hermenegildo, G. Puebla
{bueno, dcabeza, herme, german}@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid — UPM
Campus de Montengancedo
28660 — Boadilla del Monte, Spain
Phone: +34-1-336-7435
Fax: +34-1-352-4819

Keywords

Data–flow Analysis, Abstract Interpretation, Standard Prolog.

Acknowledgements

This work 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, F. Ballesteros, M. Carro, S. Prestwich, and S. Yan.

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. Such problems relate to dealing correctly with all builtins, including meta–logical and extra–logical predicates, with dynamic predicates (where the program is modified during execution), and with the absence of certain program text during compilation. Existing proposals for dealing with such issues generally restrict in one way or another the classes of programs which can be analyzed if the information from analysis is to be used for program optimization. This paper attempts to fill this gap by considering a full dialect of Prolog, essentially following the recently proposed 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 using the full power of the language.

Contents

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 [17, 26, 22, 25, 4, 13, 1, 12, 21, 8]. Most proposals to date have concentrated on proposing 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, 17, 22, 7] generally restrict the use of builtin predicates in one way or another (and thus the class of programs which can be analyzed) if the information from analysis is to be used for program optimization.

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 proposed ISO draft standard [18]. Although not yet a standard, this seems an appropriate choice because it essentially gathers and unifies most of the de-facto standard features present in current Prolog dialects. 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. The proposed alternatives are a combination of known solutions when they are useful, and novel solutions when the known ones are found lacking.

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 also would 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, in order to accommodate the first class of users, 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 arises also for example in the output of the analyzer when expressing the information gathered by the different analyses supported. We propose 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.

We argue that the proposed set of solutions is the first one to allow the correct analysis of arbitrary programs using 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 [3]. The rest of the paper proceeds as follows: after presenting some preliminary notions and notation (Section 2) we first deal with the crucial issue of program annotations (Section 3). We then review all builtins in the ISO standard (as described in [18]), and propose several solutions to the problems they pose to analysis. These builtins include those that can usually be handled by abstract functions in most domains (Section 4.1), meta–calls (Section 4.2), and database manipulation builtins and dynamic predicates (Section 4.3). Finally, we discuss program modules (Section 5), and present our conclusions.

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, 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.

We will call an abstract substitution α *topmost* w.r.t. a tuple (set) of variables $\vec{x}$ iff $vars(\alpha) = \vec{x}$ and for all other substitution α' such that $vars(\alpha') = \vec{x}$, $\alpha' \sqsubseteq \alpha$. An abstract substitution α referring to variables $\vec{x}$ 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. $\vec{x}$.

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). Traditionally, a distinction is made between top–down and bottom–up analyses based on whether computation is performed starting at the queries or at the program facts. These have in turn been associated with goal dependent and goal independent analyses, respectively. However, recent results [15, 10] show that call dependence is not tied to a given form of analysis. 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.

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. 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. The issues related to multifile and modular programs, and interactive compilation, will be addressed in Section 5. In top–down analyses, annotations may support the specification of given queries in the form of entry points and their call patterns; $AOT^\alpha(P)$ will then be referred to these queries. The issue of determining the entry points and call patterns for which the analysis is correct will be addressed in sections 3 and 5.

The *predicate spec* (or *indicator* [18]) for the predicate whose name is `predicate_name` and whose arity is `n` is `predicate_name/n`. The declaration `:- term.` is a *directive*. Directives are *not* queries [18]. A *most general goal pattern* (or simply "goal pattern," hereafter) for a predicate spec is a *normalized* goal for that predicate, i.e. a goal whose predicate symbol and arity are the predicate spec and where all arguments are distinct variables.

3 Program Annotations

Annotations are assertions regarding a program that are introduced as part of its code. They can state properties which hold for the program they appear in. In that case the annotations do not modify the semantics of the non–annotated program. Rather, they reflect (often in an abstract way) an aspect of the program meaning. They can also state properties of other programs whose text is not present but which are to be composed with the present program. In that case such annotations reflect the semantics of the composed program. Program annotations can be both input to and output from the analyzer. When used as input, annotations are a way to provide the analyzer with additional information so that it can generate more precise information. When used as output, they represent the information obtained by the analyzer that will eventually be used by other parts of the compiler (including perhaps other analyzers) or shown to the programmer.

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"). Correspondingly there are different annotations for each of these levels. At all levels annotations describe properties of the variables that appear in the program (be it in the clause or in the arguments of a procedure). We will call the descriptions of such properties *declarations*. In general, such descriptions must be done in some abstract way since often it is necessary to represent an infinite number of concrete cases. Declarations should be written in a syntax that is compatible with the language (in this case, Prolog) and, ideally, in a domain–independent way. Thus, syntactically, declarations will in general be Prolog terms containing variables (or variable designators). Such terms encode assertions about the state of such variables at run–time.

There are at least two ways of representing declarations which we will call "property oriented" and "abstract domain oriented". In a property oriented framework, there are declarations for each property that 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)`
`ground(X)`	`X` is bound to ground term
`free(X)`	`X` is bound to free variable
`depth(X,r/1)`	`X` is bound to a term `r(_)`
`aliased(X,Y)`	`X` and `Y` are aliased
`occur([X,Y])`	the same variables occur in the terms bound to `X` and `Y`
`true`	no information (the top substitution)
`false`	the computation has failed ($\perp$)

For concreteness, and in order to avoid referring to any abstract domain in particular, we propose to use such a framework. In addition to the terms such as those above we assume that declarations can also be first order formulae formed by combining declarations with the connectives of first order logic. In the following, we will assume that declarations are always in conjunctive normal form (i.e. they are simplified in that way when being read). Comma and semicolon will denote, as usual, conjunction and disjunction, respectively.

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. An alternative solution is to define declarations in an abstract domain oriented way, i.e. representing declarations through some encoding in Prolog terms of the elements of the abstract domain used in the analysis. For example, for the sharing domain [20]:

`sharing([[X],[Y,Z]])` the sharing pattern among variables X, Y, Z is $\{\{X\},\{Y,Z\}\}$

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. Although this approach can also be extended if more domains are to be incorporated for analysis, it has two other disadvantages. The semantics and the translation functions above mentioned 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 proposed above (and in the following) in that abstract substitutions can also be seen as "properties" by encapsulating abstract substitutions for each domain in a term.

3.1 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*`).`

Declarations in `entry` annotations refer to the state of the variables appearing as arguments of calls to the given predicate. These annotations simply state that calls to that predicate with the given abstract call substitution may exist at execution time. For example, the annotation:

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

states that there can be a call to predicate `p/2` in which its two arguments are ground. The annotations:

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

and

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

are equivalent. In general, several annotations for the same predicate can be viewed as forming a disjunction. The entire single declaration for a given predicate, once put in conjunctive normal form, can be seen as stating a number of alternative ways in which the predicate can be called.

The idea of `entry` annotations is not novel. They are similar in purpose to other declarations which have been previously proposed (also referred to as "mode," "qmode," "imode," etc. declarations) to guide different goal dependent analyses [17, 26, 22, 25, 4, 13, 1, 22, 12, 21, 8]. The name is due to VanRoy and we have kept it instead of the one used previously by us ("qmode"), which we feel is less natural.

Entry annotations and goal dependent analysis.

A crucial property of `entry` annotations, which makes them useful in the goal dependent analyses mentioned above, is that they must be *closed with respect to outside calls*. By this we mean that no call patterns other than those specified by the annotations of 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. Thus, for now, we assume that they define *all* entry points, and optionally, their call patterns. Obviously this is not an issue in goal independent analyses.

Note that according to this definition, a program with no `entry` annotations is not a useful program because none of its predicates may be called from outside, i.e. it is entirely dead code. There are two alternatives in this situation: the first one is to simply issue a warning to the user. However, in our effort to support the user who perhaps does not want to provide any information to the analyzer but still would like it to do what it can, another alternative is to analyze the program but assuming that any predicate may be called in any possible way from outside. This is equivalent to assuming an `entry` annotation for each predicate in the program with the topmost substitution for the argument variables (i.e., `true`). On the other hand, if any `entry` annotation is present (showing thus a will of the user to help the analysis) then it is assumed to be closed with respect to calls from outside the program text.

Entry annotations and multiple program specialization.

When optimizing a program in the presence of a multivariant analysis it is often convenient to create different versions of a predicate, through a technique known as multiple specialization [28, 24, 26]. 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. However, 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, one would be called as `mmultiply_ground/3` and the other as `mmultiply_any/3`. Of course 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. In practice, both solutions can be used simultaneously: if multiple specialization is desired but with a single entry point, a single `entry` directive should be supplied, which should express the cases in a disjunction. In this way even if there are several versions for the predicate, there will only be one exported version, and that one will keep the original name. Run–time tests will be used to determine the appropriate version.

3.2 Predicate Level: Trust Annotations

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

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

Declarations in `trust` annotations put in relation the call and the success points of calls to the given predicate. These annotations can be read as follows: *if* a literal that corresponds to *goal_pattern* is executed *and* the *call_declaration* holds for the associated call substitution, then the *success_declaration* holds for the associated success substitution. Thus, these annotations relate abstract call and success substitutions. Note that the *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 [17]).

Let $(p(\vec{x}), \alpha)$ denote the call pattern and α' the success substitution of a given `trust` annotation of a program P. The semantics of `trust` implies that $\forall \alpha_c \; (\alpha_c \sqsubseteq \alpha \Rightarrow AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) \sqsubseteq \alpha')$. 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. If the declaration of one annotation is a disjunction, the lub of the different conjunctions will be used in place of α' in the above formula. If several annotations exist for the same call pattern, or several call patterns in these annotations approximate the one actually occurring at a given point, the glb of the success substitutions (as justified below) will be used. Note that this implies that, in contrast to `entry` annotations, several `trust` annotations are assumed to form a conjunction. Also 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 definitions of predicates that are not present in the program text (we will return to this issue in greater length in sections 4.1 and 5). 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 provided that the calls to `p/2` that appear in the program conform to (i.e., are identical to or contained in) one of the two call patterns in the `trust` annotations above. 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 for example if it is observed that for (one or more) call patterns the results of the analysis are inaccurate. However, note that the existence of such an annotation does not save analyzing the predicate for the corresponding call pattern: this is still necessary in order to compute the abstract subtree underlying that call pattern, since it may contain call patterns that do not occur elsewhere in the program. Otherwise the analysis would not be correct (and no optimizations could be performed) for the predicates and goals in that subtree, since there would be missing call patterns.

After having analyzed a literal for whose predicate a `trust` annotation exists, an interesting situation arises in that, upon exit from the call, two abstract success patterns will be available for the call pattern analyzed (or a comparable one): that computed by the analysis (say α_s) and that given by the `trust` annotation. As both the `trust` information and that generated by the analyzer 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, $\sqcap$.[1] During analysis, for every abstract call substitution α_c, with corresponding success substitution α_s, s.t. $\alpha_c \sqsubseteq \alpha$, α_s is substituted by $\alpha_s \sqcap \alpha'$, where α' is the abstract success substitution given by the `trust` annotation(s) which apply to α_c. Therefore, $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) = \alpha_s \sqcap \alpha'$. 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. $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) \sqsubseteq \alpha'$ and $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) \sqsubseteq \alpha_s$.

It is interesting to study the different situations which may appear regarding the `trust` information (i.e. α') and the one computed during analysis (α_s):

1. They are identical, $\alpha_s \sqcap \alpha' = \alpha_s = \alpha'$.

2. Information from analysis is more particular than that of the annotation. In this case the computed information is preferred, $\alpha_s \sqcap \alpha' = \alpha_s$.

3. Information in the annotation is more particular than that computed in the analysis. The information supplied in the annotation is preferred (and assumed correct — "trusted"), $\alpha_s \sqcap \alpha' = \alpha'$.

4. They have non–empty intersection. The most accurate information is their intersection, thus $\alpha_s \sqcap \alpha'$ is used, too.

5. They have incompatible information. Their intersection is empty, and $\alpha_s \sqcap \alpha' = \perp$. This is an error, 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.

[1] Although this operation is not generally implemented in top–down analyzers, which compute the fixpoint upwards, we believe that, in general, it is not difficult to implement, and a small burden when compared to the utility brought in by the `trust` annotation.

An obvious alternative to the scheme outlined above is to always use the information in the `trust` annotation as success substitution, $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) = \alpha'$. In this case `trust` annotations must be used with care, because if the information supplied is more general than what the analysis can obtain, a loss of accuracy occurs.

Although the use of entry (and omode) declarations has also been proposed for some purposes similar to our `trust` annotations [26, 17], we believe that the use of `trust` provides a superior solution because they make it possible to relate several call patterns with their associated success patterns.

3.3 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.[2] We propose reserving the literal `pragma` (as in [23]) 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 intended meaning of `pragma` as part of the program is the one given by the following definition:

```
pragma(_).
```

i.e., the inclusion of pragma annotations does not alter the meaning of the program in any way.

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. Despite this, `pragma` in general is richer, since it can express certain assertions which predicate level annotations cannot, due to the lack of the appropriate context. This is because they refer to relationships between variables in different goals, possibly including also existential variables of the clause. For example, the following annotation:

```
p(X,Y) :- q(X,Z), pragma(ground(Z)), r(Z,Y).
```

states that in the clause above, after execution of `q(X,Z)`, `Z` will always be bound to a ground term. This cannot be expressed at the predicate level of `p/2`.

Pragma annotations are related to `trust` annotations in the sense that they give information that should be trusted by the compiler. They are also related to `entry`, since a `pragma` annotation needs to be exhaustive in the sense that its declaration specifies everything that can occur at the corresponding program point. In the same way as with `trust` annotations, `pragma` annotations can be used to specify the information at a program point after a particular predicate is called, but in this case for a particular literal rather than for the whole predicate definition.

[2]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.

As with `trust`, they can be used either for predicates whose definition is not available, or to improve the information derived from the analysis of a particular literal of that predicate. In the latter case again the literal and its corresponding subtree still needs to be analyzed in order for the compiler to be able to perform optimizations, but the exit information can be improved by the contents of the `pragma` annotation, if it is better than that obtained by the analysis. In general, a similar set of cases regarding the respective accuracy of the inferred and declared information as with the `trust` annotations applies.

In practice it is sometimes useful to relate information that appears in different `pragma` annotations occurring at different points of a clause. This allows for example relating two components (elements of a disjunction) of two different `pragma` annotations at the call and success point of a literal, in the same way as in a `trust` annotation. This requires a slightly more expressive syntax than that presented up to now. It suffices to add an (optional) identifier to each declaration, that we will call a key. Declarations in different `pragma` annotations in a clause can be related by giving them the same key. These extended annotations have the form:

```
pragma((key:declaration ; ...))
```

where `key` can be any ground term, which acts as a reference relating distinct annotations at different (call or success) points. The same key appearing in two (or more) declarations at different points will state that whenever the properties specified at a given point hold (and only when they hold), then those specified with the same key at later points will also hold. Under this point of view, `trust` predicate level annotations summarize a set of `pragma` goal level annotations (although `trust` annotations have to be true for *all* literals of the predicate).

4 Dealing with Standard Prolog

In the previous sections we have presented a number of user annotations and already addressed a number of general issues in the practical analysis of programs. We now 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, which are treated in the following sections.

4.1 Builtins as Abstract Functions

Many Prolog builtins are declarative and they 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 of any call to the corresponding builtin. These can be more or less elaborate, from, for example, calling the full abstract unification function for supporting `=` to simply identity for `true`. This applies also to goal independent analyses, with minor modifications. For example, where identity is used in goal dependent analysis a function returning the empty substitution is used in a goal independent analysis.

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. For example, the identity function mentioned above corresponds, in goal dependent and goal independent analyses respectively, to the following `trust` annotations:

```
:- trust(true,X,X).                    :- trust(true,true,true).
```

In this section, we revisit the usual treatment of builtins via abstract functions. Since most of the treatment is rather straightforward the presentation is very brief, concentrating only on a few, slightly more involved cases, in order to allow space for discussing the more interesting cases of meta–logical and dynamic predicates, which will be addressed in the following sections. 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. These functions are also summarized in [2]. Finally, it should be noted that the accuracy of all the solutions proposed can of course be improved by using `pragma` annotations.

Control flow.

These 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 example, `fail` can be described by the following annotation:

```
:- trust(fail,true,false).
```

For cut (`!`) it is also possible to use the identity function (i.e., 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 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 (meta–calls, including all solutions predicates, will be considered in Section 4.2). 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 those considered by the analysis when it assumes that any goal may fail. The builtin `throw`, which explicitly raises and exception, can then be approximated by the identity function (i.e., ignored). Even more accurately, if we can determine statically that the exception will be raised, then `throw` can in those cases be mapped to failure, i.e. $\perp$.

Unification and term manipulation.

As mentioned before, 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. For example, the abstraction of the standard string manipulation builtin `atom_codes` (which replaces `name/2`) could include the following `trust` annotations, among others:

```
:- trust(atom_codes(X,Y),true,ground([X,Y])).
:- trust(atom_codes(X,Y),true,(atom(X),depth(Y,'.'/2))).
```

Arithmetic, comparison, and testing.

Arithmetic builtins and base type tests such as `is`, `>`, `@>`, `integer`, `var`, `number`, etc., usually 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 the following:

```
:- trust(is(X,Y),true,ground([X,Y])).
:- 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.

Data input/output.

Program output 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 behavior of builtins which are related to external input cannot be determined beforehand. The main problem is that no success substitution can be computed during analysis for the different call patterns to the builtin since the success substitution depends on the state of objects that are external to the Prolog system, such as files or the user. Depending on the abstract domain, things can however be simple in some cases. For example, for a domain tracking groundness, simple input builtins like `get_char` can be easily abstracted (it always produces ground input). But for more complex input predicates such as `read` and for more complex domains this cannot be done. However, analysis can always proceed by simply assuming that no information is available. In this case, the most general abstract substitution $\top$ is assumed as success substitution for the call to the input predicate. In fact, analysis can do better by considering the topmost abstract substitution w.r.t. the variables in its arguments, or, even better, the topmost substitution of the call pattern w.r.t. those variables. The latter is preferred, since this type of topmost abstract substitution is 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 addition, as always, `trust/pragma` annotations may be given to improve the topmost substitutions.

Directives.

The treatment of directives is somewhat peculiar. The directive `dynamic` will be considered in Section 4.3. 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 5.

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. The directive `module` also specifies new entry points to the program. In the absence of `entry` annotations for these entry points, topmost substitutions can be used. See Section 5 for a discussion of modules.

4.2 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 (`\+`) is also a meta–predicate, since `\+` can be defined as

```
\+ X :- call(X), !, fail.
\+ X.
```

It can be dealt with by combining the treatment of cut with the treatment of meta–predicates. Single solution (`once`) is also a meta–call and can be dealt with in a similar way as above 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, 17]. In particular, this raises the following problems:

1. How to compute success substitutions for the calls to meta–call builtins.
2. How to compute the subtrees underlying calls to such builtins. Also, 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.

The first problem is easier to solve: the same approach as for input builtins can be used, i.e., using appropriate topmost substitutions. Note that this is in fact enough for goal independent analyses, for which the second problem is not relevant. However, for goal dependent analyses the second problem needs to be solved in some way.

A more general solution to both problems is possible if knowledge regarding the terms to be converted is available at compile–time. Clearly, if 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`), then the meta–call can be analyzed in a straightforward way. If the term is not obvious from the text of the program the nature of the term being used in the meta–call can sometimes be inferred via a type, or depth-k or, in general, state of instantiation analysis, as proposed in [14]. As a result of such an analysis perhaps the actual term to be called can be determined, in which case the treatment outlined above applies. If the exact term cannot be statically found but at least its main functor can be determined as a result of some 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. We will call the first class *completely determined* meta–calls and the second one *partially determined* meta–calls. These classes distinguish subclasses of the *fully determined* predicates defined in [14]. Following [14] we will refer to the cases where the meta–term is unknown as *undetermined* meta–calls.[3] In [14] 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], however has the disadvantage that it restricts the class of programs which can be optimized to those which are fully determined.

Since our aim is to analyze all programs, we provide a number of solutions for dealing with undetermined meta–calls. The first and simplest solution 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 [27] and in previous versions of the PLAI analyzer [5]. 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 modes to meta–predicates via `entry` annotations.[4] However, if no multiple specialization is used, only one variant is generated for each predicate. This variant will be more or less optimized depending on the accuracy of the information supplied by the user. If the types of calls that can appear in the meta–calls are very general, then nearly all opportunities for optimization will be lost. It can also be very tedious for the user to give information for all the possible new calls.

The above solutions have the disadvantage of putting the burden on the user — something that we would like to avoid at least for naive users. We propose two alternative solutions that are completely transparent to the user. The first is to simply observe that fully undetermined meta–calls do not completely preclude analysis of the program. Having solved the first problem above (the success substitution of the meta–call) the second 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 meta–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. However, it will probably preclude others and, in particular, program specialization (since all the predicates in the program must be prepared to receive any input value).

Finally, we propose another, complete solution which has none of the problems of the solutions above and can cover all meta–calls, 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 (although any optimizations possible with a goal independent analysis may be introduced), and another that is analyzed assuming that the only possible calls to each predicate are those that appear explicitly in the program, including completely determined meta–calls. This version will contain all the optimizations, which will be performed ignoring the effect of undetermined meta–calls. Predicates in this more optimized version are renamed in an appropriate way (we will assume for simplicity that it is by using the prefix "`opt_`").

[3] Note, however, that if at run–time the meta–call is still *undetermined* an error will be reported. Similarly, if it can be determined at compile–time that the argument of the meta–call is, for example, free an error should also be reported.

[4] The same solution is given for dynamic predicates and predicates in bodies of asserted clauses — see Section 4.3.

Calling from undetermined meta–calls into the more optimized version of the program (which will possibly be unprepared for the call patterns created by such meta–calls) is avoided by making such calls call the less optimized version of the program. This is achieved via the following transformation, where `call(X)` is assumed to be an undetermined meta–call:

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

is transformed into

```
    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 meta–call is executed. Upon return from the undetermined meta–call execution will go back to the optimized program.

Note that the renamed copy of the program will not be used by the calls in undetermined meta–calls. 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. Thus, correctness is guaranteed during the execution of the meta–call.

Meta–calls that are fully determined (either by declaration or as a result of analysis) can be incorporated into the program text and will call the more optimized version. Analysis will have taken into account the call patterns produced by such calls since at analysis time fully determined meta–calls are entered and analyzed as normal calls. I.e., for example,

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

will be simply transformed at compile–time into

```
..., 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 effect of these calls, which is much more isolated than that of undetermined calls, may be taken into account during the analysis for the more optimized version of the program by assuming all possible call patterns for only the predicates that are called in such calls. It is also necessary in that 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:

```
  ..., opt_call(X), ...

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

The impact of the optimizations performed in the 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 meta–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 meta–calls. If cases where code space is a pressing issue, the user should be given the choice of turning this copying on and off.

4.3 Database Manipulation and Dynamic Predicates

Database manipulation builtins include `assert`, `retract`, `abolish`, and `clause`. These predicates (with the exception of `clause`) affect the program itself by adding to or removing clauses from it. Predicates that can be affected by such predicates 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 idea of modifying the program during execution might appear to run, in principle, conceptually counter to the idea of static analysis. However, all is certainly not lost and there are still quite a number of opportunities for optimizing dynamic programs. The potential problems created by the use of the database manipulation builtins are threefold:

1. The literals in the body of the new clauses that are added dynamically to the program can produce new and different call patterns not considered during analysis. This has to somehow be taken into account for the analysis to be correct. We will call this the "extra call pattern" problem.

2. How to compute success substitutions for literals which call dynamic predicates. 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. We will call this the "dynamic literal success substitution" problem.

3. How to compute success substitutions for the calls to the database manipulation builtins themselves. We will 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 above. Note also that the second and third problems above can always be solved by taking appropriate topmost success substitutions, as before. We will propose later some better solutions for these two problems. But first, we will concentrate on the the extra call pattern problem which is by far the most serious and difficult to solve.

Solving the extra call pattern problem.

As in the case of meta–calls, the extra call pattern problem does not affect goal independent analyses, since such analyses do not rely on particular call patterns. Note also that, at least 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. 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.[5]

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`	default

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. These can be simply considered as a set of facts for the predicate symbol `:-/2`.

A `data` predicate can be viewed as a set of terms that can be recorded and retrieved.[6] Calls and retracts to data predicates have both the same effect — data retrieval (no further computation is performed). In any case the advantage of `data` predicates is that they are guaranteed to produce no new call patterns and therefore they are safe with respect to the extra call pattern problem. This is also the case for `memo` predicates since they only assert facts.[7]

Other dynamic predicates that are interesting with respect to the extra call pattern problem are `local_call` predicates. If all dynamic predicates are of this 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

[5] See the discussion on incremental analysis at the end of the section for a general solution to this problem.

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

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

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. An obvious alternative is to allow the user to express such a classification directly. For this purpose we propose to enrich the `dynamic` directive as follows:

```
:- dynamic(predicate_spec,declared_type).
```

where the declared types are those mentioned above. Standard dynamic directives are assumed to be of the `global_call` type.

Still, the general case in which `global_call` dynamic predicates appear in the program (either because insufficient information is given by the user, because the type of certain predicates could not be determined statically, or simply because certain dynamic predicates are really of the `global_call` type) needs to be addressed. The problem is then 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.

The simplest solution in order to cater for the naive user is, as before, to resort to analyzing all predicates for topmost call patterns. This can be time consuming and prevent some optimizations, but is always correct and implies no user burden. There is also again the alternative, used in Aquarius [26], in MA3 [27], and in previous versions of the PLAI analyzer [22, 5], of viewing calls from asserted clauses as external calls and make it the user's responsibility to declare any extra calls produced by dynamic predicates via `entry` annotations. The disadvantage here is again the burden on the user, and the advantage potentially better optimization.

Finally, we propose a similar solution to the last one proposed in the case of the undetermined meta–calls: it involves again having two copies of the program, one with few optimizations (based perhaps on a topmost call pattern analysis for all predicates) and one with the full optimizations (based on an analysis ignoring any clauses not present in the program), as explained in Section 4.2. There we showed how meta–calls would directly use the less optimized version due to the renaming mechanism. The same applies here. Whenever a clause for a dynamic predicate is asserted, the literals in its body will use the original (less optimized) predicates, which have been compiled for any call pattern. In this way correctness is always guaranteed. The discussion regarding the relevance of the optimizations is the same as in Section 4.2. In fact, the static clauses of the dynamic predicates themselves are subject to the same treatment as the rest of the program. Clearly, this solution can be combined with the previously mentioned optimizations 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 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`, as mentioned before, it is always possible to generate a correct (but possibly inaccurate) success substitution for dynamic literals by using appropriate topmost abstract substitutions. This is correct but may introduce some inaccuracy. If the user knows how the dynamic predicates are going to behave, valuable information regarding the success substitutions for the predicate can be given via `trust/pragma` annotations. 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, `retract` (and `clause`) cannot be directly analyzed. However, appropriate topmost substitutions can 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 [3] for more details).

Summary and other approaches.

In conclusion, we have studied several ways in which optimizations based on static analysis can still be guaranteed correct for dynamic programs. In particular, we have proposed a technique (keeping two copies of the program with different levels of optimization) whereby the most serious problem of the appearance of extra call patterns is solved in a very general way, without putting any extra burden on the user, and at only the cost of some code space. This, in combination with any of the other techniques for solving the other two simpler problems, offers a solution that is general enough to deal with any kind of programs, without burdening the user, in contrast with previous solutions, which rely on identifying certain classes of programs, e.g., [14] or on programmer supplied annotations [26, 27].

There is still another, quite different and interesting solution to the problem of dynamic predicates, which is based on incremental global analysis [16]. 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 (ground facts and simple facts, and clauses which can be determined not to affect the previously inferred information 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 [16] show that such a reanalysis can be made in a very small fraction of the normal compilation time.

5 Program Modules

Up to now we have assumed a single program text (in one or more files). However, programs are obviously normally better developed in a modular fashion. In this section we address the issues of modularity and also to some extent interactive development. 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 [6]). For concreteness, however, we will focus on the module system proposed in the new draft ISO standard [19]. In this standard, the module interface is *static*, i.e. each module in the program must declare the procedures it exports.[8] This

[8] 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

is done using the `module` directive. 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. Imported predicates have to be declared also.

As already pointed out in [17] `module` directives have the nice property that they provide the entry points for the analysis of a module for free. Thus, if we assume for now that the program consists of only one module with its `module` directive and there are no `entry` declarations, we can safely assume that the only entry points are the exported predicates. Analysis simply needs to start at the procedures corresponding to such points with appropriate topmost abstract substitutions. In line with our previous assumptions, if any `entry` annotations are present for such exported predicates, they will be assumed to be closed with respect to external calls. Then, analysis will start at the exported predicates with the substitutions declared in the `entry` annotations if available (and topmost otherwise).

In the general case where there are multiple modules, the analysis of literals which call imported predicates requires new approaches, some of which we discuss in the following paragraphs.

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. Thus, total correctness of the partial abstract substitutions computed at a particular moment cannot always be guaranteed. 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.

A different approach is by means of the previously mentioned technique of incremental analysis (e.g. [16]). Each call to a predicate not declared in the module being analyzed is mapped to $\perp$. Each time a new predicate is analyzed, the information obtained is applied directly to the parts of the analysis where this information may be relevant. The information obtained with incremental analysis 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 obtain the same information as with incremental analysis. 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. The information obtained for one isolated module is partial and in principle cannot be used to optimize a module independently of others, thus precluding its use for modular optimization from the point of view of software engineering. On the other hand, if optimization is also made incremental, as mentioned in previous sections, 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 com-

modular programs from the point of view of analysis.

position 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 – this allows a powerful interactive development environment. Another advantage of incremental analysis (aided by incremental optimization) is that it allows the combination of full interactive program development with full global analysis based optimization.

Trust–Enhanced Module Interface.

We propose yet another approach which is based on the fact that in [19] 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. For data–flow analysis 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. However, the fact that we may assume $\top$ for those calls that are imported from other modules makes this analysis procedure suboptimal. This can be avoided if the programmer provides `trust` annotations that are as accurate as possible for imported predicates. The disadvantage of this method is that it requires the trust–enhanced interface for each module. However, note that the process of generating these `trust` annotations can be automated. In a given module, if the programmer has provided no `trust` annotation suitable for our purposes, we assume topmost substitutions for the imported predicates. 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 for programs split in modules. The trust–enhanced interface can be used during the development phase of a modular program to compile modules independently. However, as hinted at before, the use of the trust–interface does not always guarantee that the analysis information obtained once the analysis of all modules converges is optimal. We believe, however, that analysis information is most important once the actual code for all modules is present and the resulting composed program is compiled. At this moment, incremental analysis can be used to analyze modules loading them one after the other. Annotations which appear in the code of the modules will be used, but `trust` annotations in module interfaces might be ignored at this point. In this way we will obtain the most accurate analysis information.

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 module 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 modules is compiled, we can again use incremental analysis. At that point, clauses for predicates are added to the analysis using *incremental addition* [16] (regardless of whether these clauses belong to different modules).

A case also worth discussing is that of libraries. Usually utility libraries provide predicates with an intended use. These predicates can be used by many different modules, even belonging to different programs. For such library files we can use the automatic generation of `trust` annotations after analysis to provide information regarding the exported predicates. This is done for all the different uses and the generated `trust` annotations 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 for that use, keeping track of this new use for future compilations. An alternative approach to the analysis of libraries is to perform a goal independent analysis for them, coupled with a goal dependent analysis when the library is used for the particular use in the program [10].

6 Conclusions

We have proposed a number of techniques for the analysis of a dialect of the Prolog language, essentially following the recently proposed ISO standard. We argue that these solutions, when considered as a whole, provide a means for the analysis and optimization of the full language. This can be done without any input from the user, even in the difficult cases of dynamic programs, albeit at some loss of optimization and/or increase in code size. 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. We have also discussed software engineering issues such as modular program development. The proposed techniques offer different trade–offs between accuracy, analysis cost, and user involvement. While we feel there is still plenty of work left in achieving more accurate analysis of many features (such as, for example, cut) we argue that the presented combination of known and novel techniques is the most comprehensive solution to date for the correct analysis of arbitrary programs using the full power of the language.

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. García de la Banda, M. Hermenegildo, and G. Puebla. Abstract Functions for the Analysis of Builtins in the PLAI System. Technical Report CLIP1/95.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, January 1995.

3. 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.

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. 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.

6. M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19–20:443–502, July 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. 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.

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, Charleston, South Carolina, 1993. ACM.

10. 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.

11. 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.

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

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

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 Press, 1994.

16. 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.

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

18. International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

19. International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. Working Draft 7.0 X3J17/95/1 — Part 2: Modules*, 1995.

20. D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

21. 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.

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

23. 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.

24. G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

25. 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.

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

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