Università degli Studi di Bologna
DEIS

An Abstract Interpreter for Improving the Efficiency of Dynamic Modular Logic Languages

Anna Ciampolini Evelina Lamma
Paola Mello Vito Pagone

January, 31, 1996

DEIS Technical Report no. DEIS-LIA-96-002 *LIA Series no. 9*

An Abstract Interpreter for Improving the Efficiency of Dynamic Modular Logic Languages

Anna Ciampolini [1] Evelina Lamma [1] Paola Mello [2]
Vito Pagone [3]

[1] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
Tel: +39 51 6443033, Fax: +39 51 6443073
{aciampolini,elamma}@deis.unibo.it
[2] *Istituto di Ingegneria, Università di Ferrara*
Via Saragat, 44100 Ferrara, Italy
pmello@ing.unife.it
[3] *Departamento de Computacion y Tecnologia de la Informacion*
Universidad Simon Bolivar. Apto. 8900,
Caracas 1080 A, Venezuela
vpagone@usb.ve

January, 31, 1996

Abstract. In this work, we focus on modular logic languages where module composition is performed through union of clauses and where the set of definitions to be used in order to evaluate a goal can be extended during the computation through implication goals. In order to improve the implementation of these languages, we consider the application of static analysis techniques. In particular, we apply abstract interpretation in order to compute the minimal sets of modules (*minimal contexts*) which are needed for any successful computation for a given predicate. The results of the analysis are exploited by the underlying implementation in order to reduce the overhead due to failing, useless computations. The abstract interpreter has been implemented in Prolog, through meta-interpretation techniques.

Keywords: *Modular Logic Programs, Static Analysis, Abstract Interpretation, Meta-programming*

Contents

1 Introduction

Several applications both in the Software Engineering and the Artificial Intelligence area require some forms of modularity and structuring of knowledge. This has motivated a considerable research effort about *modular logic programming* which has led to several proposals and implemented systems (see [6] for a survey). Two of the key issues when defining a modular logic language are expressive power and efficiency.

From the language design point of view, modularity in logic programming has been conceived along two orthogonal lines of research.

Some proposals [20, 14, 10, 1, 4] have focused primarily on the issue of *programming in the large*. In this first approach, module-composition is a *meta-linguistic* mechanism: modules are viewed as sets of Horn clauses and their (algebraic) composition is modeled in terms of various operations on the component clauses (e.g., union).

The other approach (see [11, 17, 18, 13] (focusing primarily on the issue of programming-in-the-small) instruments a logic language with *linguistic* abstraction mechanisms richer than those offered by Horn clauses. This approach originated with the work of Miller in [17] where he proposed a modular language based on the use of implication goals of the kind $D \supset G$ in the body of clauses (where D is a set of possibly extended clauses, and G is a goal).

In this paper, we focus on this second approach, and show how to improve the efficiency of modular logic languages where the set of definitions to be used in order to evaluate a goal can be extended during the computation through implication goals. In particular, we focus on modular logic languages adopting *dynamic scope rules*. In this respect, the current set of definitions is the union of the clauses belonging to the modules involved in the implication goals encountered. Thus, clauses can see each others, whichever the order of execution of implication goals has been.

In the case of modular languages adopting dynamic scope rules, the binding of predicate calls to definitions has to be performed only at run-time, when the call is raised. The binding requires to perform a dynamic search in the run-time program representation. For instance, in [13] a linear search takes place dynamically for each predicate call along a *context stack*, whereas in [12] a new hash function linking predicate names to their definitions has to be computed for each implication goal. Therefore, a key point for a successful implementation is to reduce the overhead due to this dynamic search. This can be obtained in two ways. First, by avoiding the search when possible. Second, by enhancing the efficiency of the search technique adopted. In this paper, we mainly focus on the former approach, and consider the application of static analysis techniques [9] in order to avoid the overhead due to try to bind a predicate which certainly leads to failing, useless computations. The abstract analysis is grounded on the fixpoint semantics defined in [3] for the class of modular logic languages with dynamic scope rules. In particular, we use abstract interpretation to determine the *minimal contexts* (i.e., the minimal sets of modules) which are needed in order to have a possibly successful computation for a given predicate p. When a call to p is raised, if the current set of modules in use is not a superset of any of the minimal contexts of p, then the computation fails immediately. If no minimal context exists, then the call can be replaced by a failure through a proper program transformation. Moreover, we extend the analysis in order to reduce backtracking. In practice, we introduce a form of indexing on program clauses in order be able to find the right clause to apply on the basis of the current context. To this purpose, we need a finer analysis that is able to distinguish between different clauses defining the same predicate, and therefore associates minimal contexts to each clause.

The analysis of modular logic languages has also been considered in [8] where the authors develop a compositional abstract interpretation for *open logic programs* [2], i.e., modular programs consisting of *definite clauses* only. Thus, no linguistic extension is introduced. Our approach is different in two respects. First, we consider a richer class of languages, also encompassing dynamic module composition. For these languages no compositional semantics has been developed till now, and thus the analysis we develop is not compositional as well. Furthermore, we develop an abstract interpretation of a fixpoint semantics with the specific purpose of improving efficiency by avoiding expensive, unsuccessful computations.

In the paper, we also present the implementation of the abstract interpreter which has been obtained through meta-interpretation. In particular, we have exploited a forward Prolog (meta-)interpreter [21], and built on top of it a forward interpreter able to deal with the modular language considered. This interpreter is able to compute the set of modules used for each argument, once a propositional version of the program is given. Then a proper filter program determines, among them, the minimal contexts for each predicate.

The results of the abstract analysis are exploited by an implementation (based on a bit-vector representation) which is described in [7]. In particular, this implementation maintains the set of minimal contexts of each predicate p in a proper data structure. When call to p is raised, before determining the binding for it, the system checks if the current set of modules in use is a superset of (or is equal to) at least one *minimal context*. If this is the case, the minimal context drives the search for the proper clause to be applied. Otherwise, the call will certainly lead to a failure, and thus the binding of the predicate call is not even searched for.

The work is organized as follows. In section 2 we present the language under consideration and its operational and fixpoint semantics. In section 3 we present the abstract interpretation framework, and its implementation in section 4. Section 5 refines the analysis in order to compute minimal contexts for clauses rather than predicates, and present the implementation of the new abstract interpreter. Some conclusions follows.

2 The Modular Language and its Semantics

In this section, we give a formal description of the modular logic language in term of both operational and fixpoint semantics.

2.1 Syntax and Operational Semantics

The modular language we consider is a proper extension of Horn clause logic: it is in practice that defined in [17], provided that no explicit quantification occurs for implication goals. A program is composed by a set of modules. Each module consists of a set of (possibly extended) definite clauses, and is identified by a unique constant name.

Following the style of [17], we will use $\top$ to denote the distinguished formula *true* and three metalinguistic variables – A, D and G – to stand respectively for atomic formulas, (extended) definite clauses and goals. Using these conventions the structure of (extended) Horn clauses can be expressed in terms of the following syntax:

$$
\begin{aligned}
D &::= A \mid D \wedge D \mid \forall x D \mid G \supset A \\
G &::= \top \mid A \mid G \wedge G \mid D \supset G
\end{aligned}
$$

We suppose that D and G are closed formulas.

According to the above definition, a logic program P can be viewed as a conjunctive D-formula or equivalently as a set of D-formulas. In [17], Miller formalized the operational semantics of implication goals by extending the provability relation $\vdash$ for Horn Clauses with the following inference rule:

$$
\text{(AUGMENT)} \quad \frac{P \cup D \vdash G}{P \vdash D \supset G}
$$

When D and G are closed formulas, the above rule provides a direct formalization of the deduction theorem: to prove $D \supset G$, assume D and prove G from $P \cup D$.

In the following discussion, we will assume that modules are designated by names, and that the same name may have multiple occurrences in a program. In particular, modules can be introduced as named collections of clauses and programs can be structured as collections of modules each one dedicated to answer a specific class of queries. Cross-referencing between modules and module-composition can then be accounted for relying on the workings of implication goals. If, in module M, the answer to a goal G requires that the clauses of module M_1 be loaded, then we will simply enforce the evaluation of G in the composition of M and M_1 by means of the implication goal $M_1 \supset G$.

For reasons of notational convenience, we will often adopt the conventional Prolog syntax for clauses and write $A{:-}G_1, ..., G_n$ as an alternative notation for $(G_1 \wedge \cdots \wedge G_n) \supset A$, and implication goals as M `>>` G where M is a module name. In doing so, we will also assume that $A{:-}G_1, ..., G_n$ is in normal form, i.e. that all the variables occurring in it are universally quantified.

Example 2.1 *Let us consider the following program:*

```
m1:     delete_or_append(X,Y,Z):-m4>>r(X,Y,Z).
m2:     delete(X,[],[]).
        delete(X,[X|Y], Z):-delete(X,Y,Z).
m3:     append([],X,X).

m4:     r([X],Y,Z):-delete(X,Y,Z).
        r(X,Y,Z):-append(X,Y,Z).
        delete(X, [U|Y], [U|Z]):-delete(X,Y,Z).
        append([X|U],V,[X|W]):-append(U,V,W).
```

Program clauses are distributed into four modules. The goal:
`:-m1>>m3>>delete_or_append(X,Y,Z)`
causes a non-deterministic selection between the `append` *operation, and the* `delete` *operation on the given list* `Y` *and element* `X`*.*

Other modular languages with the same structure, but adopting more strict scoping rules than those effecting the language of Miller, were proposed by Giordano *et al.* in [11], and by Monteiro and Porto in [18] as the foundation for Contextual Logic Programming. In this paper, we do not consider these languages since for them the binding of predicate calls can be determined quite efficiently (see [6] for a survey). In fact, it is determined at compile time for the language of [11], at context extension time for Contextual Logic Programming (see [13]).

2.2 Fixpoint Semantics

In [3], the authors provide a model-theoretic and fixpoint semantics for the modular language here presented. The two declarative semantics are proved equivalent to the operational one. In this section, we recall the fixpoint semantics since it will be adopted as a basis for the abstract analysis (see section 3).

It is worth mentioning that different, alternative approaches (mainly based on Kripke-like or situation semantics) have been followed by other researchers [16, 17, 15, 11, 19] in order to model the behavior of other (possibly richer) modular languages based on implication goals. A survey can be found in [6].

Let P be a program and $\mathcal{M}_P$ the set of names of the modules of P. The Herbrand base of P, $\mathcal{B}_P$, is defined as usual [25] as the set of all the ground atomic formulas built from the Herbrand universe of P. The Herbrand universe is the standard one, and we suppose it is disjoint from the set of module names $\mathcal{M}_P$.

Definition 2.2 *Let P be a program. The set of program compositions (*contexts*) which can be built from the modules of P is defined as: $\mathcal{C}_P = Powerset(\mathcal{M}_P)$.*

$\mathcal{M}_P$ being finite, $\mathcal{C}_P$ is finite as well.

Definition 2.3 *Let P be a program. The* extended Herbrand base *is defined as:*

$$\mathcal{B}_{CP} = \{\langle c, A\rangle \mid c \in \mathcal{C}_P, A \in \mathcal{B}_P\}$$

An interpretation is any subset of $\mathcal{B}_{CP}$.

Thus, the interpretation domain is the powerset of $\mathcal{B}_{CP}$, and it is a complete lattice under set inclusion.

Definition 2.4 *Given a context c, a goal formula G and an interpretation I, we denote by $c \models_I G$ the fact that G is true in c with respect to I.*

The immediate consequence operator is defined in terms of the following truth relation.

Definition 2.5 *Let P be a program, $I \in Powerset(\mathcal{B}_{CP})$, $A \in \mathcal{B}_P$, $M \in \mathcal{M}_P$ and G, G_1, G_2 goal formulas, and $c \in \mathcal{C}_P$. The truth relation $\models_I$ is defined as follows:*

$$\begin{array}{lll} c \models_I \top & & \\ c \models_I A & iff & \langle c, A\rangle \in I \\ c \models_I G_1, G_2 & iff & c \models_I G_1 \ \wedge \ c \models_I G_2 \\ c \models_I M \supset G & iff & \{M\} \cup c \models_I G \end{array}$$

Definition 2.6 *Let P be a program. The transformation function*

$$T_P : \ Powerset(\mathcal{B}_{CP}) \ \mapsto \ Powerset(\mathcal{B}_{CP})$$

is defined as follows:

$$T_P(I) = \{\langle\{m\} \cup c, A\rangle \mid \exists\ G \supset A \in ground(m) \ \wedge \ m \in \mathcal{M}_P \ \wedge \ c \models_I G\}$$

where $ground(m)$ represents the grounded version of m's code.

As proved in [3], this transformation function is monotone and continue, and therefore has a least fixpoint.

3 The Abstract Analysis

In this section, we show how abstract interpretation techniques can help in improving the implementation of dynamic modular logic programs.

The aim is to statically determine, for each predicate symbol p occurring in a given program P, the *minimal contexts* (i.e., the minimal sets of modules) which are needed in order to possibly have a successful computation. More precisely, if a call to a predicate p is raised in a set of modules C (*i.e.*, a *context*) which is not a superset of (or equal to) some minimal context of p, then the computation for p will certainly lead to failure, and therefore it is convenient to raise a failure immediately. Notice that the fact that C is equal or is a superset of some minimal context of p does not guarantee that a successful computation for p exists. In this respect, the condition above is only a *necessary* but not a *sufficient* condition.

3.1 The Abstract Domain and Transformation Function

For the concrete domain, $\mathcal{D}$, we rely upon the definitions introduced in section 2.2. In particular, $\mathcal{D}$ is defined as the powerset of the extended Herbrand base (see definition 2.3): $\mathcal{D} = Powerset(\mathcal{B}_{CP})$

Definition 3.1 *Let $\mathcal{P}_P$ denote the set of predicate symbols occurring in program P. We define $\mathcal{D}^\alpha$ as the powerset of the cartesian product $\mathcal{C}_P \times \mathcal{P}_P$:*

$$\mathcal{D}^\alpha = Powerset(\mathcal{C}_P \times \mathcal{P}_P)$$

We associate with the domain, $\mathcal{D}^\alpha$ a pre-order relation, $\sqsubseteq$, defined as follows.

Definition 3.2 *Let I^α and J^α be two elements of $\mathcal{D}^\alpha$. We say that $I^\alpha \sqsubseteq_{pre} J^\alpha$ iff $\forall \langle c, p\rangle \in I^\alpha\ \exists \langle c', p\rangle \in J^\alpha : c' \subseteq c$.*

To get a partial order, it suffices to consider the domain $\mathcal{D}^\alpha$ modulo the equivalence relation induced by the pre-order $\sqsubseteq_{pre}$ by grouping together those sets that are equivalent with respect to $\sqsubseteq_{pre}$ [23]. Two sets I and J belong to the same equivalent class iff $I \sqsubseteq_{pre} J$ and $J \sqsubseteq_{pre} I$.

Let $[\mathcal{D}^\alpha]$ denote the quotient of $\mathcal{D}^\alpha$ with respect to $\sqsubseteq_{pre}$. The equivalence classes are *partially ordered* by $\sqsubseteq_{pre}$: for equivalence classes $P, Q \in [\mathcal{D}^\alpha]$, $P \sqsubseteq Q$ iff for all $A \in P$ and $B \in Q$, $A \sqsubseteq_{pre} B$. Let $[I^\alpha]$ represent the equivalence class containing the set $I^\alpha \in \mathcal{D}^\alpha$.

We choose $[\mathcal{D}^\alpha]$ as abstract domain. It can be proved that $[\mathcal{D}^\alpha]$ equipped with the partial order $\sqsubseteq$ is a complete lattice, with bottom element $\perp^\alpha = [\emptyset]$, and top element $\top^\alpha = [\{\langle \mathcal{C}_P, p\rangle \mid p \in \mathcal{P}_P\}]$.

Abstraction is performed by the function α, defined as

$$\alpha = Pred \circ Min \circ \{\_\}$$

where:

- $Pred : \mathcal{D} \mapsto \mathcal{D}^\alpha$ is such that $\forall I \in \mathcal{D}$:

$$Pred(I) = \{\langle c, p\rangle | \langle c,\ A\rangle \in I\ \wedge\ p \text{ is the predicate symbol of } A\}$$

- $Min : \mathcal{D}^\alpha \mapsto \mathcal{D}^\alpha$ is such that $\forall I^\alpha \in \mathcal{D}^\alpha$:

$$Min(I^\alpha) = \{\langle c, p\rangle |\ \nexists \langle c', p\rangle \in I^\alpha : c' \subset c\}$$

- $\{\_\} : \mathcal{D}^\alpha \mapsto [\mathcal{D}^\alpha]$ maps $I^\alpha \in \mathcal{D}^\alpha$ to $[I^\alpha]$

In practice, the function $Pred$ abstracts each atom $A \in \mathcal{B}_P$ with its predicate symbol p.

As concerns function Min, a set of contexts, $C(q)$, can be associated with each predicate symbol q appearing in a given interpretation $I^\alpha \in \mathcal{D}^\alpha$:

$$C(q) = \{c | c \in \mathcal{C}_P \wedge \langle c, q\rangle \in I^\alpha\}$$

The powerset of $C(q)$ is a complete partial order: each chain in this set represents the contexts associated to q linked together by the (usual) subset relation $\subset$. Function Min reduces each set $C(q)$ of I^α to the set composed by only the lower bound elements of each chain. In other words, the function Min eliminates from the set I^α all couples $\langle c, p\rangle$ such that $c \supset c' \wedge \langle c', p\rangle \in I^\alpha$. For this reason, each context c appearing in $Min(I^\alpha)$ is said to be *minimal*.

Finally, function $\{\_\}$ maps elements of $\mathcal{D}^\alpha$ in equivalence classes of $[\mathcal{D}^\alpha]$.

Let us now introduce the concretization function:

Definition 3.3 *The concretization function $\gamma : [\mathcal{D}^\alpha] \mapsto \mathcal{D}$ is defined as follows:*

$$\forall [I^\alpha] \in [\mathcal{D}^\alpha] : \gamma([I^\alpha]) = \{\langle c, A\rangle\ | \langle c, p\rangle \in I \wedge I \in [I^\alpha] \wedge p = pred(A) \wedge A \in \mathcal{B}_P\}$$

Therefore, for the equivalence class $[\{\langle c', p\rangle\}] \in [\mathcal{D}^\alpha]$, function γ generates all the couples $\langle c, A\rangle \in \mathcal{D}$ such that $A \in \mathcal{B}_P$ have predicate symbol p, and $c' \subseteq c$. In practice, besides mapping predicate names into ground atoms, γ builds a set of chains having c' as the lower bound, thus generating all possible contexts including c'.

Assuming the ordering relationship $\subseteq$ on the concrete domain $\mathcal{D}$ and the ordering relationship $\sqsubseteq$ on the abstract domain $[\mathcal{D}^\alpha]$, it is easy to prove that the abstraction function α is total and monotone. Furthermore, the pair of functions α and γ yields a Galois insertion [9].

Proposition 3.4 *Given* $(\mathcal{D}, \subseteq)$ *and* $([\mathcal{D}^\alpha], \sqsubseteq)$*, the following properties hold:*
(i) α *is total and monotone;*
(ii) γ *is total and monotone;*
(iii) $\forall d \in \mathcal{D}: \ d \subseteq \gamma(\alpha(d)))$*;*
(iv) $\forall d^\alpha \in [\mathcal{D}^\alpha]: \ \alpha(\gamma(d^\alpha))) = d^\alpha$*.*

Relying upon definition 2.6, we define the abstract transformation function, T_P^α, as the composition:

$$T_P^\alpha = \alpha \circ T_P \circ \gamma$$

As usual, the abstract analysis of the given program P should iteratively apply the transformation function T_P^α, starting from the set $\perp^\alpha = [\emptyset]$, until the least fixpoint is reached. In practice, the least fixpoint is a set of couples $\{\langle c, p\rangle\}$ where each element represents a *minimal context* (c) that might (but doesn't guarantee) lead to a successful computation for an atom A such that: $pred(A) = p$. The advantage is that we can statically determine which contexts will certainly lead to failure for A (i.e., all contexts that do not include any context belonging to couples in $T_P^\alpha \uparrow \omega$), and thus we can avoid to execute certainly useless computations.

Example 3.5 *Let us consider the program of example 2.1. We are interested in finding, for each predicate symbol of* $\mathcal{P}_P$*, the minimal contexts which are separately needed for a possibly successful computation, through the iterated application of the abstract function, as follows:*
$T_P^\alpha \uparrow 0 = \perp^\alpha = [\emptyset]$
$T_P^\alpha \uparrow 1 = \alpha(T_P(\gamma(\perp^\alpha))) = \{\langle\{m2\}, delete\rangle, \langle\{m3\}, append\rangle\}$
$T_P^\alpha \uparrow 2 = T_P^\alpha \uparrow 1 \ \bigcup \ \{\langle\{m2, m4\}, r\rangle, \langle\{m3, m4\}, r\rangle\}$
$T_P^\alpha \uparrow 3 = T_P^\alpha \uparrow 2 \ \bigcup \ \{\langle\{m1, m2\}, delete\_or\_append\rangle, \langle\{m1, m3\}, delete\_or\_append\rangle\}$
Finally, in the next iteration T_P^α *reaches its least fixpoint:*
$T_P^\alpha \uparrow 4 = T_P^\alpha \uparrow 3 = T_P^\alpha \uparrow \omega$
Thus, from the results of the abstract analysis we can conclude that any call to predicate `delete_or_append` *certainly fails if neither* `m1` *and* `m2` *nor* `m1` *and* `m3` *belong to the current context.*

It is worth to notice that, starting from analysis results, a trivial source-to-source transformation can be applied in order to improve performances. Basically, this transformation amounts to deleting some clauses (which certainly lead to failure) from the original source program.

Let us focus on a predicate $p \in \mathcal{P}_P$ with no minimal context associated with. Then, we can conclude that no successful computation for p exists, and therefore delete any definition of p from each module of P. Moreover, we can also conclude that the application of any clause $C_l = q(\ )$:- $\ldots\ p(\ ), \ldots$ of P, having a call to p in its body, will fail, and therefore delete C_l as well.

3.2 The Impact of the Analysis

The abstract analysis presented so far yields for each predicate p, the set of minimal contexts that are necessary for a successful computation for p. This information can be employed during the computation to avoid to raise a call when it certainly will fail.

In the following, we refer to the implementation schema described in [7].This implementation extends the Warren's Abstract Machine (defined by D.H.D. Warren in [27]) with new instructions and data structures. Each module of a program P is separately compiled. For each predicate p, the compiler produces the following structures, maintained in the code area of the program: (1) a *predicate vector*, i.e. a bit vector recording the modules where predicate p is defined. If p has a definition in module m_i, for instance, then the i-th bit of the predicate vector of p is set to 1; (2) a *predicate frame*. This data structure records, for each module m_i defining p, the address of the first instruction for p in m_i's compiled code.

The implementation maintains the dynamic representation of a program (i.e., the set of modules currently in use or *context*) through a *bit vector* (with dimension equal to the number of modules of the program). At each instant of the computation, the run-time representation of the set of clauses in use is, in practice, given by the current context. In terms of the bit vector representation, if module m_i belongs to the current context, the i-th bit is set to 1. In the underlying extended WAM, a register `CR` maintains the current context as a bit vector. `CR` register is possibly modified when an implication goal is encountered.

This implementation schema also exploits the results produced by the abstract analysis by associating with each predicate p of the program its *minimal contexts*. We call this set the *predicate label* of p. Each *minimal context* in the label of p is represented by a bit vector. For instance, if a module m_i belongs to a *minimal context* c, the i-th bit in the vector representing c is set. With this representation schema, a call to a predicate p implies a dynamic check of the current context. If the current context in use is a superset of (or is equal to) a particular *minimal context*, the call *might* lead to a successful computation; otherwise (i.e., none of the minimal contexts of predicate p is a subset of - or equal to - the current context), the call will certainly lead to a failure and therefore can be safely avoided. Each comparison between the current context (represented by the content of `CR` register) and one of the elements of the *label* of predicate p is easily implemented with bit-wise logic operations between bit vectors (see [7] for details). The overhead imposed by this operations is very limited, for to the very low cost of bit-wise operations, and is largely balanced by the advantages deriving from avoiding the call, as shown by the following example.

Example 3.6 *Let us consider again example 2.1. The set of minimal contexts obtained for this program can be represented by the following* predicate labels*:*

$$L(delete\_or\_append) = \{[1,1,0,0],[1,0,1,0]\}$$
$$L(delete) = \{[0,1,0,0]\}$$
$$L(append) = \{[0,0,1,0]\}$$
$$L(r) = \{[0,1,0,1],[0,0,1,1]\}$$

Let us consider the top goal:
`:- m1>>m3>>delete_or_append([g],[a,b,c,d,e,f], X).`
In this case, the use of abstract analysis can avoid to raise an expensive call to the predicate `delete`*. In fact, when a call to* `r` *is raised, the first clause in* `m4` *is activated. After the unification, the abstract machine compares the value of* `CR` *(*$[1,0,1,1]$*) with the* minimal contexts *in predicate label* $L(delete)$ *in order to test if* `CR` *is a superset of (or equal to) at least one them. The check gives an unsuccessful result, and the call to* `delete` *is therefore avoided. It is worth to notice that without this check, the call to* `delete` *predicate would have caused a number of applications of the* `delete` *clause in* `m4` *equal to the length of the second argument of* `delete_or_append` *(i.e., equal to six in the example above), before reaching a failure condition (due to the non visibility of module* `m2`*).*

4 The Abstract Interpreter Implementation

In this section, we present an implementation of the minimal contexts computation. This implementation is based on meta-interpretation [22, 24]. A meta-interpreter for a language is an interpreter for the language written in the language itself. Meta-interpretation is widely adopted in the Prolog programming context. Some properties of the Prolog language make an easy task the construction of a program which reproduces the backward chaining or top-down evaluation strategy used by Prolog . Such a program is shown below:

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- system(A), !, A.
solve(A) :- clause(A,B), solve(B).
```

The predicate `clause/2` is a built-in Prolog predicate which provides access to program clauses.

It is worth noticing that the application of the immediate consequence operator adopted by the static analysis corresponds rather to a forward or bottom-up evaluation strategy. In the following paragraphs we introduce bottom-up meta-interpretation, and apply it for computing the minimal contexts for each predicate name of a program. We begin the presentation with a description of the language used.

The forward rule language Consider a program composed by a set of rules of the form: `H <= B` where `H` is an atom which is termed head and `B` is a conjunction of atoms termed body. Facts are represented by rules of the form `H <= true`.
Informally, the procedural semantic of this kind of rules is as follows: if the conjuncts appearing in the body belong to the set of known facts, then `H` is added to the set of known facts. This criteria is repeatedly applied until no new facts are generated. In order to get an actual implementation of the forward rule language, we build a forward engine on Prolog. This engine allows to write programs expressed by rules as the previous. The Appendix A contains the code of the forward engine. This code, and all the code showed throughout this section and the Appendixes, has been written in SICTUS Prolog 2.1 on a Sun Sparc workstation.

A base forward meta-interpreter Note that the forward processing of the base meta-interpreter clauses produces infinite facts of the form: `solve(true)`, `solve((true,true))`, `solve(((true, true), true))`, ... In order to simulate in the right way the repetitive application of the immediate consequence operator until the fixed point is reached, we need a predicate that provides control over inferences [21, 5]. We call this predicate `instance/1`, and its definition is as follows:

```
instance(B) <= clause(H, B).
instance(A) <= instance((A, B)).
instance(B) <= instance((A,B)), solve(A).
```

The first rule expresses that for each clause `H <= B` in the program, the body `B` must to be solved. Note that a body rule can be a conjunction, so the second rule expresses that if a conjunction of the form `(A,B)` must to be solved, then the conjunction `A` must to be processed first. Finally, the last clause expresses that if the conjunction `(A,B)` must to be solved, and the first conjunct is already solved, then the rest of the conjunction must to be solved.
The above three rules are the starting point for a base forward meta-interpreter defined as follows:

```
solve(true).
solve((A,B)) <= (instance((A,B)), solve(A), solve(B)).
solve(A) <= (solve(B), clause(A,B)).
```

The main predicate of the meta-interpreter is `solve/1`. `solve(A)` means that it is possible to derive `A` from the program clauses and the original facts. We now describe briefly the rules that define `solve/1`. The first rule indicates that `true` is solved anytime. The second rule expresses that if a conjunction `(A,B)` must to be solved and the conjuncts `A` and `B` are already solved, then the conjunction is solved. Finally the third rule expresses that if `B` is solved, and `B` is the body of a rule of the form `A <= B`, then the head `A` is solved.

The base forward meta-interpreter can be enriched in several ways in order to obtain extensions to the original forward language. In particular, a simple modification to the above meta-interpreter allows to manage modular programs and implication goals. First of all, we need a new predicate to access program clauses defined in a module. To do this we adopt the `program/3` predicate, where `program(M,H,B)` means that the rule `H <= B` is defined in module `M`. Furthermore, we add a new argument to `solve` predicate: `solve(Ctx,A)` now means that `A` is solved in the context `Ctx`, where a context is as usual a set of modules and it is represented as a list of module names. The resulting meta-interpreter is the following:

```
solve([], true).
solve(Ctx, (A,B)) <=    (instance((A,B)),
                        solve(Ctx1, A),
                        solve(Ctx2, B),
                        union(Ctx1, Ctx2, Ctx)).
solve(Ctx, P) <=                (solve(Ctx1, B),
                        program(U,P,B),
                        union([U], Ctx1, Ctx)).
solve(Ctx, M>>A) <=     (instance(M>>A),
                        solve(Ctx1, A),
                        elim(U, Ctx1, Ctx)).
instance(B) <=          program(_, _, B).
instance(A) <=          instance((A, _)).
instance(B) <=          (instance((A,B)),
                        solve(_, A)).
instance(A) <=          instance(M >> A).
```

The `instance/1` predicate definition is modified in such a way it includes as a special case the implication goals. The last clause of this definition indicates that whenever an implication goal of the form `M>>A` is found, the goal `A` has to be solved as well. The context in which the goal `M>>A` is considered solved, is calculated by the last clause of the `solve/2` definition, which expresses that if an implication goal `M>>A` must be solved, and `A` was solved in a context `Ctx1`, then the goal `M>>A` is solved in a context $\texttt{Ctx} = \texttt{Ctx1} \setminus \texttt{M}$. This setminus operation is performed by the `elim/3` predicate. The other rules defining `solve/2` are explained in the following. The first rule indicates that the `true` is solved in the empty context. The second rule expresses that if a conjunction `(A,B)` must be solved, and `A` was solved in a context `Ctx1` and `B` was solved in a context `Ctx2`, then the conjunction `(A,B)` is solved in the context $\texttt{Ctx} = \texttt{Ctx1} \cup \texttt{Ctx2}$. The remaining rule defining `solve/2` expresses that if `B` is solved in a context `Ctx1` and `B` is an instantiation of a body rule of the form `P <= B` defined in a module `M`, then `P` is solved in a context $\texttt{Ctx} = \texttt{Ctx1} \cup \texttt{M}$.

Simplifying the original program From the static analysis point of view, we are interested only in predicate names, so the original program can be syntactically transformed into a propositional version. This transformation substitutes each atom of the form `p(a1, ..., an)` appearing in the program by the propositional symbol `pn`. The program code devoted to this transformation is reported in the Appendix B.

Calculating the minimal contexts Consider a produced fact of the form: `solve(Ctx,A)`. Due to the way in which the first argument is computed, `Ctx` contains a context which allows to derive `A`. The minimal context for a predicate `p`, or $MINSET_{\mathtt{p}}$, can be obtained from the information produced by the meta-interpreter in this way:

$MINSET_{\mathtt{p}} = \{\mathtt{C}|\mathtt{functor(A,p,\_)}, \mathtt{solve(C,A)} \wedge \ \nexists \mathtt{C'}\mathtt{such\ that}$ $\mathtt{solve(C',A)} \ \wedge \ \mathtt{C'} \subset \mathtt{C})\}$

This calculation is performed by the `select_min/2` predicate, where `select_min(S, Min)` computes the set of minimal contexts `Min` from `S`, which is the set of all possible contexts that allow to derive a particular fact. The code for `select_min/2` is the following:

```
select_min(S,Min) :- select_min(S,S,[],Min).

select_min([],_,Min,Min) :- !.
select_min([S1|R],S,Min,Min1) :- member(M, S),
          \+ (M = S1),
          subset(M,S1), !,
          select_min(R,S,Min,Min1).
select_min([S1|R],S,Min,Min1) :- select_min(R,S,[S1|Min],Min1).
```

`select_min/2` processes the list of contexts looking for contexts which are not a subset of any other context in the list through a tail recursive version (`select_min/4`). The first clause above is an initialization. The second clause considers the case in which the first context in the list is contained by another context, so the first element is discarded. The last rule is the opposite situation, so the first context is minimal and is included in the computed answer.

Putting all together Up to this moment we have considered three phases for the calculation of the minimal contexts: syntactical transformation, finding all the possible contexts, and finding the minimal contexts. This phases are controlled by the `min_set/2` predicate, where `min_set(P,S)` means that `S` is the set of minimal contexts for the predicate `P`. The code of `min_set/2` is the following:

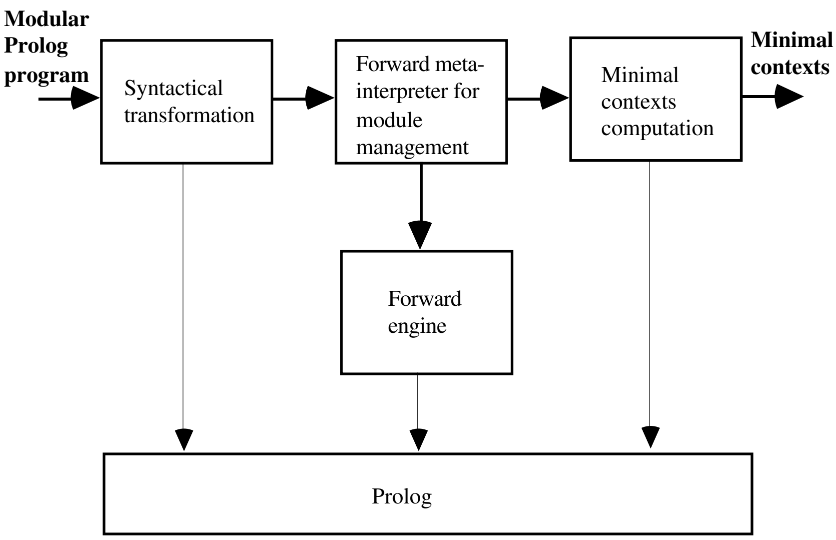

Figure 1: The software architecture

```
min_set(P,Min) :- transf,
                  go,
                  pred_sym(P),
                  setof(Ctx, fact(solve(Ctx,P)), S),
                  select_min(S,Min).
```

The first step, or syntactical transformation, is performed through the `transf/0` predicate. The `go` predicate starts the forward engine performing in this way the second step. All possible contexts are collected in variable `S` which is passed to the `select_min/2` predicate to perform the last step.
The relationship between the software components (represented as boxes) which implement each phase is shown in figure 1.

Example 4.1 *Let us consider again example 2.1 and 3.6. First of all, we show the transformed program. It is represented by the* `program/3` *predicate, where a fact of the kind* `program(M,H,B)` *means that the module* `M` *contains the clause* `H <= B`.

```
program(m1,delete_or_append3,m4>>r3).
program(m2,delete3,delete3).
program(m4,r3,delete3).
program(m4,r3,append3).
program(m4,delete3,delete3).
program(m4,append3,append3).
program(m2,delete3,true).
```

```
        program(m3,append3,true).
```

Then, the corresponding minimal contexts are returned for the following goal:

```
?- min_set(P,S).
        P = d_o_a3,
        S = [[m1,m3],[m1,m2]]
        P = delete3,
        S = [[m2]]
        P = r3,
        S = [[m4,m3],[m4,m2]]
        P = append3,
        S = [[m3]]
```

5 Refining the Analysis

In this section, we show how the abstract analisys scheme presented in section 3 (and consequently its implementation, see section 4) can be refined, in order to achieve better performances. The idea is to compute the set of minimal contexts needed for a (possibly) successful clause application rather than for a predicate call. Thus, the perspective moves from predicate to clauses. We first refines the abstract analisys schema (in a rather informal way and through examples) and then present the implementation of the new abstract interpreter.

5.1 Clause Specific Analysis

In fact, let us consider the case of several alternative clauses for the same predicate p. In this case, the analisys could be more specific: instead of looking for the minimal contexts associated with predicate p, it could return a set of minimal contexts each associated with a distinct clause of p (*i.e.*, having the predicate p in its head). This could be easily achieved without any modification of the abstract analisys framework described above, by simply renaming program clauses. The idea is to identify each clause of a program with a unique name. To this purpose, we label each clause C with the information about the module in which it is defined, and the relative position of the clause C in that module.

For instance, let P be a program, and q a predicate. Let us suppose that P provides N definition for q, distributed onto several modules $m_1, m_2, \ldots m_k$, $k \leq N$. We associate with each clause $C_q = q :- \ldots$ defined in the module m_i the two indexes i and j, where: i is the index that identifies the i-th module m_i of P, and j represents the position of C_q in the module m_i. The clause is therefore renamed as $q_{i,j} :- \ldots$.

We take care of this clause renaming when performing the abstract analisys, in order to obtain more detailed information about which clauses could be successfully applied. The framework presented in section 3 does not vary provided that we now interpret the notion of truth (see definition 2.5) in a slightly different way. In particular, the truth of an atom A in a context c with respect to an interpretation I has to be restated as follows: $c \models_I A \quad iff \quad \exists i,j : \langle c, A_{i,j}\rangle \in I$ where j and j are indexes used for renaming.

In this way, the abstract interpretation returns the set of minimal contexts associated with each program clause. Let us focus on the set of program clauses $\{q_{i,j} : - \ldots \mid (i \in [1,\ldots,k],\ j \in [1,\ldots,N])\}$ for predicate q, and let $L_{i,j}$ the set of minimal contexts associated with $q_{i,j} : - \ldots$ by the abstract analysis. When a call to q is raised, if the current context is not a superset of some of the minimal contexts in $L_{i,j}$, then clause $q_{i,j} : - \ldots$ is not applied. This allow us to further reduce the backtracking overhead in all cases when several alternative clauses are available for the predicate call q, but some of them do not have support (in terms of modules at disposal in the current context) for possibly succeeding. As a drawback, the number of checks increases. However, assuming an implementation scheme based on bit-vectors (see section 3.2), the performance gain should be evident.

Example 5.1 *Let us consider, for instance, the example 2.1. That program could be rewritten as follows:*

```
m1:     delete_or_append_1_1:-m4>>r(X,Y,Z).
m2:     delete_2_1(X,[],[]).
        delete_2_2(X,[X|Y], Z):-delete(X,Y,Z).
m3:     append_3_1([],X,X).
m4:     r_4_1([X],Y,Z):-delete(X,Y,Z).
        r_4_2(X,Y,Z):-append(X,Y,Z).
        delete_4_1(X, [U|Y], [U|Z]):-delete(X,Y,Z).
        append_4_1([X|U],V,[X|W]):-append(U,V,W).
```

If we apply abstract analysis to this renamed version of the program, the $T_P^\alpha \uparrow \omega$ gives the following minimal contexts:

$$T_P^\alpha \uparrow \omega = \{\langle\{m_1, m_2\}, delete\_or\_append_{1,1}\rangle, \langle\{m_1, m_3\}, delete\_or\_append_{1,1}\rangle$$

$$\langle\{m_2\}, delete_{2,1}\rangle, \langle\{m_2\}, delete_{2,2}\rangle, \langle\{m_2, m_4\}, delete_{4,1}\rangle,$$

$$\langle\{m_3\}, append_{3,1}\rangle, \langle\{m_3, m_4\}, append_{4,1}\rangle$$

$$\langle\{m_2, m_4\}, r_{4,1}\rangle, \langle\{m_3, m_4\}, r_{4,2}\rangle\}$$

Let us consider again the top goal:

```
:- m1>>m3>>delete_or_append([g],[a,b,c,d,e,f], X).
```

In this case we obtain a lower number of backtracking phases (one, instead of three), with only one more check (six checks instead of five) with respect to the execution sketched in example 3.6 in section 3.2.

5.2 The Implementation

The rewriting of head rules required by the refined analysis is implemented prior the syntactic transformation phase. Each head clause of the form `p(a1, ..., an)` is transformed into a labeled predicate `pn_m_i`, where `n` is the arity of `p`, `m` is the module name in which the corresponding clause appears and `i` is the position of this rule in `m`. For example, if the rule:

```
p(X) :- q(X).
```

is the second one in module `m4`, the transformed rule will be:

```
p1_m4_2 :- q_1.
```

The code of this new transformation process is shown in Appendix C. Since the transformed program has labeled predicates only in the rule heads, we need to modify unification in order to capture instantiations between labeled and non labeled predicates. The meta-interpreter that incorporates the modified unification is shown below:

```
solve([], true).
solve(Ctx, (A,B)) <= (instance((A,B)),
                      solve(Ctx1, A1),
                      solve(Ctx2, B1),
                      ext_unif(A, A1),
                      ext_unif(B, B1),
                      union(Ctx1, Ctx2, Ctx)).
solve(Ctx, P) <= (solve(Ctx1, B),
                  program(U,P,B1),
                  ext_unif(B, B1),
                  union([U], Ctx1, Ctx)).
solve(Ctx, U>>A) <= (instance(U>>A),
                     solve(Ctx1, A1),
                     ext_unif(A,A1),
                     elim(U, Ctx1, Ctx)).

instance(B) <= program(_, _, B).
instance(A) <= instance((A, _)).
```

```
instance(B) <= (instance((A,B)),
                solve(_, A1),
                ext_unif(A,A1)).
instance(A) <= instance(_ >> A).
```

Comparing this meta-interpreter with the previous one reported in section 4, we note that each time a unification is required, an explicit call to the `ext_unif/2` predicate is made. `ext_unif(A,A1)` succeeds when the terms `A` and `A1` unifies according to the following criteria: `A` and `A1` correspond the same predicate name and the same arity. The code of the `ext_unif` predicate is shown below:

```
ext_unif((A,R), (A1,R1)) :- !, ext_unif(A,A1),
                               ext_unif(R,R1).
ext_unif(S_body, P_body) :- \+ S_body = (_,_),
                               \+ P_body = (_,_),
                               ext_functor(S_body, N),
                               ext_functor(P_body, N).

ext_functor(_>>A, N) :- !, get_name(A,N).
ext_functor(A,N) :- get_name(A,N).

get_name(A, N) :- name(A, L),
                  append(L1, [95|_], L), !,
                  name(N, L1).
get_name(A, A).
```

The first rule specifies that two conjunctions unifies if each conjunct unifies. The second rule specifies that two atoms unifies if they have the same predicate name and the same arity. This is achieved through predicate `ext_functor` and `get_name`.

Example 5.2 *Let us consider again example 2.1 and 5.1. The transformed program is:*

```
program(m1,delete_or_append3_m1_1,m4>>r3).
program(m2,delete3_m2_1,delete3).
program(m4,r3_m4_1,delete3).
program(m4,r3_m4_2,append3).
program(m4,delete3_m4_3,delete3).
program(m4,append3_m4_4,append3).
program(m2,delete3_m2_2,true).
program(m3,append3_m3_1,true).
```

The corresponding minimal contexts are as follows:

```
?- min_set(P,S).
        P = delete_or_append3_m1_1,
        S = [[m1,m3],[m1,m2]]
        P = delete3_m2_1,
        S = [[m2]]
        P = r3_m4_1,
        S = [[m4,m2]]
        P = r3_m4_2,
        S = [[m4,m3]]
        P = delete3_m4_3,
        S = [[m4,m2]]
        P = append3_m4_4,
        S = [[m4,m3]]
        P = delete3_m2_2,
        S = [[m2]]
        P = append3_m3_1,
        S = [[m3]]
```

6 Conclusions

We focused on modular logic languages with implication goals and dynamic scope rules, and discussed how to improve them by using static analysis techniques. In particular, we have applied abstract interpretation in order to reduce the overhead due to failing, useless computations. The results of the analysis are exploited by an underlying WAM-based implementation (described in [7] and which using a bit vector representation for contexts of modules) in order to reduce the overhead due to the certainly failing computations.

The abstract interpreter has been implemeted in SICStus Prolog via meta-interpretation techniques. The underlying extended WAM has been written in C, and runs on different platforms. The compiler is written in Prolog, as extension of that implemented by P. Van Roy [26]. Future work will be devoted to test the impact of the analysis on the considered WAM-based implementation on the basis of more significant examples. It is worth to notice, however, that the abstract interpreter here presented could be similarly used to improve other implementations such as those described in [13, 12]. The impact of the analysis on these systems is subject for future work.

Acknowledgments

We would like to thank Francesco Battini for his help in implementing the extended WAM. This work has been partially supported by C.N.R. "Project on Program Analysis and Transformation ANATRA".

References

[1] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[2] A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. Contributions to the semantics of open logic programs. In H. Tanaka, editor, *Proc. FGCS'92 Int. Conference*, pages 570–580, 1992.

[3] A. Brogi, E. Lamma, and P. Mello. Hypothetical Reasoning in Logic Programming: A Semantics Approach. *Information Processing Letters*, 36:285–291, 1990.

[4] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1–21, 1992.

[5] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.

[6] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19-20:443–502, 1994.

[7] A. Ciampolini, E. Lamma, and P. Mello. Improving the efficiency of dynamic modular languages. In *Proceedings of Gulp-Prode 95 Int. Conference*, pages 347–358, 1995.

[8] M. Codish, S.K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[9] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13:103–180, 1992.

[10] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM Press, 1989.

[11] L. Giordano, A. Martelli, and G.F. Rossi. Local definitions with static scope rules in logic languages. In *Proc. FGCS'88 Int. Conference*, pages 389–396, 1988.

[12] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *Proc. 8th Int. Conference on Logic Programming*, pages 871–886. The MIT Press, 1991.

[13] E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, 14:187–222, 1992.

[14] P. Mancarella and D. Pedreschi. An algebra of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1006–1023. The MIT Press, 1988.

[15] L. T. McCarty. Clausal Intuitionistic Logic I: Fixed point Semantics. *Journal of Logic Programming*, 5(1):1–31, 1988.

[16] D. Miller. A theory of modules in logic programming. In *Proceedings of Symposium on Logic Programming*, pages 106–114, 1986.

[17] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[18] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

[19] L. Monteiro and A. Porto. A Language for Contextual Logic Programming. In J.W. de Bakker K.R. Apt and J.J.M.M. Rutten, editors, *Logic Programming Languages, Constraints, Functions and Objects*, pages 115–147. The MIT Press, 1993.

[20] R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

[21] V.A. Pagone. Una propuesta de evaluacion parcial para la programacion logica forward. Technical report, 1995.

[22] S. Safra and E. Shapiro. Metainterpreters for real. In H.-J. Kugler, editor, *Information Processing 86*, pages 271–278. North-Holland, 1986.

[23] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon Inc., 1986.

[24] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[25] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[26] P. van Roy. A Prolog Compiler for the PLM. Technical report, Computer Science Division (EECS), University of California, Berkeley, Ca., 1984.

[27] D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.

Appendix A

The forward engine code is reported in the following.

```
solve(true) :- !.
        solve(A) :-
        \+ (A = _<=_),          % add a fact
        \+ fact(A),             % loop detection
        assert(fact(A)),
```

```
        fail.
solve(C <= (A,R)) :-            % add a rule with a conjunction body
        solve(C <= R).
solve(C <= A) :-                % add a rule with an atom in the body
        \+ (A = (_,_)),
        eval(A),
        \+ fact(C),             % loop detection
        solve(C).
solve(A) :-                     % use a fact which is the first one
                                % in a body rule
        \+ (A = _<=_),
        rule(C <= (A,R)),
        solve(C <= R).
solve(A) :-                     % use a fact which is alone in a body rule
        \+ (A = _<=_),
        rule(C <= A),
         \+ fact(C),            % loop detection
        solve(C).

eval((A,B)) :-
        !, eval(A),
        eval(B).
eval(not(A)) :-
        !, \+ solve(pregunta(A)).
eval(A) :-
        (system(A);
         usr_built_in(A)), !,
        call(A).
eval(A) :-
        fact(A).
```

Appendix B

The code for the syntactical transformation over the original program is the following.

```
transf :- source_program(U, H, B),
                      transf_atom(H, H1),
                      transf_body(B, B1),
                      assert(meta_data(program(U, H1, B1))),
                      \+ pred_sym(H1),
                      assert(pred_sym(H1)),
                      fail.
transf.

transf_atom(true, true) :- !.
```

```
transf_atom(M>>A, M>>A1) :- !, transf_atom(A, A1).
transf_atom(A, A1) :- functor(A, F, N),
                      gen_pred(F, N, A1).

gen_pred(P, N, P1) :- name(P, L),
                      name(N, LN),
                      append(L, LN, L1),
                      name(P1, L1).

transf_body(A, A1) :- \+ (A = (_,_)), !,
                      transf_atom(A, A1).
transf_body((A,B), (A1, B1)) :- transf_atom(A, A1),
                                transf_body(B, B1).


select_min(S, Min) :- select_min(S, S, [], Min).

select_min([], _, Min, Min) :- !.
select_min([S1|R], S, Min, Min1) :- member(M, S),
                                    \+ (M = S1),
                                    subset(M, S1), !,
                                    select_min(R, S, Min, Min1).
select_min([S1|R], S, Min, Min1) :- select_min(R, S, [S1|Min], Min1).
```

Appendix C

The modified syntactic transformation implementation is the code which follows.

```
transf :- init_transf,
          source_program(U, H, B),
          get_unit_info(U, C_count),
          transf_head(H, U, C_count, H1),
          transf_body(B, B1),
          assert(meta_data(program(U, H1, B1))),
          \+ pred_sym(H1),
          assert(pred_sym(H1)),
          fail.
transf.

init_transf :- retract(meta_data(_)), fail.
init_transf :- retract(pred_sym(_)), fail.
init_transf :- retract(unit_info(_,_)), fail.
init_transf.

% we implicitly suppose the unicity of module names, so the module name
% is used as an identificator for the module
```

```
get_unit_info(U, 1) :- \+ unit_info(U, _), !,
                       assert(unit_info(U, 1)).
get_unit_info(U, C_count) :- retract(unit_info(U, C_count1)),
                             C_count is C_count1 + 1,
                             assert(unit_info(U, C_count)).

transf_head(H, U, C, H1) :- functor(H, F, A),
                            name(F, LF), name(A, LA),
                            name(U, LU), name(C, LC),
                            append(LF, LA, L1), append(L1, [95], L2),
                            append(L2, LU, L3), append(L3, [95], L4),
                            append(L4, LC, L5), name(H1, L5).

transf_atom(true, true) :- !.
transf_atom(M>>A, M>>A1) :- !, transf_atom(A, A1).
transf_atom(A, A1) :- functor(A, F, N),
                      gen_pred(F, N, A1).

gen_pred(P, N, P1) :- name(P, L),
                      name(N, LN),
                      append(L, LN, L1),
                      name(P1, L1).

transf_body(A, A1) :- \+ (A = (_,_)), !,
                      transf_atom(A, A1).
transf_body((A,B), (A1, B1)) :- transf_atom(A, A1),
                                transf_body(B, B1).
```