# A FRAMEWORK FOR THE ABSTRACT INTERPRETATION OF LOGIC PROGRAMS

Maurice Bruynooghe*

## Abstract

A general and intuitively appealing framework is developed for the abstract interpretation of logic programs. An algorithm is given to build an abstract AND/OR tree. The algorithm uses a small set of primitive operations. Conditions for the primitive operations and the abstract domain guaranteeing termination of the algorithm and correctness of the abstract AND/OR tree are proved. The elegance of the method is illustrated with a mode inferencing application.

* Computer Science Department, K. U. Leuven, Research Associate supported by Belgian National Fund for Scientific Research.

## 1. Introduction

Declarative semantics is one of the biggest assets of logic programming. However, also the procedural behavior of programs has some importance. Often, it is useful to know which substitutions are possible, at run-time, over the variables in a procedure call before or after its execution. A compiler can use such information to optimize the code, a programmer to judge the procedural behavior : will the program terminate, will the performance be acceptable,... . A familiar example of such procedural behavior is mode information.

The procedural reasoning by programmers has been formalized in an inductive assertion method for logic programs by Drabent and Maluszynski [4]. It is a manual approach, taking into account the program at hand, one can express properties with as much precision as one wishes, but of course, the more precision, the harder the proof of correctness.

Abstract interpretation is complementary to the inductive assertion method. It allows to generate automatically correct statements about the substitutions which can be expected at different program points during the execution of a certain class of queries. An abstract interpreter must be designed and proved correct for a certain class of properties (the subtler the properties, the harder the design and correctness proof). Once finished, the abstract interpreter can automatically produce true statements, related to the properties it has been designed for, and this for any given program.

Attempts to formalize the abstract interpretation of logic programs have been undertaken by Mellish [8] and Jones and Sondergaard [5]. In our opinion, they lack intuitive appeal and the gap with applications remains substantial. [1] contains a very rough sketch of a general framework for the abstract interpretation of logic programs, also, three applications, namely type inferencing, mode inferencing and compile-time garbage collection are sketched. In this paper, we describe the framework with more mathematical rigour and greater precision and include the necessary proofs.

In a very naive approach to the problem of abstract interpretation, one could run, for a given program, all possible queries. Each time a certain call is executed, one could collect the concrete substitutions over the variables of the call which hold prior and past the execution of the call. For all but the simplest programs this is an impossible approach as one would have to run an infinite number of queries. The core idea of abstract interpretation is to work with abstract substitutions (each abstract

substitution representing a possibly infinite set of concrete substitutions) and to execute the program with an abstract interpreter able to manipulate such abstract substitutions.

It is well known that, while executing a query in a top-down fashion, an AND/OR tree is traversed (in actual Prolog implementations, finished OR-branches disappear due to the backtracking). In a similar vein, our generic abstract interpretation algorithm will construct an abstract AND/OR tree adorned with abstract substitutions. The purpose is to obtain a *correct* abstract AND/OR tree. This means it must be possible to follow the execution of any concrete query on the abstract AND/OR tree. A concrete substitution appearing at some stage of the execution must be in the set of concrete substitutions which is represented by the corresponding abstract substitution.

As the execution of a concrete query for a recursive program can traverse an AND/OR tree of any depth (also non terminating programs must be correctly modeled by the abstract AND/OR tree), recursion poses a problem. Abstract interpretation can construct only a finite AND/OR tree. To solve this problem, some recursive calls will not be expanded but the abstract substitution which holds after successful termination of such a call will be "guessed" by a fixpoint computation.

In section 2, we develop the notion of abstract substitution and give a precise description of the abstract interpretation algorithm. The algorithm is generic and uses 7 primitive operations which are application dependent. Termination of the algorithm is discussed in section 3, correctness in section 4. Having developed conditions for termination and correctness, some application independent properties of the primitive operations are discussed in section 5. An example application, mode inferencing, is developed in section 6. Concluding remarks are in section 7.

## 2. Construction of the abstract AND/OR tree

### 2.1 Abstract substitutions

In the context of this paper, a concrete substitution $\theta$ is a set $\{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$; the *domain* of $\theta$ is the set $\{V_1, \ldots, V_n\}$, the *range* is the set of variables appearing in $t_1, \ldots, t_n$ and the intersection of domain and range is empty (i.e. we restrict our attention to idempotent substitutions [6]). By *var(t)* we denote the set of variables occuring in t. An *abstract substitution* is a finite notation for a possibly infinite set of concrete substitutions over a certain set of relevant variables. This set of

variables is the *domain* of the abstract substitution; it is a superset of the union of the domains of the concrete substitutions in the set. Domains of abstract substitutions used throughout this paper will be the set of variables in (the textual representation of) a clause, a query, a clause head or a call. Conform to the usage in work on abstract interpretation, we can define the *concretization function* $\gamma$ as the function which, applied on (the notation for) an abstract substitution, yields the set of concrete substitutions.

Some *notation*: We use the symbols $\beta$ and $\delta$ for abstract substitutions, S for the domain of an abstract substitution, $\theta$ and $\sigma$ for concrete substitutions, $\Theta$ for a set of concrete substitutions and, as the reader could already notice, variable names start with an upper case. Symbols are decorated with subscripts and superscripts as the need arises. For example we write $\beta_S$ to stress that the domain of the abstract substitution $\beta$ is S, in a slight abuse of notation, we sometimes write $\beta_{S_X}$ to stress that the domain $S_X$ is the set $\{X_1, \ldots, X_n\}$. We also often use the notation $R_S(\theta)$ for $\theta$ restricted to the variables of S, i.e. $R_S(\theta) = \{X \leftarrow t \mid X \in S \wedge X \leftarrow t \in \theta\}$. Notice that for any term t : $t\,\theta = t\,R_S(\theta)$ if S is the set of variables occurring in t.

**Examples**

In the following examples, the domain is {X,Y}

$\beta$ = {X ← ground,Y ← unbound}

$\gamma(\beta)$ = {{X ← t} : ground(t)}

$\beta$ = {X ← ground,Y ← var}

$\gamma(\beta) = \{\{X \leftarrow t\} : \text{ground}(t)\} \cup \{\{X \leftarrow t_1, Y \leftarrow t_2\} : \text{ground}(t_1) \wedge \text{var}(t_2)\}$.

Remind, the intersection of domain and range of concrete substitutions must be empty. In these examples, the components of the abstract substitutions are generators specifying an allowed set of terms for each variable in the domain. Their notation resembles concrete substitutions, this need not be the case. Other components of abstract substitutions can be considered as a kind of n-ary constraints ($n > 1$) limiting the set of values which can coexist :

$\beta$ = {X and Y have the same value} :

$\gamma(\beta) = \{\{X \leftarrow Y\}, \{Y \leftarrow X\}\} \cup \{\{X \leftarrow t, Y \leftarrow t\}\}$.

Sometimes one prefers to represent the absence of a constraint :

$\beta$ = {X and Y are allowed to share a free variable}

$\gamma(\beta) = \{\{X \leftarrow t_1, Y \leftarrow t_2\}$: no constraint}

whereas

$\beta = \{\ \}$

$\gamma(\beta) = \{\{X \leftarrow t_1, Y \leftarrow t_2\} : \text{var}(t_1) \cap \text{var}(t_2) = \phi\}$.

Two abstract substitutions deserve special attention, namely $\bot$ and $\top$. $\gamma(\bot) = \phi$, the empty set; as label in a program point, it indicates that control can never reach that point. $\gamma(\top)$ is the set of all possible substitutions over the domain. For the domain {X,Y} :

$\gamma(\top) = \{\epsilon\} \cup \{\{X \leftarrow t\}\} \cup \{\{Y \leftarrow t\}\} \cup \{\{X \leftarrow t_1, Y \leftarrow t_2\}\}$

It may be helpful to the intuition of the reader that, up to now, we have not developed applications where an abstract substitution refers to *names* of variables outside its domain.

Developing an abstraction over a set of variables consists of making an appropriate choice of sets of concrete substitutions to be included in what is called the *abstract domain*. Our abstract domains always include $\bot$ and $\top$.

Sets of concrete substitutions are partially ordered by the set inclusion operator $\subseteq$. Each pair $\Theta_1,\Theta_2$ has a least upper bound, their union and a greatest lower bound, their intersection. Thus the partial order over sets of concrete substitutions is a lattice. This lattice is the concrete domain of the interpretation.

The order over the concrete domain induces an order between abstract substitutions over the same domain of variables. (In the sequel, such abstract substitutions are called an abstract domain). This order, denoted $\leqslant$ is defined as follows : $\beta_S \leqslant \delta_S$ iff $\gamma(\beta_S) \subseteq \gamma(\delta_S)$. This results in a partially ordered set (poset) with a maximal element $\top_S$ and a minimal element $\bot_S$. Notice that, due to the definition of $\leqslant$, $\gamma$ is monotone. $\beta \geqslant \delta$ and $\beta \leqslant \delta$ implies $\beta = \delta$; abstract substitutions describing the same set of concrete substitutions are considered as identical. The partial order trivially extends to n-tuples : $(\beta_1, \ldots, \beta_n) \leqslant (\delta_1, \ldots, \delta_n)$ iff $\beta_i \leqslant \delta_i$ for $1 \leqslant i \leqslant n$.

Complementary to the concretization function $\gamma$, we can define an abstraction function $\alpha$. $\alpha$ maps a set of concrete substitutions $\Theta$ in an abstract substitution $\beta = \alpha(\Theta)$ such that $\gamma(\beta) \supseteq \Theta$ and that $\beta$ is minimal, i.e. $\forall\delta$, $\delta < \beta \rightarrow \gamma(\delta)$ nlsupset $\Theta$. This definition implies that $\alpha(\gamma(\beta)) = \beta$ and $\gamma(\alpha(\Theta)) \supseteq \Theta$. These properties are known as the adjointness of $\alpha$ and $\gamma$.

With a finite abstract domain, there always exists a terminating algorithm, which, given $\Theta$, computes $\alpha(\Theta)$ (by enumeration of the elements).

In some cases, $\alpha$ can be such that $\alpha(\Theta)$ is least, i.e. $\forall\delta$, $\gamma(\delta) \supseteq \Theta$ implies $\delta \geqslant \alpha(\Theta)$. With this property, $\alpha$ is monotone.

Proof

(1)$\Theta_1 \supseteq \Theta_2$ assumption

(2)$\gamma(\alpha(\Theta_1)) \supseteq \Theta_1$ from definition of $\alpha$

(3)$\gamma(\alpha(\Theta_1)) \supseteq \Theta_2$ (1), (2) and transitivity of $\supseteq$

(4)$\alpha(\Theta_1) \geqslant \alpha(\Theta_2)$ (3) and "least" property with $\delta = \alpha(\Theta_1)$.

(5)$\Theta_1 \supseteq \Theta_2 \rightarrow \alpha(\Theta_1) \geqslant \alpha(\Theta_2)$ : monotonicity of $\alpha$ follows from (1) and (4).

□

This "least" property also implies that $\alpha(\gamma(\beta_1) \cup \gamma(\beta_2))$ is the least upper bound of $\beta_1$ and $\beta_2$. this means that every pair of abstract substitutions has a lub, also there is a least element $\perp$ , thus the abstract domain is a lattice. However, the abstract domain being a lattice is not a sufficient condition for $\alpha$ to have the "least" property and to be monotone.
The abstraction function $\alpha$ plays only a minor role in what follows.

## 2.2 An informal sketch of abstract interpretation

It is our purpose to decompose the abstract interpretation process into a number of simple steps which are easy to grasp and to prove correct. To find out which operations are appropriate, let us look at the different steps in executing a call P, at the concrete substitutions we can distinguish at different stages of the execution of P and at corresponding abstract substitutions in the abstract execution of P.

Fig. 1 shows a fragment of the familiar AND-tree which results from executing the call P. Fig. 2 shows the corresponding abstract AND/OR tree. Both trees are adorned with substitutions occurring at different stages. Notice that the concrete AND-tree can also represent unfinished computations.

The call P occurs either in the body of a clause or in the initial query. This clause/query contains a set of variables $S_X$ in its textual representation. A first substitution we can distinguish is the substitution which holds prior to the execution of P over the set $S_X$, in Fig. 1, it is called $\theta_{init}$, the corresponding abstract substitution is $\delta_{S_X}^{init}$. $\theta_{init}$ should be a member of $\gamma(\delta_{S_X}^{init})$. Notice that $\theta_{init} = R_{S_X}(\sigma_1 \text{ na } \sigma_2 \text{ na } \cdots \text{ na } \sigma_n)$ with $\sigma_1, \ldots, \sigma_n$ the mgu's of the resolution steps leading from the initial query to the current goal. Our purpose is to obtain $\theta_{next}$ and $\delta_{S_X}^{next}$, the substitutions over $S_X$ after successfully solving the call P.

The call P contains in general only a subset of the variables in $S_X$, let $S_Y$ be this subset. We can also consider the substitution over $S_Y$ prior to the execution of P. $\theta_{rstr}$ is the concrete substitution with $\theta_{rstr} = R_{S_Y}(\theta_{init})$. In the corresponding abstract interpretation, we obtain $\delta_{S_Y}^{rstr}$ by applying a *restriction* operation on $\delta_{S_X}^{init}$. Correctness requires that $\theta_{rstr} \in \gamma(\delta_{S_Y}^{rstr})$. $\delta_{S_Y}^{rstr}$ is the *call-abstraction* of P.

The concrete interpretation performs unification between $P(...)\,\theta_{rstr}$ and H=P(...), the head of the $j^{th}$ clause. After this unification, we can distinguish $\theta_h^j$, the substitution over the variables of the head. As we will explain, we finally only consider calls of the form $P(Y_1, \ldots, Y_n)$ and

heads of the form $P(Z_1, \ldots, Z_n)$ with all variables distinct. This means that, with $\theta_{rstr} = \{Y_i \leftarrow t_i\}$, we have that $\theta_h^j = \{Z_i \leftarrow t_i\}$, i.e. is the result of a renaming of $\theta_{rstr}$. In the abstract interpretation, we distinguish $\delta_h^j$, it is derived from $\delta_{S_Y}^{rstr}$ by a *renaming* operation. Notice that we have an OR-node in the abstract interpretation because it has to cope with all alternatives for solving P(...) : we need a branch for each clause defining P. Again, correctness requires $\theta_h^j \in \gamma(\delta_h^j)$.

$\theta_1$, the substitution over the variables of the clause $H \leftarrow B_1, \ldots, B_n$ which holds prior to the execution of $B_1$ is equal to $\theta_h^j$. However, in the abstract interpretation, it is slightly different because the domain needs to be extended with the variables only occurring in the body. An *initialization* operation applied on $\delta_h^j$ gives $\delta_1$. Correctness again requires that $\theta_1 \in \gamma(\delta_1)$.

Now, we are in a position equivalent to the starting position (we will consider built-ins later) : a call $B_1$ with substitution $\theta_1$ over the variables of the clause prior to execution $B_1$. Thus we can execute $B_1$ yielding $\theta_2$ and $\delta_2$ as substitution over the variables of the clause after the interpretation of $B_1$ which are also the substitutions prior to the execution of $B_2$. Similarly, we can obtain the pairs $\theta_3, \delta_3, \ldots, \theta_{n+1}, \delta_{n+1}$. Preservation of correctness requires $\theta_{n+1} \in \gamma(\delta_{n+1})$. $\theta_{succ}^j$, the substitution over the variables of the head after successfully executing the body is equal to $R_{S_Z}(\theta_{n+1})$. Applying the *restriction* operation on $\delta_{n+1}$ gives $\delta_{succ}^j$, the corresponding abstract substitution. In the concrete interpretation, $\theta_{succ}$ is the substitution over the variables of the call after successfully executing the body. Again, with a call of the form $P(Y_1, \ldots, Y_n)$ and a head of the form $P(Z_1, \ldots, Z_n)$, we have that $\theta_{succ}^j = \{Z_i \leftarrow t_i\}$ iff $\theta_{succ} = \{Y_i \leftarrow t_i\}$, i.e. both substitutions are renamings of each other. Thus one could obtain $\delta_{S_Y}^j$ from $\delta_{succ}^j$ by renaming. However, correctness does not imply minimality, $\gamma(\delta)$ may contain some $\theta$'s which cannot occur during execution of a query. Each step during the abstract interpretation of the body may have created such "rubbish". This final step creates an oppurtunity to eliminate some of this rubbish : elements in $\gamma(\delta_{succ}^j)$ which are not an instantiation of some element in $\gamma(\delta_h^j)$ can be eliminated. Therefore, we define an operation called *backward unification*. It possibly uses $\delta_h^j$ to reduce the size of $\delta_{succ}^j$, then it renames the variables to obtain $\delta_{S_Y}^j$. (This observation is due to W. Drabent). Of course, to be correct, it must be that $\theta_{succ} \in \gamma(\delta_{S_Y}^j)$. Notice that $\theta_{succ} = R_{S_Y}(\theta_{rstr}$ na $\sigma_{m+1}$ na $\cdots$ na $\sigma_n)$ with $\sigma_{m+1}, \ldots, \sigma_n$ the mgu's of the resolution steps solving P.

The abstract interpretation not only describes the particular execution shown in Fig. 1 but must describe all possible executions for which $\theta_{init} \in \gamma(\delta_{S_X}^{init})$. In other words, it must also deal with solutions obtained by applying other clauses. In the abstract interpretation, an *upper bound* operation applied on $\delta_{S_Y}^1, \ldots, \delta_{S_Y}^k$ yields $\delta_{S_Y}^{succ}$. $\delta_{S_Y}^{succ}$ is the *success-abstraction* of the call P.

Finally, $\theta_{next}$ is the substitution over the variables of $S_X$ after successfully solving P. We have that $\theta_{next} = R_{S_X}(\theta_{init}$ na $\sigma_{m+1}$ na $\cdots$ na $\sigma_n)$. In the abstract interpretation, we obtain $\delta_{S_X}^{next}$ by applying an *extension* operation on $\delta_{S_X}^{init}$ and $\delta_{S_Y}^{succ}$. To be correct, $\theta_{next} \in \gamma(\delta_{S_X}^{next})$ must hold.

We have only discussed renaming and backward unification for the simple case where arguments of call and head are distinct variables. To achieve this one has to consider = as a built-in and to introduce calls of the form $X = Y$ and $X = f(Y_1, \ldots, Y_n)$, $n \geqslant 0$, f a functor. Concrete execution of such a call maps the initial substitution $\theta$ into $R_S(\theta$ na $\sigma)$ with $\sigma$ the mgu of left and right handside of the equality and S the set of variables in the call. The corresponding operation in the abstract domain is the *abstract interpretation of built-ins*, it maps $\beta$ into $\delta$; to be correct, if $\theta \in \gamma(\beta)$ then $R_S(\theta$ na $\sigma) \in \gamma(\delta)$. The advantage of this approach is that it yields simpler abstract operations : parameter passing is separated from unification. Also, actual Prolog compilers break unification up into such simpler steps and optimizing compilers may shift around these simple instructions to achieve better code. This is another reason for having these simpler steps explicit during abstract interpretation.

The transformation of the source program making unification explicit is called *normalization*. The question arises where to put these unification built-ins. For a call, one can put them directly in front of the call or directly after the call. For a head, one can put them at the beginning or at the end of the body. Which solution is preferable depends on the application at hand. For example, in the compile-time garbage collection application [1], certain output producing unifications are placed at the end of the body (based on a local analysis, they are moved in front of the last call to preserve last call optimizations). Because abstract interpretation of the body may cause a lot of rubbish in the final abstract substitution $\delta_{S_Y}^{next}$, it can be advantageous to perform abstract interpretation of certain built-ins twice : to repeat them after the call when they were put in front of it, to repeat them at the end of the body when they were put at the beginning of it. This *repeat previous call* strategy may also be applied on other

procedure calls to improve the precision. Repeating a previous call does not affect the concrete substitution but may yield a more precise (smaller) abstract substitution. This can be the case when the current abstract substitution over the variables over a previous call is different from the one used for its abstract interpretation.
Of course, this has a cost and stronger criteria are desirable, e.g. the abstract substitution over the input arguments of the call should be smaller than the original one. Also, one should avoid infinite loops. A further discussion of this is outside the scope of the current paper; it is too dependent on the application at hand.

To summarize, we have broken up the process of abstract interpretation into a number of *primitive operations* (Fig. 2). These are :

1. a *restriction* operation mapping $\delta_{S_X}^{init}$ into $\delta_{S_Y}^{rstr}$ and $\delta_{n+1}$ into $\delta_{succ}^{j}$.
2. a *renaming* operation mapping $\delta_{S_Y}^{rstr}$ into $\delta_{h}^{j}$
3. an *initialization* operation mapping $\delta_{h}^{j}$ into $\delta_{1}$
4. *backward unification* mapping $\delta_{succ}^{j}$ and $\delta_{h}^{j}$ into $\delta_{S_Y}^{j}$
5. an *upper bound* operation mapping $\delta_{S_Y}^{1}, \ldots, \delta_{S_Y}^{k}$ into $\delta_{S_Y}^{succ}$
6. an *extension* operation mapping $\delta_{S_X}^{init}$ and $\delta_{S_Y}^{succ}$ into $\delta_{S_X}^{next}$
7. *abstract interpretation of built-ins* $X = Y$, $X = f(Y_1, \ldots, Y_N)$ mapping $\delta_{S_Y}^{rstr}$ into $\delta_{S_Y}^{succ}$.

Also, we have mentioned the repeat previous call strategy as a way to improve the precision of abstract substitutions.

The choice of these primitive operations may look somewhat arbitrarily. Indeed, it is, one could have defined a different set. As will become clear in the following sections, the requirements imposed on the operations to guarantee termination and correctness are generic, they do not depend on the particular definition of the primitives. What matters is the following :

— Given $\delta_{S_X}^{init}$, one must be able to characterize the abstraction of the call which is described by the pair $\delta_{S_Y}^{rstr}$, P(...) (the call need not be in normal form). This is important for the proper handling of recursion.

— One must be able to obtain $\delta_1$ the abstract substitution over the variables of the applied clause prior to the execution of the body.

— Given $\delta_{n+1}$ and similar abstract substitutions for the other clauses, one must be able to derive $\delta_{S_Y}^{succ}$. Also the pair $\delta_{S_Y}^{succ}$, P(...) which characterizes the call after successful completion is needed for the paper handling of recursion.

— One must be able to obtain $\delta_{S_X}^{next}$.

Any set of primitives allowing for the computation of these abstract substitutions is appropriate if it can be proven that they satisfy the conditions for termination and correctness as described in the next section.

## 2.3 Generic abstract interpretation algorithm

Without loss of generality we can assume a query $Q(X_1, \ldots, X_n)$ and an abstract substitution $\delta$ over $\{X_1, \ldots, X_n\}$ are given. The algorithm below constructs an AND/OR tree for the given query. As shown in Fig. 2 an OR-node is a call adorned with four abstract substitutions, two describing the state before the call, two describing the state after the call. Before and after the call, we have an abstract substitution describing how the variables in the call are instantiated and one describing how *all* variables of the clause/query are instantiated. The call has a child for each matching clause. Each child is an AND-node containing the head and adorned with three abstract substitutions. The first describing how the variables in the head are instantiated after unification between call and head; the second describes the same variables but after executing the body and the last how the variables in the call are instantiated after executing the body. The AND-node has a child for each call $B_l$ in the body of the applied clause.

The AND/OR tree is initialized with an OR-node $(\delta,-,Q(X_1, \ldots, X_n),-,-)$. $\delta$ being the abstract substitution over all variables in the query and $Q(X_1, \ldots, X_n)$ being the initial call. The abstract interpretation algorithm has to build the complete AND/OR tree and to compute the three other abstract substitutions in the root node.

Now we give the algorithm for processing an OR-node $(\delta_{S_X}^{init}, \delta_{S_Y}^{rstr}, P(Y_1, \ldots, Y_n), \delta_{S_Y}^{succ}, \delta_{S_X}^{next})$ with $\delta_{S_X}^{init}$ and P(...)[1] given (Fig. 2).

1. Apply the *restriction* operation on $\delta_{S_X}^{init}$ to obtain $\delta_{S_Y}^{rstr}$ over the variables

1. For the built-ins, the arguments are not necessarily variables.

$\{Y_1, \ldots, Y_n\}$ of the call.

2. Processing the call $P(Y_1, \ldots, Y_n)$; we distinguish between :

   a. the call is a built-in. Perform *abstract interpretation of the built-in* to obtain $\delta_{S_Y}^{succ}$, the abstract substitution over its variables after executing the call.

   b. the call has no ancestor node $(\delta_{S_U}^{init}, \delta_{S_V}^{rstr}, P(V_1, \ldots, V_n), -, -)$ in other words P is not a recursive call.

      — the tree is extended with a child for each clause defining the predicate P/n as shown in Fig. 2. (The child is an AND-node having a child for each call in the body).

      — $\delta_{S_Y}^{rstr}$ is *renamed* into $\delta_h^j$ (notice that $\delta_h^1 = \delta_h^2 = \cdots \delta_h^k$ up to renaming).

      — the *initialization* operation is applied to $\delta_h^j$ : adding the information that variables of the applied clause not appearing in the head are free yields $\delta_1$.

      — applying the algorithm on the node $(\delta_1, -, B_1, -, -)$ yields $\delta_2$; repeatedly applying the algorithm finally yields $\delta_{n+1}$, the abstract substitution over the variables of the applied clause after executing the body.

      — apply the *restriction* operation on $\delta_{n+1}$ to obtain $\delta_{succ}^j$, the abstract substitution over the variables of the head after successful execution of the body.

      — apply *backward unification* on $\delta_{succ}^j$ and $\delta_h^j$, this yields $\delta_{S_Y}^j$, the abstract substitutions over the variables of the call after applying the $j^{th}$ clause.

      — apply the *upper bound* operation on $\delta_{S_Y}^1, \ldots, \delta_{S_Y}^k$, with k the number of clauses defining P, this yields $\delta_{S_Y}^{succ}$, the abstract substitution over the variables of the call after successfully completing the call.

   c. the call has an ancestor node $(\delta_{S_U}^{init}, \delta_{S_V}^{rstr}, P(V_1, \ldots, V_n), -, -)$ such that, up to renaming, $\delta_{S_Y}^{rstr} = \delta_{S_V}^{rstr}$. Take some initial value for $\delta_{S_Y}^{succ}$, this allows to continue the computation. Finally, one will obtain $\delta_{S_V}^{succ}$. In case $\delta_{S_V}^{succ}$ differs, up to renaming, from the initial value, then, the computation is repeated, now with the renaming of $\delta_{S_V}^{succ}$ as

value for $\delta_{S_Y}^{succ}$. This iteration is continued until $\delta_{S_V}^{succ}$ is, up to renaming, equal to $\delta_{S_Y}^{succ}$ (a fixpoint). (Choice of initial value and termination is the subject of the next section).

d. the call has an ancestor node $(\delta_{S_U}^{init}, \delta_{S_V}^{rstr}, P(V_1, \ldots, V_n), -, -)$ but, up to renaming, $\delta_{S_V}^{rstr}$ is different from $\delta_{S_Y}^{rstr}$.
One has the choice between d1 and d2 :

d1. expansion of the node : apply step b

d2. if $\delta_{S_Y}^{rstr} \leqslant \delta_{S_V}^{rstr}$ (after renaming) then replace $\delta_{S_Y}^{rstr}$ by the renaming of $\delta_{S_V}^{rstr}$ and apply step c.
else apply the *upper bound* operation on $\delta_{S_V}^{rstr}$ and $\delta_{S_Y}^{rstr}$ (after renaming) and repeat the computation with this upper bound starting from the ancestor node until $\delta_{S_Y}^{rstr} \leqslant \delta_{S_V}^{rstr}$ (after renaming) (a fixpoint) and one can proceed as above.

3. Apply the *extension* operation on $\delta_{S_X}^{init}$ and $\delta_{S_Y}^{succ}$ to obtain $\delta_{S_X}^{next}$, the abstract substitution over *all* variables in the callers environment.

Notes

1. Different branches of the abstract AND/OR tree may contain, up to renaming of variables, identical subtrees. Of course an implementation should avoid this and use for example an extension table [3] to look up whether the abstract substitution resulting from expanding a call is already available.

2. A distinction is made between calls with a different specification (initial abstract substitution). The final code generation can either maintain the distinction and generate separate specialized code for each call or lump all cases together and generate code which is general enough to cope with each call. The choice between option $d_1$ and $d_2$ in step 2 is in fact a choice between further specialization and generalization. A pragmatic choice should be based on the relevance of the difference for the code generator.

3. Other optimizations can be based on the AND/OR tree. Branches adorned with $\perp$ in all their abstract substitutions can never be reached, they can be removed, no code is required. In the absence of side effects one can remove an OR-branch if its success-abstraction ($\delta_{succ}^{J}$) is $\perp$ . (This has the potential of avoiding some infinite loops at run-time!)

4. It is intuitively clear that $\perp$ is a correct outcome for any operation, except upper bound, if one of the inputs is $\perp$ because $\perp$ stands for the empty set of concrete substitutions, in other words for "the control never comes here". (The formalization of correctness is in section 4). It means a call whose call-abstraction is $\perp$ need not to be expanded, its success abstraction is necessarily $\perp$. The use of $\perp$ as initial success-abstraction for a recursive call is in fact the formalization of the well known implementation approach (e.g. [3]) to delay the computation of the recursive call until a non $\perp$ approximation is obtained by processing the other branches of the AND/OR tree.

5. As already stated, the algorithm is quite independent from the chosen primitive operations. The important point is the treatment of recursion. The primitive operations are chosen in such a way that it is plausible, when studying applications, to define correct primitives yielding sufficient precision. One could have chosen other primitives, the following discussion concerning termination and correctness would be unaffected.

## 3. Termination

Termination of the algorithm requires that the AND/OR tree remains finite and that the iterations described in step 2 case c and case d2 terminate.

To be finite, the AND/OR tree must be of finite depth. Because a program contains only a finite number of predicates, an infinite branch must contain an infinite number of calls $P(Y_1, \ldots, Y_n)$ for some predicate P. This is only possible if P is (directly or indirectly) recursive and the abstract domain over $\{Y_1, \ldots, Y_n\}$ contains an infinite number of distinct abstract substitutions (otherwise case c of step 2 applies). To conclude, the AND/OR tree remains finite if the abstract domain for recursive predicates is finite. If the abstract domain is infinite for some recursive predicate, then one must limit the number of expansions for that predicate (case d1) on each branch by a constant.

The iterations in case c and d2 of step 2 are also concerned with recursion. Be $(-,\delta_{Sv}^{rstr},P(V_1, \ldots, V_n),\delta_{Sv}^{succ},-)$ the ancestor call and $(-,\delta_{Sy}^{rstr},P(Y_1, \ldots, Y_n),\delta_{Sy}^{succ},-)$ the recursive call.

Let us first consider case d2. Be $\beta_0$ the initial value for $\delta_{Sv}^{rstr}$ and $\beta_1$ the computed value for $\delta_{Sy}^{rstr}$. Another iteration is required only if $\beta_1 \not\leqslant \beta_0$, but this means that upper bound $(\beta_1,\beta_0) > \beta_0$. Thus the successive values for $\delta_{Sv}^{rstr}$ form a monotonically increasing sequence. With a finite height

poset, at some point $\beta_1 \leqslant \beta_0$ must hold (it is certainly the case with $\beta_0 = \top$ ) thus the iteration must terminate.

One should notice that this iteration computes a fixpoint. Also the iteration in case c computes a fixpoint. Be aware that precision requires the fixpoint to be as small as possible but, as far as correctness is concerned, any fixpoint will do.

Similarly, the iteration in case c must terminate if the abstract domain for the recursive predicates is a finite height poset and the successive values $\beta_1, \beta_2, \cdots$ for $\delta_{Sy}^{succ}$ form a monotone sequence. A crude way to assure the latter is by slightly modifying the algorithm : when $\beta_i$ is computed as new abstract substitution for the recursive call, one assigns to $\delta_{Sy}^{succ}$ the upper bound of $\beta_i$ and $\beta_{i-1}$. This is feasible because enlarging an abstract substitution does not affect correctness, it only reduces precision. In this way, termination is assured independent from the chosen initial value for $\delta_{Sy}^{succ}$. Of course, precision requires to start from a sufficiently small starting value (e.g. $\bot$ ).

It is more appealing to formulate a different condition for termination in case c. Monotonicity is a natural requirement for the primitive operations. Actually, in section 4 we will prove that a primitive operation which does not introduce rubbish is monotone. Thus our monotonicity condition boils down to the requirement that the operations are also monotone in the rubbish they create. Be $\beta_0$ the initial value for $\delta_{Sy}^{succ}$ and $\beta_1$ the first computed value for $\delta_{Sy}^{succ}$. $\beta_1$ is used as next value for $\delta_{Sy}^{succ}$. If $\beta_1 \geqslant (\leqslant) \beta_0$ then $\beta_2 \geqslant (\leqslant) \beta_1$ because of the monotonicity. Thus one obtains a monotonically increasing (decreasing) sequence which must terminate with some $\beta_n = \beta_{n-1}$.

Taking $\bot$ as $\beta_0$ assures $\beta_1 \geqslant \beta_0$; taking $\top$ as $\beta_0$ assures $\beta_1 \leqslant \beta_0$. The latter starting value is usually not so interesting as one can expect a rather imprecise abstract substitution (fixpoint too large). In some applications one can prove that other starting values, usually derived from $\delta_{Sy}^{rstr}$, are adequate. The advantage is that less iterations are required.

Finally, we can remark that it is often possible to work with a symbolical value say $\beta_0$, this yields an expression in $\beta_0$ say $E(\beta_0)$ for $\delta_{Sy}^{j}$. The final upper bound operation expresses that $\beta_1 \geqslant \delta_{Sy}^{j}$ for all j, one of the inequalities is $\beta_1 \geqslant E(\beta_0)$. Actually, one looks for a fixpoint for which $\beta_1 = \beta_0$. Instead of repeating the computation on the tree, one can apply techniques for computing fixpoints [10] on the inequalities. Especially in

the case of tail recursion, it is often feasible to obtain $E(\beta_0)$ without making any assumption on the value of $\beta_0$. A similar observation holds for the iteration in case d2. With $\beta_0$ the initial value for $\delta_{S_V}^{rstr}$, it is sometimes possible to obtain $\delta_{S_Y}^{rstr}$ as $E(\beta_0)$. The value one looks for is the fixpoint of $\beta_1 \geqslant \beta_0$ and $\beta_1 \geqslant E(\beta_1)$, e.g. using the method in [10].

To summarize, restrictions must be imposed on the abstract domains of recursive predicates (not on the abstract domains of nonrecursive predicates!). These domains must be finite height lattices/
posets, also there must be a limit on the number of expansions of the predicate on the same branch of the AND/OR tree. If the primitive operations are monotone, then the initial success-abstraction $\delta_{S_Y}^{succ}$ must be such that a monotone sequence is started. If the primitive operations are not monotone, then a slight modification of the algorithm can still assure termination.

The difference between the abstract domain being a poset and a lattice has to do with precision of the result. With a lattice as abstract domain and monotone primitive operations which return least abstract substitutions, one will find a least fixpoint for the success-abstraction of a recursive call.

## 4. Correctness

To be useful, the AND/OR tree adorned with abstract substitutions must reflect what can happen at run-time : For each call to a predicate P which can occur at run-time, one must be able to point out a call to P in the AND/OR tree such that, prior to the execution, the concrete substitution over the variables of P is in $\delta_{S_Y}^{rstr}$, the corresponding abstract substitution. Moreover, after execution of the call, the new substitution must be part of $\delta_{S_Y}^{succ}$ while also $\delta_{S_X}^{next}$ must contain the concrete substitution over the variables in $S_X$.

Whether this will be the case depends on the definitions of the primitive operations in a particular application. In this section, we introduce the notion of consistency of a primitive operation and prove that consistency of all primitive operations implies correctness of the abstract AND/OR tree.

The primitive abstract operations are associated with the abstract AND/OR tree. As inputs they take abstract substitutions located at specific points in the AND/OR tree and they compute an abstract substitution for a specific point in the tree. (Remind Fig. 2 and the operations introduced in section

2.2).

A primitive operation maps $\beta^1_{S_1}, \ldots, \beta^n_{S_n}$ $(n \geqslant 1)$ into $\beta_S$. Be $\beta^1_{S_1}$ the abstract substitution corresponding to the first involved program point in the usual left to right tree traversal. We can initiate a partial computation with a $\theta_1 \in \gamma(\beta^1_{S_1})$.

Consistency requires that each such partial computation which

(1)runs up to the program point corresponding to $\delta_s$ (be $\rho_1, \ldots, \rho_m$ the computed substitutions)

(2)for $1 \leqslant i \leqslant n$ : at the program point corresponding to $\beta^i_{S_i}$ (be $\rho_1, \ldots, \rho_{m_i}$ the computed substitutions) it holds that

$$\theta_i = R_{S_i}(\theta_1 \text{ na } \rho_1 \text{ na } \cdots \text{ na } \rho_{m_i}) \in \gamma(\beta^i_{S_i})$$

has the property that $R_S(\theta_1 \text{ na } \rho_1 \text{ na } \cdots \text{ na } \rho_m) \in \gamma(\delta_S)$.

The important point about this is that for non-unary operations $(n > 1)$, one need not to consider all tuples $(\theta_1, \ldots, \theta_n)$ belonging to the cross product of $\gamma(\beta^1) \times \cdots \times \gamma(\beta^n)$. One can use knowledge about the concrete computation, most notably that variables can only become further instantiated, to exclude certain tuples when defining the primitive operations (they become partial functions) and proving their consistency.

A good example is extension which maps $\delta^{init}_{S_X}$, $\delta^{succ}_{S_Y}$ into $\delta^{next}_{S_X}$. We need only consider pairs $\theta_{init}, \theta_{succ}$ for which $\exists \sigma_1 : \theta_{succ} = R_{S_Y}(\theta_{init} \text{ na } \sigma)$.

Notes

1. A consistent operation may introduce rubbish. $\gamma(\delta)$ may contain substitutions $\sigma$ for which there is no tuple $(\theta_1, \ldots, \theta_n)$ satisfying the conditions of the definition.
2. A consistent definition always exists : take $O(\beta_1, \ldots, \beta_n) = \top$. However, this lacks precision.
3. Lemma : A consistent operation adding no rubbish is monotone.

**Given**

(1) $(\beta'_1, \ldots, \beta'_n) \geqslant (\beta_1, \ldots, \beta_n)$

(2) $O(\beta_1, \ldots, \beta_n) = \beta$, $O(\beta'_1, \ldots, \beta'_n) = \beta'$

(3) O is consistent : for every partial computation yielding $\sigma$ and corresponding to O such that $\theta_i \in \gamma(\delta_i)$ $(1 \leqslant i \leqslant n)$ it holds that $\sigma \in \gamma(O(\delta_1, \ldots, \delta_n))$.

(4) O does not add rubbish : for every $\sigma \in \gamma(O(\delta_1, \ldots, \delta_n))$, there exists a partial computation yielding $\sigma$ such that $\theta_i \in \gamma(\delta_i)$ $(1 \leqslant i \leqslant n)$.

**To be proven :** O is monotone : $\beta' \geqslant \beta$.

**Proof**

(5) $\beta'_i \geqslant \beta_i$ for $1 \leqslant i \leqslant n$ (from (1) and definition of $\geqslant$)

(6) $\gamma(\beta'_i) \supseteq \gamma(\beta_i)$ for $1 \leqslant i \leqslant n$ ((5) and monotonicity of $\gamma$)

(7) $\sigma \in \gamma(\beta)$ (assumption)

(8) There exists $\theta_i$ such that $\theta_i \in \gamma(\beta_i)$ for $1 \leqslant i \leqslant n$ (from (7), (2) and (4))

(9) $\theta_i \in \gamma(\beta'_i)$ for $1 \leqslant i \leqslant n$ (from (6) and (8))

(10) $\sigma \in \gamma(\beta')$ (from (9), (2) and (3))

(11) $\sigma \in \gamma(\beta) \rightarrow \sigma \in \gamma(\beta')$ (from (7) and (10))

(12) $\gamma(\beta') \supseteq \gamma(\beta)$ (from 11 and definition of $\supseteq$)

(13) $\beta' \geqslant \beta$ (from (12) and definition of $\geqslant$).

To prove that consistency implies correctness, one could convert the AND/OR tree in a program annotated with assertions and prove that consistency implies the correctness of these assertions by proving that consistency implies the conditions of the theorem given in the Drabent-Maluszynski paper [4] on the inductive assertion method for logic programs. Such a proof involves some tedious details and familiarity with [4]. As suggested by W. Drabent, we develop an intuitively more appealing proof.

For any abstract operation mapping $\beta_1, \ldots, \beta_n$ into $\delta$, there is a corresponding concrete operation. Consistency says that if the concrete

operation is applied on $(\theta_1, \ldots, \theta_n)$ yielding $\sigma$ and $\theta_i \in \gamma(\beta_i)$ for $1 \leqslant i \leqslant n$ then $\sigma \in \gamma(\delta)$. It means one can follow the execution of a concrete query on the abstract AND/OR tree. Look again at Fig. 1 and Fig. 2. Assume $\theta_{init} \in \gamma(\delta_{S_X}^{init})$, i.e. the substitution which holds prior to executing P is in the abstraction. In the absence of recursion, one can follow without problems the execution of the concrete query on the abstract tree (the $j^{th}$ OR-branch is taken when the $j^{th}$ clause is used to solve P). Because of consistency, $\theta \in \gamma(\delta)$ for each pair $\theta - \delta$ of corresponding substitutions.

Recursion poses a problem, one can arrive in an unexpanded leaf of the abstract AND/OR tree. $\theta_{rstr} \in \gamma(\delta_{S_Y}^{rstr})$ but how to establish that, after returning from the call, $\theta_{succ} \in \gamma(\delta_{S_Y}^{succ})$? The unexpanded leaf node must have an ancestor node with $\delta_{S_Y}^{rstr} = \delta_{S_Y}^{rstr}$ due to the way the abstract AND/OR tree is constructed. The subtree rooted at this ancestor node can be used to expand the leaf (alternatively, we can position ourselves at the ancestor node, on exit, we have to switch back to the leaf) and we can continue, using the consistency argument to show that $\theta \in \gamma(\delta)$ for each pair of corresponding substitutions. This shows that the AND/OR tree is correct : it correctly predicts all substitutions which can possibly appear during the execution of any query Q(...) satisfying the initial abstract substitution of the root node of the AND/OR tree. It is interesting to note the role of the fixpoint in the leaf node. It assures that $\theta_{succ} \in \gamma(\delta_{S_Y}^{succ})$ independent of the number of expansions which are needed to mimic the concrete query.

This correctness implies that an optimizing compiler can use the abstract AND/OR tree and can exploit the properties of the abstract substitutions to generate code which is less general and more efficient than the code from an ordinary Prolog compiler. For example, an abstract substitution can express that a certain argument of a procedure call is a free variable, a ground term, a term with principal functor f, ... . This allows to avoid a number of tests, also the creation of backtrack points can sometimes be avoided.

## 5. Primitive operations revisited

Having discussed the importance of monotonicity and consistency for respectively termination and correctness, we can return to the primitive operations : we can point out some useful properties independent from any application. First let us recall the notion of abstract substitution as

described in section 2.1. We mentioned the distinction between generators and n-ary constraints ($n > 1$). Also, that one sometimes prefers to represent the complement of a constraint.

Another important notion is precision. Operations should preferably return an abstract substitution which is consistent but also as small as possible. A desirable property is that the returned value is *least* : that all other consistent abstract substitutions are definitely larger than the obtained one. A weaker property is *minimality* : that all other consistent abstract substitutions are not smaller than the obtained one.

Finally, it is useful to remind that it is consistent for all operations, except upper bound, to yield $\bot$ if one of the operands is $\bot$ .

## 5.1 Restriction

Restriction maps $\delta_{S_X}$ into $\delta_{S_Y}$ with $S_Y \subseteq S_X$. Restriction retains the generators of variables in $S_Y$. Constraints require more care, one certainly retains constraints mentioning only variables of $S_Y$. The treatment of other constraints is application dependent.

## 5.2 Renaming

The concrete operation corresponding to renaming is the unification between $P(X_1,\ldots,X_n)$ and $P(Y_1,\ldots,Y_n)$ where a substitution $\theta = \{X_i \leftarrow t_i\}$ holds over the $X_i$ and the empty substitution $\epsilon$ over the $Y_i$. The substitution of interest is $\sigma = \{Y_i \leftarrow t_i\}$ which holds over the $Y_i$ after the unification. The abstract operation maps $\delta_{S_X}$ into $\delta_{S_Y}$. Replacing all occurrences of $X_i$ in $\delta_{S_X}$ by the corresponding $Y_i$ gives a $\delta_{S_Y}$ which is consistent and does not create any rubbish, in other words, it is least and monotone. Indeed $\theta = \{X_i \leftarrow t_i\} \in \gamma(\delta_{S_X})$ iff $\sigma = \{Y_i \leftarrow t_i\} \in \gamma(\delta_{S_Y})$.

## 5.3 Initialization

Initialization extends $\delta_{S_X}$ into $\delta_{S_Y}$ with $S_Y$ a superset of $S_X$. In the concrete operation, the variables of $S_Y \backslash S_X$ are unbound. Often, abstract substitutions will add some rubbish e.g. allowing that variables of $S_Y \backslash S_X$ are bound to an unbound variable outside the domain $S_Y$. Any operation extending $\delta_{S_X}$ with generators for $S_Y \backslash S_X$ allowing the variable to remain unbound is consistent.

## 5.4 Backward unification

The concrete operation unifies $P(X_1,\ldots,X_n)\theta_1$ with $P(Y_1,\ldots,Y_n)\theta_2$, yielding a substitution for the $X_i$. However, $\theta_1$ and $\theta_2$ are not independent.

With $\theta_1 = \{X_i \leftarrow t_i\}$, there exists a $\sigma$ such that $\theta_2 = \{Y_i \leftarrow t_i\ \sigma\}$. It means that the resulting substitution for the $X_i$ is $\theta = \{X_i \leftarrow t_i\ \sigma\}$. The abstract operation maps $\delta_1$ and $\delta_2$ into $\delta$. From the above, it follows that the replacement of all occurrences of $Y_i$ in $\delta_2$ by corresponding $X_i$ yields a consistent $\delta$. Because $\delta_2$ can contain rubbish, the thus obtained $\delta$ will also contain rubbish. Some of the rubbish can be eliminated by observing that each element of $\gamma(\delta)$ must be an instance of some element of $\gamma(\delta_1)$. In some applications, this can allow to reduce $\delta$. Note that renaming is a monotone operation. Whether the removal of rubbish is monotone is application dependent.

### 5.5 Upper bound

This is an operation on the poset/lattice of the abstract domain. A least upper bound exists and the corresponding operation is monotone when the abstract domain is a lattice. It is necessarily consistent.

### 5.6 Extension

Extension maps $\beta_{S_X}$ and $\beta_{S_Y}$ into $\delta_{S_X}$ with $S_Y$ a subset of $S_X$. Also here one can exploit knowledge from the concrete operation : $\theta_2 \in \gamma(\beta_{S_Y})$ is such that there exists $\theta_1 \in \gamma(\beta_{S_X})$ and a $\sigma$ such that $\theta_2 = R_{S_Y}(R_{S_Y}(\theta_1)\ \mathrm{na}\ \sigma)$. Extension is usually the most difficult operation. It must adapt the generators of $S_X \backslash S_Y$ such that if $X \leftarrow t$ was in $\beta_{S_X}$, $X \leftarrow t\ \sigma$ is in $\delta_{S_X}$. Also constraints must be adapted. Removal is an easy but imprecise solution. Moreover, it is not consistent when the absence of constraints is represented.

### 5.7 Abstract interpretation of built-ins

This operation mapping $\beta$ to $\delta$ has to cope with $X = Y$ and $X = f(Y_1, \ldots, Y_n)$. The unification modifies the sets of terms which are possible instantiations of X and Y (the $Y_i$). To be consistent, the abstract interpretation must reflect this modification. $\delta = \beta$ is consistent in case $\beta$ only allows ground terms; in general, this is not the case.

## 6. An application : mode inferencing

In this section, we apply the general framework on the problem of mode inferencing. We show an abstraction which is simple while giving nontrivial results similar to those of Debray in [3].

Abstract substitutions different from $\perp$ are represented as sets. Such a set contains a generator for each variable in the domain of the abstract

substitution and possibly some constraints. A generator represents the *mode* of a variable. For a variable X, we distinguish between *free*, *ground* and *any*. The first mode, denoted M(X,f) forces X to be either unbound or bound to a variable; the second, denoted M(X,g) allows only ground terms while the last, denoted M(X,a) does not impose any restriction on concrete substitutions (remind, concrete substitutions are idempotent). Notice that the values a, f and g form a *poset* with $f < a$, $g < a$.

We write $PSHR(\{Y_1, \ldots, Y_n\})$ to express that the variables $Y_1, \ldots, Y_n$ are allowed to share free variables. An abstract substitution over a set of variables S partitions the nonground variables into sets whose elements are allowed to share free variables with each other. Thus, with elements $PSHR(S_1), \ldots, PSHR(S_n)$ we have that (1) $S_i \cap S_j = \phi$ for $i \neq j$ and (2) $\bigcup_{1 \leqslant i \leqslant n} S_i = S \backslash S_g$. (Throughout this section, we will use $S_g$ to denote the subset with the ground variables).

For example, with $\beta = \{M(X,a), M(Y,a), PSHR(\{X\}), PSHR(\{Y\})\}$, we have that

$$\gamma(\beta) = \{\epsilon, \{X \leftarrow t\}, \{Y \leftarrow t\}\} \cup$$
$$\{\{X \leftarrow t_1, Y \leftarrow t_2\} : var(t_1) \cap var(t_2) = \phi\}.$$

This representation of constraints is inspired by S. Debray.

With $\beta$ and $\delta$ two abstract substitutions over the same domain S, we have $\beta \leqslant \delta$ iff

(1) $\forall X$ ($X \in S \rightarrow$ the mode of X in $\beta \leqslant$ the mode of X in $\delta$).

(2) $\forall S_1$ ($PSHR(S_1) \in B \rightarrow \exists S_2$ ($S_1 \subseteq S_2$ and $PSHR(S_2) \in \delta$)).

In words, the domains of the generators of $\delta$ are at least as large and $\delta$ allows at least as much sharing.

The abstract domain is clearly finite, there is a minimal element $\perp$ and every two elements have a least upper bound. Thus the abstract domain is a lattice.

The lub of two abstract substitutions $\beta_1$ and $\beta_2$ (different from $\perp$) is obtained as follows :

— For the mode of a variable, one takes the lub in the poset formed by a, f and g.

— For the constraints, one has to merge sets of $\beta_1$ with sets of $\beta_2$ when they overlap. This is achieved by the following algorithm.

Initialization :

$S_i \in S$ if $PSHR(S_i) \in \beta_1$ or $PSHR(S_i) \in \beta_2$.

Repeat

if $S_i \cap S_j \neq \phi$ then $S = S \backslash \{S_i, S_j\} \cup \{S_i \cup S_j\}$

until all pairs $S_i, S_j$ are disjoint

$PSHR(S_i) \in \delta$ if $S_i \in S$.

The lattice for two variables is shown in Fig. 3.

Now, we can define the primitive operations. To facilitate reading, we use a Horn clause like format (omitting quantors were possibly), we also omit domain subscripts if no confusion can arise.

— Restriction maps $\beta_{S_X}$ into $\delta_{S_Y}$ with $S_Y \subseteq S_X$.
The variables of $S_Y$ have the same mode in $\delta$ as in $\beta$ :
$M(Y,M_y) \in \delta \leftarrow Y \in S_Y$[2], $M(Y,M_y) \in \beta$.

In $\delta$ we only retain sharing between variables of $S_Y$ :
$PSHR(S) \in \delta \leftarrow PSHR(T) \in \beta, T \cap S_Y = S, S \neq \phi$.

It is evident that this operation is monotone and consistent.

— Renaming maps $\beta_{S_X}$ into $\delta_{S_Y}$ with $S_X = \{X_1, \ldots, X_n\}$, $S_Y = \{Y_1, \ldots, Y_n\}$. As explained in section 5, $\delta$ is obtained from $\beta$ by replacing the $X_i$ by the corresponding $Y_i$. The operation is consistent and monotone. Moreover, it does not create rubbish.

— Initialization maps $\beta_{S_X}$ into $\delta_{S_Y}$ with $S_X \subseteq S_Y$.
Variables of $S_X$ keep their mode :
$M(X,M_x) \in \delta \leftarrow M(X,M_x) \in \beta$.

2. Here $S_Y$ does not stand for a quantified variable but for the domain of $\delta_{S_Y}$. We hope the meaning is clear from the context and apologize for the abuse of notation.

The new variables take the mode free :
$M(Y,f) \in \delta \leftarrow Y \in S_Y \backslash S_X$.

Sharing between variables of $S_X$ is unaffected :
$PSHR(S) \in \delta \leftarrow PSHR(S) \in \beta$.

The new variables do not share :
$PSHR(\{Y\}) \in \delta \leftarrow Y \in S_Y \backslash S_X$.

It is clear that the operation is monotone and consistent. It creates some rubbish, indeed the concretization allows $Y_i \leftarrow U$ for $Y_i \in S_Y \backslash S_X$ and $U \notin S_Y$.

— Backward unification maps $\beta_{S_X}$ (and $\beta_{S_Y}$) into $\delta_{S_Y}$ with $S_X = \{X_1, \ldots, X_n\}$, $S_Y = \{Y_1, \ldots, Y_n\}$.
As argued in section 5, renaming of $\beta_{S_X}$ is a consistent and monotone realization of this operation. With the available information, it is the best we can do.

— Upper bound : the least upper bound in the lattice of the abstract substitutions is of course a consistent and monotone realization of this operation.

— Extension maps $\beta_{S_X}$, $\beta_{S_Y}$ into $\delta_{S_X}$ with $S_Y \subseteq S_X$.
The modes in $\delta$ of the variables in $S_Y$ are those of $\beta_{S_Y}$ :
$M(Y,M_Y) \in \delta \leftarrow M(Y,M_y) \in \beta_{S_Y}$.

For the variables in $S_X \backslash S_Y$, the situation is more complex. Variables can only become further instantiated. Thus for modes a and g, it is consistent to retain the same mode as in $\beta_{S_X}$ :
$M(X,M_x) \in \delta \leftarrow X \in S_X \backslash S_Y$, $M(X,M_x) \in \beta_{S_X}$, $M_x \neq f$.

A free variable can be instantiated due to the instantiation of variables in $S_Y$. This can only have happened if that variable possibly shares with a variable in $S_Y$ which is not free in $\beta_{S_Y}$ :
$M(X,a) \in \delta \leftarrow X \in S_X \backslash S_Y$, $M(X,f) \in \beta_{S_X}$, $X \in S$, $PSHR(S) \in \beta_{S_X}$, $Y \in S$, $M(Y,f) \notin \beta_{S_Y}$.

Free variables remain free otherwise :
$M(X,f) \in \delta \leftarrow X \in S_X \backslash S_Y$, $M(X,f) \in \beta_{S_X}$, not $(\exists Y,S : X \in S, PSHR(S) \in \beta_{S_X}, Y \in S, M(Y,f) \notin \beta_{S_Y})$.

For what concerns sharing, variables which have become ground in $\beta_{S_Y}$ cannot share anymore, they can be removed from the sets in $\beta_{S_X}$. Also sets in $\beta_{S_X}$ must be merged if they become linked due to the new sharing in $\beta_{S_Y}$. The following algorithm achieves this :

Initialization :

$S_i \in S \leftarrow PSHR(S_i) \in \beta_{S_Y}$

$S_i \in S \leftarrow PSHR(S_i') \in \beta_{S_X}$, $S_i = S_i' \backslash S_g$ ($S_g$ the set of ground variables in $\beta_{S_Y}$ or $\delta$).

Repeat

if $S_i \cap S_j \neq \phi$ then $S = S \backslash \{S_i, S_j\} \cup \{S_i \cup S_j\}$

until all pairs $S_i, S_j$ are disjoint

$PSHR(T) \leftarrow T \in S$

Monotonicity is easily checked. We hope the argument for consistency has been convincing although it was informal.

— Abstract interpretation of $X = Y$ mapping $\beta$ to $\delta$.
Be $S_X = \{X\}$, $S_Y = \{Y\}$.
Both variables become ground if one of them was ground :
$M(U,g) \in \delta \leftarrow U \in S_X \cup S_Y, \exists .V\ (M(V,g) \in \beta)$

Both variables remain free if both were free, moreover, now they become sharing :
$M(U,f) \in \delta \leftarrow U \in S_X \cup S_Y, X \in S_X, M(X,f) \in \beta, Y \in S_Y, M(Y,f) \in \beta$
$PSHR(\{X,Y\}) \in \delta \leftarrow X \in S_X, M(X,f) \in \beta, Y \in S_Y, M(Y,f) \in \beta.$

Otherwise, we obtain the top element :
$M(U,a) \in \delta \leftarrow U \in S_X \cup S_Y$
$PSHR(\{X,Y\}) \in \delta \leftarrow X \in S_X, Y \in S_Y.$

Monotonicity and consistency should be evident.

— Abstract interpretation of $X = f(Y_1, \ldots, Y_n)$ mapping $\beta$ to $\delta$.
Be $S_Y = \{Y_1, \ldots, Y_n\}$, $S_X = \{X\}$.
All variables take mode ground if X initially has mode ground.
$M(U,g) \in \delta \leftarrow X \in S_X, M(X,g) \in \beta, U \in S_X \cup S_Y.$

This is also the case if all $Y_i$ are initially ground :
$M(U,g) \in \delta \leftarrow \text{not}(\exists Y : Y \in S_Y, M(Y,g) \notin \beta), U \in S_X \cup S_Y$.

Otherwise, if X was free, then it takes mode any while nothing happens to the modes of the $Y_i$. The sharing sets must be extended with X. (we assume programs do not create circular data structures).
$M(X,a) \in \delta \leftarrow M(X,f) \in \beta$
$M(Y,M_Y) \in \delta \leftarrow M(X,f) \in \beta, M(Y,M_y) \in \beta$
$PSHR(S) \in \delta \leftarrow M(X,f) \in \beta, PSHR(S') \in \beta, S = S' \cup \{X\}$.

Otherwise all nonground modes become any while sharing is possible between all nonground variables.
$M(U,a) \in \delta \leftarrow U \in S_X \cup S_Y, M(U,g) \notin \beta$
$M(U,g) \in \delta \leftarrow M(U,g) \in \beta$
$PSHR(S) \in \delta \leftarrow S = S_X \cup S_Y \backslash S_g$ ($S_g$ the ground variables).

Also here, monotonicity and consistency are evident.

**Example**

P(X,Y) ← Q(X,Y),R(X),S(Y).
Q(X,Y) ← X = Y.
R(X) ← X = a.
S(X) ←
← P(X,Y).

Suppose the call abstraction of P is {M(X,f),M(Y,f)}. The call abstraction of Q(X,Y) is the same, executing X = Y adds PSHR({X,Y}) yielding {M(X,f),M(Y,f),PSHR({X,Y})} in the program point between Q and R. The call abstraction of R is {M(X,f)}, its success abstraction is {M(X,g)}. Extension now changes M(Y,f) to M(Y,a) and removes PSHR({X,Y}), this yields {M(X,g),M(Y,a)}. The call abstraction of S is {M(Y,a)} which is correct although imprecise. More precision can be obtained with the repeat previous call strategy : again interpreting Q(X,Y) after finishing R(X) or by refining the abstract domain, e.g. adding constraints SVAL(X,Y) with the meaning "X and Y have the same value".

Note:

Clauses whose unnormalized head does not unify with an unnormalized call should not be used in expanding that call. The failure is not recognized and adds imprecision because of the normalization. A more fundamental approach towards more precision consists of extending the abstract domain, e.g. by including principal functors.

## 7. Concluding remarks

The use of abstract interpretation in computer science can be traced back [12] to Peter Naur [9] who used it for type checking in Algol compilers. Worth mentioning is also the work of Sintzoff [11]. The seminal paper of the Cousot's [2] provides a (complex) theoretical foundation for the abstract interpretation of programs in imperative languages.

Global analysis of programs also attracted the interest of researchers in the area of logic programming. One of the first and best known examples is the work of Mellish on automatic generation of modes. It is reported in [7]. The early attempts were very pragmatic. Later on, inspired by [2], one looked for a more formal basis. A not so satisfying attempt by Mellish is described in [8]. For his Ph.D. thesis Debray also studied the work of the Cousot's very carefully and developed an abstract interpreter for mode analysis [3]. The "safety criterion" he needed is not very elegant. Jones and Sondergaard developed a rigorous theory based on denotational semantics. The foundation in denotational semantics is not very appealing to the logic programming community and the development of applications is far from trivial.

We have attempted to liberate the theory of abstract interpretation from the heritage of imperative languages and the work of the Cousot's. We have developed a framework which is, we hope, appealing to logic programmers. We claim that our approach is simple and elegant. We have reduced the problem of developing a correct application to the problem of selecting an appropriate abstraction, of selecting suitable primitive operations (we have proposed a set) and of proving that these operations have some crucial properties (consistency - monotonicity). To support this claim, we have developed the case of mode analysis where a well chosen abstraction allowed to define a simple extension operation, avoiding Debrays unelegant safety criterion [3]. For other applications, most noteworthy compile-time garbage collection, we refer to [1]. What remains to be done is the development of a more formal and systematic approach

towards proving consistency of primitive operations.

**Acknowledgement**

This work has grown out of an effort to provide a theoretical basis for the engineering effort towards higher performance of Prolog-compilers. The lack of intuitive appeal of existing work was the incentive. We are indebted to many people for their helpful discussions, patience, support and insistence for more rigor. Most notably, W. Drabent, H. Sondergaard, F. Kluzniak, K. Musumbu, G. Janssens, B. Demoen, A. Callebaut, B. Martens, D. De Schreye and P. Weemeeuw.

**References**

[1] Bruynooghe, M., Janssens, G., Callebaut, A. and Demoen, B., Abstract interpretation : towards the global optimization of PROLOG programs, Proc. 4th Symposium on Logic Programming, San Francisco (September 1987).

[2] Cousot, P. and Cousot, R., Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints, Proc. 4th ACM POPL Symposium, Los Angeles (June 1977), 238-252.

[3] Debray, S.K. and Warren, D.S., Automatic mode inferencing for Prolog Programs, Proc. 3th Symposium on Logic Programming, Salt Lake City (September 1986), IEEE Society Press, 78-88.

[4] Drabent, W. and Maluszynski, J., Inductive assertion method for logic programs, Proc. TAPSOFT 87, Pisa (March 1987), Springer Verlag, 167-181.

[5] Jones, N.D. and Sondergaard, H., A semantics based framework for the abstract interpretation of Prolog, in Abstract Interpretation of Declarative Languages, eds. S. Abramsky and C. Hankin, Ellis Horwood, in print.

[6] Lassez, J.-L., Maher, M.J. and Marriott, K., Unification revisited, Draft report, IBM Thomas J. Watson Research Center, York Town Heights, New York, 1986.

[7] Mellish, C.S., Some global optimizations for a Prolog compiler, Journal of Logic Programming 2, 1 (April 1985), 43-66.

[8] Mellish, C.S., Abstract interpretation of Prolog programs, Proc. 3th Int. Conf. on Logic Programming, London, (July 1986), Springer-Verlag, 463-474.

[9] Naur, P., Checking of operand types in Algol compilers, BIT 5 (1965), 151-163.

[10] O'Keefe, R.A., Finite fixed-point problems, Proc. 4th Int. Conf. on Logic Programming, Melbourne (May 1987), MIT Press, 729-743.

[11] Sintzoff, M., Calculating properties of programs by valuation on specific models, Proc. ACM Conf. on Proving Assertions about Programs, New Mexico (Sigplan Notices 7, 1), (January 1972), 203-207.

[12] Sondergaard, H., Abstract interpretation of logic programs, Draft, DIKU, Univ. of Copenhagen, January 1987.

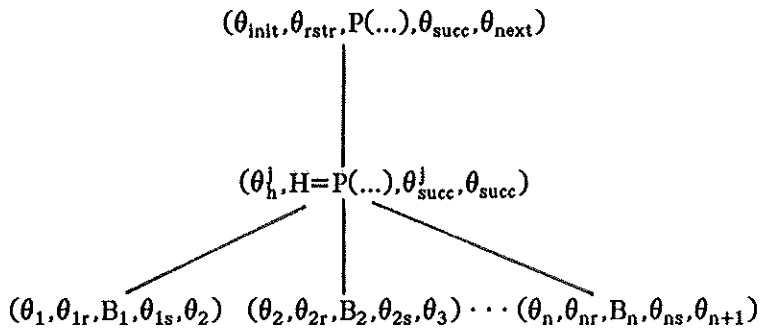


Fig. 1

Fragment of the concrete interpretation of all call P using the $j^{th}$ clause $H \leftarrow B_1,\ldots,B_n$ : AND-tree adorned with snapshots of substitutions. The substitutions $\theta_{1s}$, $\theta_{i+1}$ , . . . . , $\theta_{n+1}$, $\theta_{succ}^j$, $\theta_{succ}$ and $\theta_{next}$ and the subtrees $B_{i+1},\ldots,B_n$ are absent in case the execution of subgoal $B_i$ is unfinished.

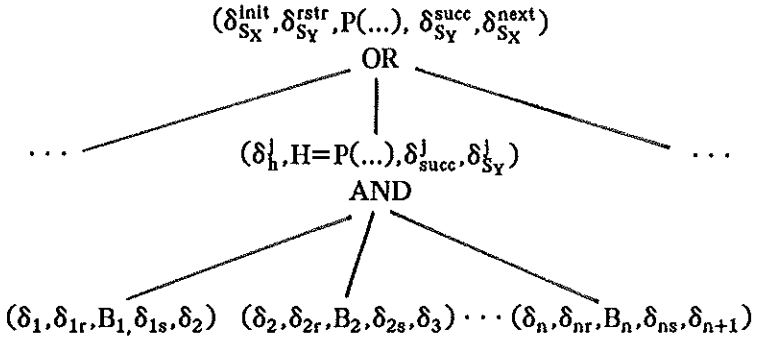


Fig. 2

Fragment of the abstract interpretation of a call P : AND/OR-tree showing the OR-branch of the $j^{th}$ clause. The tree is adorned with abstract substitutions.

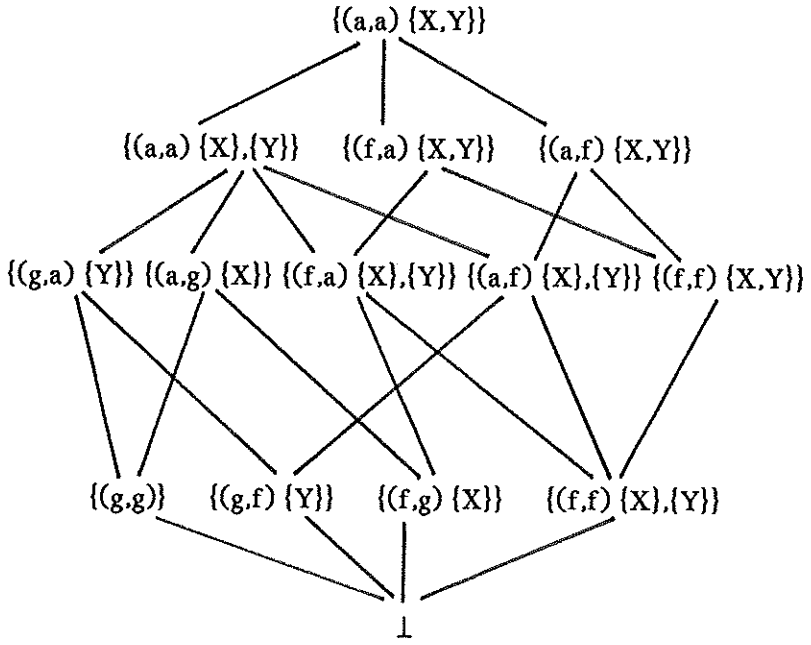


Fig. 3 : abstract domain over 2 variables X and Y.
(p,q) stands for M(X,p), M(Y,q), {S} stands for PSHR({S}).