(Journal of Logic Programming, Vol. 13, N. 2–3)

# Compile-Time Derivation of Variable Dependency Using Abstract Interpretation

**K. Muthukumar**
MCC and Department of Computer Science
The University of Texas at Austin
Austin, TX 78712 - USA
muthu@cs.utexas.edu

**M. Hermenegildo**[1]
Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain
herme@cs.utexas.edu *or* herme@fi.upm.es

**Abstract**

Traditional schemes for abstract interpretation-based global analysis of logic programs generally focus on obtaining procedure argument mode and type information. Variable sharing information is often given only the attention needed to preserve the correctness of the analysis. However, such sharing information can be very useful. In particular, it can be used for predicting run-time goal independence, which can eliminate costly run-time checks in and-parallel execution. In this paper, a new algorithm for doing abstract interpretation in logic programs is described which concentrates on inferring the dependencies of the terms bound to program variables with increased precision and at all points in the execution of the program, rather than just at a procedure level. Algorithms are presented for computing abstract entry and success substitutions which extensively keep track of variable aliasing and term dependence information. In addition, a new, abstract domain independent fixpoint algorithm is presented and described in detail. The algorithms are illustrated with examples. Finally, results from an implementation of the abstract interpreter are presented.

## 1 Introduction

The technique of abstract interpretation for flow analysis of programs in imperative languages was first presented in a sound mathematical setting by Cousot and Cousot [5] in their landmark paper. Later, it was shown by Bruynooghe [1], Jones and Sondergaard [17], and Mellish [21] that this technique can be extended to flow analysis of programs in logic programming languages. Specific algorithms for such global analysis in logic programs have been given by a number of researchers ([8], [18], [23], [26], [27], [2], [24], [20], [19], [11], [4] ...). These schemes, mostly geared towards optimizing the *sequential* execution of logic programs, generally focus on computing information about the arguments of *predicates* used in

[1]The research reported in this paper was performed at MCC, U. of Texas at Austin, and U. of Madrid (UPM). This work was funded in part by MCC and also in part by ESPRIT project 2471 "PEPMA."

the program, such as (1) the *mode* of an argument, i.e., whether a particular argument of a predicate is instantiated on input or on output or both and (2) the *type* of an argument, i.e., set of terms that an argument is bound to when the predicate is called or when it succeeds. Variable sharing (or "aliasing"), i.e., the fact that unification can bind variables to other variables or to terms which in turn share variables, is "dealt with" in these methods in order to preserve the correctness of the approach, but it is not generally considered as an output of the analysis and often computed in a very conservative way [6].

However, the variable sharing information itself can often be of the utmost importance for a compiler. For example, such information can be used for compile-time optimization of backtracking [3]. Knowledge of variable sharing information also makes it possible to predict run-time goal independence, which is particularly relevant for a compiler which targets execution on a system which supports Independent And-Parallelism (IAP) (see, for example, [13, 15, 9] and their references for more details on this type of parallelism): in IAP subgoals in the body of a clause are executed in parallel provided they are *independent*, i.e., their run-time instantiations do not share any variables. As shown in [15, 13], this condition can be ensured by run-time checks on the *groundness* and *independence* of certain program variables.[2] However, these checks can be expensive, increasing overhead and reducing the amount of speed-up achievable through parallelism. Thus, it is of great advantage to eliminate as many checks as possible by gathering highly accurate information at compile-time regarding the groundness and independence of the terms to which programs variables will be bound at run-time. Furthermore, it is useful to have this information for all points in the program, rather than globally for each procedure. The inference of such information is the main subject of this paper.[3] Our main contributions are as follows:

- Starting with an approach for representing abstract substitutions (in the form of sharing information) suggested to us by Jacobs and Langen[4] [16] we present new abstract unification algorithms which compute abstract *entry* substitutions and abstract *success* substitutions while extensively keeping track of variable aliasing and term dependence information. These algorithms can be used in isolation (if only variable sharing information is to be the output of the analysis) or in combination with conventional abstract domains as a method for accurately keeping track of

[2] Program variables are variables that are in the text of the given program.

[3] Due to the similarities between the search tree explored by a program executed in IAP and that of sequential execution [13], conventional abstract interpretation techniques can be applied (with only minor modifications) to programs which are to be evaluated in IAP (Debray presents in [7] an analysis framework for other types of parallelism where the properties of IAP regarding the similarity with sequential execution don't hold). In [27] we reported some results obtained from an abstract interpreter for IAP constructed more or less along the lines of conventional systems, except for the techniques used to improve its efficiency. This interpreter is most apt at generating groundness information and it was shown in [27] to be reasonably effective at reducing run-time checks. The approach presented in this paper is targeted at improving those results through better tracking of terms which are independent but not ground.

[4] Even though the representation that we use for abstract substitutions is essentially the same as in Jacobs and Langen [16], there are fundamental differences between our approach and theirs. Most importantly, our algorithm for abstract interpretation uses a *top-down directed bottom-up approach* while theirs uses a *pure bottom-up approach* ([8], [20], [19]). Consequently, we use a novel fixpoint computation algorithm which takes care of additional complexities brought about by the top-down directed approach, as opposed to the conventional bottom-up fixpoint computation.

variable aliasing.

- We present and give a complete description of a new algorithm for performing top-down driven, bottom up fixpoint computation which avoids recalculation (by performing fixpoint computation over subsets of the program, rather than reanalyzing the whole program at each step) and uses approximations as seeds for convergence improvement. Its output includes abstract substitution information for all points in the program. While the essential ideas behind computation of fixpoints in the context of logic programs are understood, the formulation presented herein takes care of practical efficiency and correctness issues and many details which, to our knowledge, and particularly in the case of a top-down driven algorithm, have not been described elsewhere.

The algorithms are illustrated with examples. We assume that the reader is familiar with logic programming (and Prolog to some extent) and the basic concepts of abstract interpretation of logic programs. However, the following section provides a brief overview of the process in order to introduce the notation and place in context the algorithms to be presented later. The rest of the paper is organized as follows: section 3 introduces the concept of abstract substitution used throughout the paper. Sections 4 and 5 deal with abstract unification respectively explaining how the abstract entry substitution for a clause and the abstract success substitution for a subgoal are computed. Section 7 presents the fixpoint algorithm. Section 8 illustrates the complete abstract interpretation algorithm through examples and presents results obtained from an implementation of our algorithm aimed at the detection of groundness and independence. Finally, section 9 summarizes our conclusions and discusses suggestions for future work.

## 2 Abstract Interpretation of Logic Programs

As mentioned previously, abstract interpretation is a useful technique for performing a global analysis of a program in order to compute, at compile-time, characteristics of the terms to which the variables in that program will be bound at run-time for a given class of queries. In principle, such an analysis could be done by an interpretation of the program which computed the *set of all possible substitutions* (collecting semantics) at each step. However, these sets of substitutions can in general be infinite and thus such an approach can lead to non-terminating computations. Abstract interpretation offers an alternative in which the program is interpreted using *abstract substitutions* instead of actual substitutions. An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain. The set of all possible terms that a variable can be bound to in abstract substitutions represents an "abstract domain" which is usually a complete lattice or cpo of finite height (such finiteness required, in principle, for termination of fixpoint computation), whose ordering relation is herein represented by "$\sqsubseteq$." Abstract substitutions and sets of concrete substitutions are related via a pair of functions referred to as the *abstraction* ($\alpha$) and *concretization* ($\gamma$) functions. In addition, each primitive operation $u$ of the language (unification being a notable example) is abstracted to an operation $u'$ over the abstract domain. Soundness of the analysis requires that each concrete operation $u$ be related to its corresponding abstract operation $u'$ as follows: for

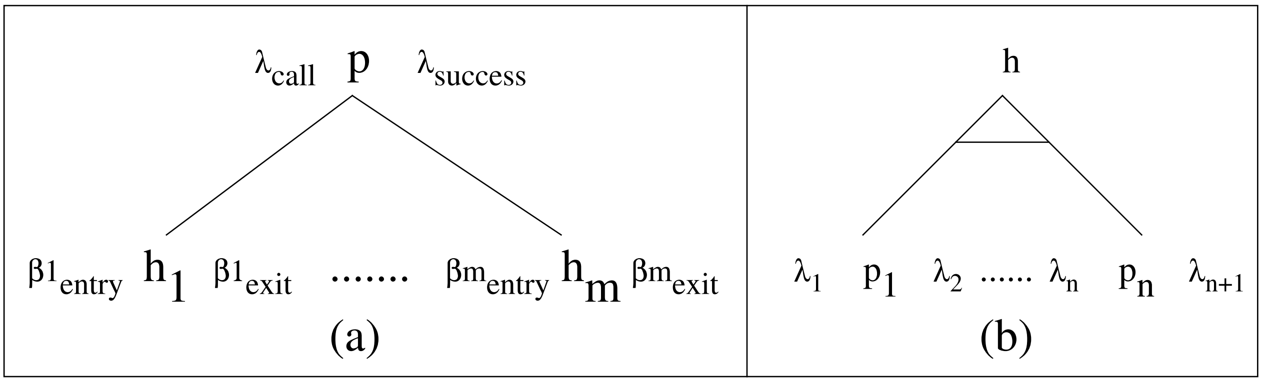


Figure 1: Illustration of the abstract interpretation process

every $x$ in the concrete computational domain, $u(x) \sqsubseteq \gamma(u'(\alpha(x)))$.

The input to the abstract interpreter is a set of clauses (the program) and set of "query forms." In its minimal form (least burden on the programmer) such query forms can be simply the names of the predicates which can appear in user queries (i.e., the program's "entry points"). In order to increase the precision of the analysis, query forms can also include a description of the set of abstract (or concrete) substitutions allowable for each entry point. The goal of the abstract interpreter is then to compute in abstract form the set of substitutions which can occur at all points of all the clauses that would be used while answering all possible queries which are concretizations of the given query forms. It is convenient to give different names to abstract substitutions depending on the point in a clause to which they correspond. Consider, for example, the clause $h$ :- $p_1, \ldots, p_n$. Let $\lambda_i$ and $\lambda_{i+1}$ be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. See figure 1(b).

**Definition 1** *$\lambda_i$ and $\lambda_{i+1}$ are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal $p_i$. For this same clause, $\lambda_1$ is the* abstract entry substitution *(also represented as $\beta_{entry}$) and $\lambda_{n+1}$ is the* abstract exit substitution *(also represented as $\beta_{exit}$).* □

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one being to essentially follow a top-down strategy starting from the query forms.[5] Several frameworks for doing abstract interpretation in logic programs follow along these lines. One such framework is described in detail for example in [1]. In a similar way to the concrete top-down execution, the abstract interpretation process can then be represented as an abstract AND-OR tree, in which AND-nodes and OR-nodes alternate. A clause head $h$ is an AND-node whose children are the literals

[5]More precisely, this strategy can be seen as a *top-down driven* bottom up computation. As will be shown later, some degree of fixpoint, bottom up computation is required for correctness in the presence of recursive predicates. A purely bottom-up analysis scheme is also possible ([8], [20], [19]). The advantage of the top-down driven strategy is that it restricts the abstract computation to that required for the query forms given rather than that for all possible query forms. Note that query forms are routinely present in actual programs in the form of module entry point declarations, so no extra burden need be placed on the user. Additional information from the user can, of course, focus the abstract computation even further and increase its precision.

in its body $p_1, \ldots, p_n$ (figure 1(b)). Similarly, if one of these literals $p$ can be unified with clauses whose heads are $h_1, \ldots, h_m$, $p$ is an OR-node whose children are the AND-nodes $h_1, \ldots, h_m$ (figure 1(a)). During construction of the tree, computation of the abstract substitutions at each point is done as follows:

- *Computing success substitution from call substitution*: Given a call substitution $\lambda_{call}$ for a subgoal $p$, let $h_1, \ldots, h_m$ be the heads of clauses which unify with $p$ (see figure 1(a)). Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ corresponding to these clauses. The success substitution $\lambda_{success}$ is then the *least upper bound* (LUB) of $\lambda 1_{success}, \ldots, \lambda m_{success}$. Of course the LUB computation is dependent on the abstract domain and the definition of the $\sqsubseteq$ relation.

- *Computing exit substitution from entry substitution*: Given a clause $p_0$ :- $p_1, \ldots, p_n$ whose body is non-empty and an entry substitution $\lambda_1$, $\lambda_1$ is the call substitution for $p_1$. Its success substitution $\lambda_2$ is computed as above. Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. Finally, $\lambda_{n+1}$ is obtained, which is the exit substitution for this clause. See figure 1(b). For a unit clause (i.e. whose body is empty), its exit substitution is the same as its entry substitution.

Given this basic framework, it is clear that a particular analysis strategy needs to:

- Define an abstract domain and substitution framework, and the $\sqsubseteq$ relation,

- Describe how to compute the *entry substitution* for a clause C given a subgoal $p$ (which unifies with the head of C) and its call substitution,

- Describe how to compute the *success substitution* for a subgoal $p$ given its call substitution and the exit substitution for a clause C whose head unifies with $p$.

Such information represents the "core" of a particular analysis strategy. Sections 3, 4 and 5 respectively address the corresponding definitions and algorithms for the approach presented in this paper.

In addition to the three points above, there is, however, one more issue that needs to be addressed. The overall abstract interpretation scheme described works in a relatively straightforward way if the program has no recursion. Consider, on the other hand, a recursive predicate $p$. If there are two OR-nodes for $p$ in the abstract AND-OR tree such that

- they are identical (i.e., they have the same atoms),

- one is an ancestor of the other, and

- the call substitutions are the same for both,

then the abstract AND-OR tree is *infinite* and an abstract interpreter using the simple control strategy described above will not terminate. In order to ensure termination, some sort of *fixpoint computation* is required. In order to support such fixpoint computation, *memo tables* [10] are used, for example, in [8] and stream predicates are used in [26]. In this paper we propose a novel scheme for fixpoint computation within the context of abstract interpretation. This is described in section 7.

# 3 Abstraction Framework

In this section, we describe the representation of abstract substitutions used in our abstract interpreter. As mentioned before, in the concrete interpretation the collecting semantics for a top down execution of logic programs is usually given in terms of the sets of substitutions associated with each program point [1]. The traditional approach ([1],[8],[18]) to abstracting such sets of substitutions is to define an *abstract domain* and then to describe a method for constructing an *abstract substitution* corresponding to a set of substitutions.

For example, the abstract domain used in [1] consists of three elements `ground`, `free` and `any`. These elements respectively correspond to the set of all ground terms, the set of all unbound (free) variables, and the set of all terms. An abstract substitution is then defined as a mapping from program variables (of a clause) to elements of the abstract domain. For example, if $X$ and $Y$ are the program variables in a clause, then an abstract substitution at a point in that clause could be $\{X/ground, Y/free\}$. This abstract substitution actually represents the set of all substitutions in which $X$ is bound to a ground term and $Y$ is bound to a free variable.

The approach used for defining abstract substitutions in this paper is entirely different. We are not *per se* interested in the set of terms that a program variable is bound to at a point in a clause. Rather, we are interested in the *sharing* of variables among the sets of terms that program variables are bound to.[6] For example, let $X$ and $Y$ be the program variables in a clause. The abstract substitution in our abstract interpreter should tell us whether the sets of terms that $X$ and $Y$ are bound to, share any variables or not.

We define the abstract substitution for a clause to be *a set of sets of program variables* in that clause following an approach initially suggested in [16]. Informally, a set of program variables appears in the abstract substitution if the terms to which these variables are bound share a variable. For the example clause of the previous paragraph, the value of an abstract substitution may be $\{\{X\},\{X,Y\}\}$. This abstract substitution corresponds to a set of substitutions in which $X$ and $Y$ are bound to terms $t_X$ and $t_Y$ such that (1) at least one variable occurs in both $t_X$ and $t_Y$ (this corresponds to the element $\{X,Y\}$) and (2) at least one variable occurs only in $t_X$ (this corresponds to the element $\{X\}$).

In a sense, the term *abstract substitution* may be a misnomer for such a data structure. The reason for such an objection would be that this data structure only *abstracts* a set of substitutions but it does not (explicitly) tell us about the set of terms a program variable is bound to in a set of substitutions (which the conventional abstract substitutions do, as discussed above). Nevertheless, we use the term *abstract substitution* for the data structure introduced above, since it does abstract the information contained in a set of substitutions.

Before formally describing the representation for abstract substitutions, we review some basic definitions about substitutions. A substitution for the variables of a clause is a mapping from the set of program variables in that clause ($Pvar$) to terms that can be formed from the universe of all

[6] Note that this approach to abstracting substitutions is *complementary* to the traditional approach, i.e., it is possible to combine the two approaches and use abstract substitutions which provide information about both sharing between program variables and the terms that they are bound to.

variables ($Uvar$), and the constants and functors in the given program and query. The domain of a substitution $\theta$ is written as $dom(\theta)$. We consider only idempotent substitutions. The instantiation of a term $t$ under a substitution $\theta$ is denoted as $t\theta$ and $var(t\theta)$ denotes the set of variables in $t\theta$.

Let $\theta$ be a given substitution for a clause C. A program variable X, which is in C, is *ground* under this substitution if $var(X\theta) = \emptyset$. Program variables X and Y, which are in C, are *independent* if $var(X\theta) \cap var(Y\theta) = \emptyset$ [13]. We say that variable V occurs in program variable X under the substitution $\theta$ if $V \in var(X\theta)$. Clearly, a program variable X is ground under a substitution $\theta$ if there is no variable V which occurs in X under $\theta$ and program variables X and Y are independent if there is no variable V which occurs in both of them under $\theta$.

Below, we formally define the abstract substitution $\mathcal{A}(\theta)$ which corresponds to a concrete substitution $\theta$ and later we extend it to sets of substitutions. The basic idea behind this definition is as follows: a set S of program variables appears in $\mathcal{A}(\theta)$ iff there is a variable V which occurs in each member of S under $\theta$. Thus, a program variable is ground iff it does not appear in any set $\mathcal{A}(\theta)$, and two program variables are independent iff they do not appear together in any set in $\mathcal{A}(\theta)$.

**Definition 2** Subst *is the set of all substitutions which map variables in* Pvar *to terms constructed from variables in* Uvar *and constants and functors in the given program and query.* □

**Definition 3** Asubst *is the set of all abstract substitutions for a clause, i.e.,* $Asubst = \wp(\wp(Pvar))$ *where* $\wp(S)$ *denotes the powerset of* $S$. □

**Definition 4** *The function* Occ *takes two arguments,* $\theta$ *(a substitution) and* $U$ *(a variable in* Uvar*) and produces the set of all program variables* $X \in Pvar$ *such that* $U$ *occurs in* $var(X\theta)$ *i.e.*

$$Occ(\theta, U) = \{X | X \in dom(\theta) \wedge U \in var(X\theta)\}$$

□

**Definition 5 (Abstraction of a substitution)**

$$\mathcal{A} : Subst \rightarrow Asubst$$

$$\mathcal{A}(\theta) = \{Occ(\theta, U) | U \in Uvar\}$$

□

**Example**: Let $\theta = \{W/a, X/f(A_1, A_2), Y/g(A_2), Z/A_3\}$. $Occ(\theta, A_1) = \{X\}$, $Occ(\theta, A_2) = \{X, Y\}$, $Occ(\theta,\ A_3) = \{Z\}$ and $Occ(\theta, U) = \emptyset$ for all other $U \in Uvar$. hence, $\mathcal{A}(\theta) = \{\emptyset, \{X\}, \{X, Y\}, \{Z\}\}$.□

The abstraction function $\mathcal{A}$ is extended to sets of substitutions as follows:

**Definition 6 (Abstraction of a set of substitutions)**

$$\alpha : \wp(Subst) \rightarrow Asubst$$

$$\alpha(\Theta) = \bigcup_{\theta \in \Theta} \mathcal{A}(\theta)$$

□

Essentially, $\alpha$ constructs the union of the sharing information found in all substitutions in $\Theta$. The corresponding concretization function is:

**Definition 7 (Concretization)**

$$\gamma : Asubst \rightarrow \wp(Subst)$$

$$\gamma(SS) = \{\theta | \theta \in Subst \wedge \mathcal{A}(\theta) \subseteq SS\}$$

□

If a clause has $N$ program variables, there can be at most $2^{2^N}$ different abstract substitutions for it. A *partial order* can be defined on these abstract substitutions. $\lambda_1 \sqsubseteq \lambda_2$ iff $\gamma(\lambda_1) \subseteq \gamma(\lambda_2)$. It can be easily shown that $\lambda_1 \sqsubseteq \lambda_2$ iff $\lambda_1 \subseteq \lambda_2$. Since the set of all abstract substitutions for a clause is finite and is *closed* under union, it follows that the *least upper bound* of two abstract substitutions is equal to their *union* and the *greatest lower bound* is equal to their *intersection*.

We can make the following observations from the above definitions:

- Since the lattice of abstract substitutions for a clause is finite and hence has a finite depth, we are assured that fixpoint computation (discussed in section 7) terminates [1].
- For a given clause, the *top* element in the lattice is the powerset of all the program variables in that clause.
- The *bottom* element in the lattice for all clauses is $\emptyset$. The meaning of this abstract substitution can be explained as follows: suppose a clause has a subgoal *sg* which cannot be satisfied under its abstract call substitution $\lambda$ i.e., *sg* fails. The abstract success substitution for *sg* would then be $\emptyset$.
- The abstract substitution which makes all program variables in a clause ground is $\{\emptyset\}$.
- $\emptyset$ is an element of every non-empty abstract substitution $\lambda$. This is a consequence of the fact that every concrete substitution $\theta$ has a finite *range*.

Since the abstract interpreter manipulates only abstract substitutions and since these abstract substitutions do not have complete information about the term each program variable is bound to, approximations are introduced in our computations of abstract substitutions. We require that these be *safe* approximations.

**Definition 8 (safe approximation)** *Suppose the concrete set of substitutions that occurs at a point in a clause is $\Theta$ and the abstract interpreter computes the abstract substitution at this point as $\lambda$. $\lambda$ is a safe approximation to the actual abstract substitution at this point if, whenever variables $X$ and $Y$ are* dependent *according to at least one substitution in $\Theta$, there is a set $S \in \lambda$ such that $X \in S$ and $Y \in S$ i.e., the abstract substitution should capture all the sharing information. Similarly, if a variable $X$ is ground according to $\lambda$, it should be ground according to all substitutions in $\Theta$.* □

Thus a computed abstract substitution which is a safe approximation to the actual one is allowed to be conservatively imprecise: it can indicate that two variables are dependent when actually they are independent according to the concrete set of substitutions. Similarly, a variable can be nonground according to such an abstract substitution even if it is ground according to the concrete set of substitutions. Therefore, the sharing information in such an abstract substitution is characterized as *potential* sharing. All the abstract substitutions that are mentioned in subsequent sections of this paper are *conservative* abstract substitutions i.e., they are safe approximations to the actual abstract substitutions.

### 3.1 Other definitions

In this section, we present some definitions and results that are used in sections 4 and 5.

Given a set of program variables $S$ and a subgoal $pred(u_1, \ldots, u_n)$, $pos(pred(u_1, \ldots, u_n), S)$ gives the set of all argument positions of this subgoal in which at least one element of $S$ occurs.

**Definition 9**

$$pos(pred(u_1, \ldots, u_n), S) = \{i | S \cap var(u_i) \neq \emptyset\}$$

□

Given a subgoal $pred(u_1, \ldots, u_n)$ and an abstract substitution $\lambda$, the function $\mathcal{P}(pred(u_1, \ldots, u_n), \lambda)$ computes the dependencies among the argument positions of this subgoal due to $\lambda$. This is expressed as a subset of the powerset of $\{1, \ldots, n\}$ (similar to representing an abstract substitution as a set of sets of program variables).

**Definition 10**

$$\mathcal{P}(pred(u_1, \ldots, u_n), \lambda) = \{pos(pred(u_1, \ldots, u_n), S) | S \in \lambda\}$$

□

**Example**: Let $n = 2$, $u_1 = f(X, Y)$, $u_2 = g(Y, Z)$, and $\lambda = \{\emptyset, \{X\}, \{Y\}, \{X, Z\}\}$.

$$pos(pred(f(X, Y), g(Y, Z)), \emptyset) = \emptyset$$

$$pos(pred(f(X, Y), g(Y, Z)), \{X\}) = \{1\}$$

$$pos(pred(f(X, Y), g(Y, Z)), \{Y\}) = \{1, 2\}$$

$$pos(pred(f(X, Y), g(Y, Z)), \{X, Z\}) = \{1, 2\}$$

Therefore, $\mathcal{P}(pred(f(X, Y), g(Y, Z)), \lambda) = \{\emptyset, \{1\}, \{1, 2\}\}$.□

**Definition 11 (Closure under union)** *For a set of sets $SS$, the closure $SS^*$ of $SS$ is the smallest superset of $SS$ that satisfies: $S_1 \in SS^* \wedge S_2 \in SS^* \Rightarrow S_1 \cup S_2 \in SS^*$.* □

**Proposition 1** *Let $\sigma$ and $\mu$ be two concrete substitutions, whose domains are* Pvar *and* Uvar *respectively. Let $\lambda$ be an abstract substitution such that $\mathcal{A}(\sigma) \subseteq \lambda$. Then $\mathcal{A}(\mid \sigma \circ \mu \mid_{dom(\sigma)}) \subseteq \lambda^*$, where $\mid \sigma \circ \mu \mid_{dom(\sigma)}$ indicates the restriction of $\sigma \circ \mu$ to the domain of $\sigma$.* $\square$

**Proof:** We note that

$$Occ(\mid \sigma \circ \mu \mid_{dom(\sigma)}, X) \quad = \bigcup_{X \in var(Y\mu)} Occ(\sigma, Y)$$

Since $\mathcal{A}(\sigma) = \{Occ(\sigma, U) \mid U \in Uvar\}$, we have $\mathcal{A}(\mid \sigma \circ \mu \mid_{dom(\sigma)}) \subseteq (\mathcal{A}(\sigma))^* \subseteq \lambda^*$.$\square$

**Corollary 1** *Let $\lambda_{call}$ and $\lambda_{success}$ be the abstract call and success substitutions for a subgoal* sg, *respectively corresponding to $\Theta_{call}$ (the set of all its call substitutions) and $\Theta_{success}$ (the set of all its success substitutions). Then $\lambda_{success} \subseteq \lambda^*_{call}$.* $\square$

**Proof:** For each $\theta_{call} \in \Theta_{call}$, there exists a $\theta_{success} \in \Theta_{success}$ and a substitution $\mu$ (this is the substitution obtained by "solving" the subgoal *sg*) such that $\theta_{success} = \mid \theta_{call} \circ \mu \mid_{dom(\theta_{call})}$. Therefore,

$$\begin{aligned}
\lambda_{success} &= \alpha(\Theta_{success}) \\
&= \bigcup_{\theta_{success} \in \Theta_{success}} \mathcal{A}(\theta_{success}) \\
&= \bigcup_{\theta_{success} \in \Theta_{success}, \theta_{call} \in \Theta_{call}, \theta_{success} = \mid \theta_{call} \circ \mu \mid_{dom(\theta_{call})}} \mathcal{A}(\mid \theta_{call} \circ \mu \mid_{dom(\theta_{call})}) \\
&\subseteq \bigcup_{\theta_{call} \in \Theta_{call}} (\mathcal{A}(\theta_{call}))^* \\
&\subseteq (\bigcup_{\theta_{call} \in \Theta_{call}} \mathcal{A}(\theta_{call}))^* \\
&= (\alpha(\Theta_{success}))^* \\
&= (\lambda_{call})^*
\end{aligned}$$

$\square$

**Corollary 2** *Let $\lambda_{call}$ and $\lambda_{success}$ be as in corollary 1. Then $\mathcal{P}(sg, \lambda_{success}) \subseteq (\mathcal{P}(sg, \lambda_{call}))^*$.* $\square$

**Proof:** From corollary 1 we get $\lambda_{success} \subseteq \{S \mid \exists S_i \in \lambda_{call}(S = \bigcup_i S_i)\}$. We observe that $pos(sg, \bigcup_i S_i) = \bigcup_i pos(sg, S_i)$. Therefore,

$$\begin{aligned}
\mathcal{P}(sg, \lambda_{success}) &= \{pos(sg, S) \mid S \in \lambda_{success}\} \\
&\subseteq \{pos(sg, \bigcup_i S_i) \mid \exists S_i \in \lambda_{call}\} \\
&= \{\bigcup_i pos(sg, S_i) \mid \exists S_i \in \lambda_{call}\} \\
&= \{pos(sg, S) \mid S \in \lambda_{call}\}^* \\
&= (\mathcal{P}(sg, \lambda_{call}))^*
\end{aligned}$$

$\square$

**Definition 12 (Projection)** *Let $\mu$ be an abstract substitution for a subgoal* sg *and $S_{sg}$ be the set of program variables in this subgoal. The projection of $\mu$ on* sg *is defined as the abstract substitution $\{S \mid S = S' \cap S_{sg}, S' \in \mu\}$.* □

**Corollary 3** *Let the subgoal* sg *(with a projected abstract call substitution $\lambda$) be unified with the head* hd *of a clause C. The abstract entry substitution for C, $\beta_{entry}$ satisfies the condition $\mathcal{P}(hd, \beta_{entry}) \subseteq (\mathcal{P}(sg, \lambda))^*$.* □

**Proof:** Let $\lambda_{unify(sg,hd)}$ be the abstract substitution for *sg* after its unification with *hd*. After unification, the dependencies among the argument positions are the same for both *sg* and *hd*, since they have been instantiated to the same term. Therefore, $\mathcal{P}(hd, \beta_{entry}) = \mathcal{P}(sg, \lambda_{unify(sg,hd)})$. By arguments similar to the proofs of corollaries 1 and 2, it can be shown that $\mathcal{P}(sg, \lambda_{unify(sg,hd)}) \subseteq (\mathcal{P}(sg, \lambda))^*$. □

Unless otherwise noted, all substitutions referred to in the rest of this paper are *abstract* substitutions.

# 4 Computing the Abstract Entry Substitution

In this section, we describe an algorithm to compute the (abstract) entry substitution for a clause C given a subgoal *sg* (which unifies with the head *hd* of this clause) and *sg*'s (abstract) call substitution.

If the program variables in *hd* belong to a set $S_{hd}$, then a *conservative* entry substitution for this clause would be $\wp(S_{hd})$. But this is too *pessimistic* an estimate, since it says that every program variable in *hd* is *potentially* dependent on every other program variable. To get a more accurate estimate, we determine which program variables in $S_{hd}$ are ground and try to reduce the sharing information in the entry substitution. An algorithm for performing this task is given in section 4.1. Section 4.2 illustrates this algorithm with an example. This algorithm can be summarized as follows:

- `Perform abstract unification`: Do a term by term unification for *sg* and *hd* and determine the potential sharing information between the program variables in *sg* and *hd*. This is done in steps 1 through 3.

- `Propagate groundness information`: A program variable in $S_{hd}$ is ground if it is unified with a ground term in *sg*. This term could be ground either because the program variables in it are ground in *sg*'s call substitution, because it does not contain any program variables, because some of its program variables are ground due to unification with terms in *hd*, or because of a combination of the above. This is done in steps 4 through 6.

- `Apply independence information in` *sg*`'s call substitution`: Take the remaining program variables (which are *potentially* nonground) in $S_{hd}$. Form dependencies among them based on the results of abstract unification and groundness analysis. Eliminate some of these dependencies based on the information in *sg*'s call substitution. This is done in steps 7 through 10.

## 4.1 Algorithm

Let the set of program variables which occur in $sg$ be $S_{sg} = \{X_1, X_2, \ldots, X_m\}$. Let $sg = pred(s_1, s_2, \ldots, s_n)$ and the head $hd$ (which is unifiable with $sg$) $= pred(t_1, t_2, \ldots, t_n)$. Let the set of the program variables in $hd$ be $S_{hd} = \{Y_1, Y_2, \ldots, Y_p\}$ and the set of program variables which do not occur in $hd$ but occur in the body of the clause of $hd$ be $\{Y_{p+1}, \ldots, Y_q\}$. We assume[7] that $S_{sg} \cap \{Y_1, \ldots, Y_q\} = \emptyset$. Let $\lambda_{call}$ be the call substitution of the subgoal $sg$. Below we describe the algorithm for computing the entry substitution $\beta_{entry}$ for the clause C = *hd* :- *body*.

1. `Projection`: Compute $\lambda$ by projecting $\lambda_{call}$ on to the set $S_{sg}$, i.e.,

   $$\lambda \leftarrow \{S \mid S = (S' \cap S_{sg}), S' \in \lambda_{call}\}$$

   $\lambda$ contains all the *potential* sharing information among program variables in $sg$.

2. `Normalize unification equations`: i.e., for each pair of terms $s_i, t_i, 1 \leq i \leq n$, normalize the equation $s_i = t_i$ so that it is replaced by a set of equations $Z = Term_Z, Z \in S_{sg} \cup S_{hd}$. Form the set $\mathcal{U}$ as follows:

   $$\mathcal{U} \leftarrow \{(Z, Set_Z) \mid Set_Z = var(Term_Z), Z = Term_Z \text{ } is \text{ } a \text{ } normalized \text{ } equation\}$$

3. `Grouping`: For each $Z$ such that $(Z, Set1_Z), \ldots (Z, Setk_Z)$ are elements of $\mathcal{U}$, replace these elements with $(Z, \{Set1_Z, \ldots Setk_Z\})$. The presence of this element in $\mathcal{U}$ means that, due to the unification of $sg$ and $hd$, the program variable $Z$ is bound to $k$ different terms, respectively containing the sets of program variables $Set1_Z, \ldots, Setk_Z$.

4. `Initialize the set of ground program variables`: Let $G$ denote the set of program variables in $sg$ and $hd$ that are ground. Initialize $G$ as follows: for all $(Z, SS_Z) \in \mathcal{U}$ such that

   - $\emptyset \in SS_Z$ (i.e., $Z$ is bound to a ground term due to the current unification), or
   - $Z$ belongs to the set $S_{sg}$ and is ground according to $\lambda$,

   add $Z$ to $G$. We also maintain a *queue* $L$ of ground program variables, whose groundness has not been propagated to other program variables. Initially $L$ contains the same elements as $G$ in some order.

5. `Groundness propagation`:
   Repeat

   (a) Dequeue $Z$ from $L$;

   (b) Let $G1 \leftarrow \{W \mid W \notin G, (Z, SS) \in \mathcal{U}, S \in SS, W \in S\}$. Update $G \leftarrow G \cup G1$. Also, enqueue the elements in $G1$ to the queue $L$ and remove $(Z, SS)$ from $\mathcal{U}$ (this step ensures that the "groundness" of $Z$ is transmitted to all the program variables that occur in the terms that $Z$ is bound to);

[7] This assumption is valid due to renaming of variables in clauses.

(c) For all $W, S, SS$ such that $(W, SS) \in \mathcal{U}$, $S \in SS$ and $Z \in S$, remove $Z$ from $S$. If $S$ becomes an empty set and if $W$ is not in the set $G$, enqueue $W$ in the queue $L$ and add it to the set $G$ (this step ensures that occurrences of $Z$ are removed from the RHS of the unification equations);

Until the queue $L$ is empty.

6. `Update` $\lambda$: $\lambda \leftarrow \{S \mid S \in \lambda, S \cap G = \emptyset\}$. This is an update of the call substitution $\lambda$ to reflect the fact that some variables in $S_{sg}$ have become ground due to unification of *sg* with *hd*.

7. `Potential dependency graph formation`: Build an undirected graph $G_{ST}$ which will reflect potential sharing between instantiations of program variables. Let $G_{ST} = (V, E)$, where $V = (S_{sg} \cup S_{hd}) - G$ and an edge between two vertices indicates a potential sharing between program variables represented by the two vertices. $E = E1 \cup E2$ where $E1$ and $E2$ are computed as follows:
   - $E1 \leftarrow \{(X_i, X_j) \mid X_i \in S, X_j \in S, S \in \lambda, i \neq j\}$ (In this step, we carry over the sharing information between program variables in $\lambda$ to the graph $G_{ST}$).
   - $E2 \leftarrow \{(W, Z) \mid (W, SS) \in \mathcal{U}, Z \in S, S \in SS\}$ (In this step, we carry over the sharing information due to unification to the graph $G_{ST}$).

8. `Graph partitioning`: Let $S_{hd} - G$ be partitioned into mutually disjoint sets $HP_1, \ldots, HP_r$ such that $Y_i$ and $Y_j$ belong to the same partition if and only if there is a path between them in the graph $G_{ST}$.

9. `Form a first approximation to` $\beta_{entry}$:
$$\beta \leftarrow \bigcup_{i=1}^{r} \wp(HP_i)$$
   It is clear that the entry substitution $\beta_{entry}$ for the clause C is a subset of $\beta$.

10. `Prune` $\beta$ `down to form` $\beta_{entry}$: $\beta$ may contain some sharing information among the arguments of the subgoal predicate that is not compatible with $\lambda$. In this step, we remove such "spurious" sharing information from $\beta$. Consider $\mathcal{P}(sg, \lambda)$. This gives the sharing information among the arguments of *sg* due to the abstract substitution $\lambda$. By unifying *sg* with the head *hd* of the clause $C$, the new sharing among the arguments of this subgoal can only be a subset of $(\mathcal{P}(sg, \lambda))^*$. This is proved in Corollary 3 (section 3). We take advantage of this fact in "pruning" down $\beta$. $\beta_{hd} \leftarrow \{S \mid S \in \beta, pos(hd, S) \in (\mathcal{P}(sg, \lambda))^*\}$. The entry substitution for the clause $C$ is $\beta_{entry} = (\beta_{hd}) \cup \{\{Y_{p+1}\}, \ldots, \{Y_q\}\}$.

**Proposition 2** *Given a subgoal* sg *whose abstract call substitution is* $\lambda_{call}$ *and a clause* $C$ *whose head* hd *unifies with* sg*, let* $\beta_{entry}$ *be the abstract entry substitution for* $C$ *as computed by the above algorithm. Then,* $\beta_{entry}$ *is a safe approximation in the following sense: In the concrete interpretation, let* $\Omega_{entry}$ *be the set of entry substitutions for clause* $C$ *computed from* sg*'s set of call substitutions* $\gamma(\lambda_{call})$*. Then,* $\Omega_{entry} \subseteq \gamma(\beta_{entry})$*.* $\square$

**Proof (Outline):** The main proof burden is to show that the dependencies among the program variables in *hd* induced by the dependencies in $\lambda_{call}$ and by the unification of *sg* with *hd* are *safely* computed. This is precisely done when the *potential dependency graph* is formed. Firstly, the dependencies due to unification are computed in steps 2 and 3. Secondly, the program variables that are bound to ground terms due to unification and $\lambda_{call}$ are identified in a straightforward manner in steps 4, 5 and 6. Now the potential dependency graph, which shows potential dependencies among its possibly nonground variables, is formed. Two variables are potentially dependent iff there is a path between them i.e. they are dependent according to $\lambda_{call}$ or they are dependent due to unification or both. Consider a partition P in this graph. The powerset of P describes all possible dependencies among the variables of P. Therefore, in step 9, we form a first approximation to $\beta_{entry}$ by taking the union of the powersets of all partitions (restricted to variables in $S_{hd}$) of the potential dependency graph. However, we can refine this value of $\beta_{entry}$ further by removing some spurious dependencies in it by using corollary 1 of proposition 1. This is done in step 10. The final value of $\beta_{entry}$ as computed by this algorithm is thus a *safe* approximation. $\square$

## 4.2 An Example

We illustrate the above algorithm with the aid of an example.

| | |
|---|---|
| The subgoal *sg* | $pred(X_1, f(X_2, X_4), X_3, g(X_3), f(X_4, h(X_4)), X_5)$ |
| The head *hd* (of clause $C$) | $pred(p(Y_1), Y_2, q(Y_3, Y_6), Y_4, f(r(Y_5), Y_6), Y_6)$ |
| The calling substitution $\lambda_{call}$ | $\{\emptyset, \{X_1\}, \{X_3\}, \{X_6\}, \{X_1, X_2, X_7\}, \{X_3, X_4\}\}$ |

Here $S_{sg} = \{X_1, X_2, X_3, X_4, X_5\}$ and $S_{hd} = \{Y_1, Y_2, Y_3, Y_4, Y_5, Y_6\}$. Let $\{Y_7, Y_8\}$ be the set of variables in the body of the clause $C$ that do not occur in its head *hd*. In the following, we illustrate how $\beta_{entry}$, the entry substitution for the clause $C$, is computed given the above information:

1. `Projection`: $\lambda = \{\emptyset, \{X_1\}, \{X_3\}, \{X_1, X_2\}, \{X_3, X_4\}\}$

2. `Normalize unification equations`:

$$\mathcal{U} = \{(X_1, \{Y_1\}), (Y_2, \{X_2, X_4\}), (X_3, \{Y_3, Y_6\}), (Y_4, \{X_3\}), (X_4, \{Y_5\}), (Y_6, \{X_4\}), (Y_6, \{X_5\})\}$$

3. `Grouping`: In this step we simplify $\mathcal{U}$ by collecting together tuples which have the same LHS.

$$\begin{aligned}\mathcal{U} \;=\; & \{(X_1, \{\{Y_1\}\}), (Y_2, \{\{X_2, X_4\}\}), (X_3, \{\{Y_3, Y_6\}\}), \\ & (Y_4, \{\{X_3\}\}), (X_4, \{\{Y_5\}\}), (Y_6, \{\{X_4\}, \{X_5\}\})\}\end{aligned}$$

4. Initially, $G = \{X_5\}$ and the queue $L$ contains only one element, $X_5$.

5. `Groundness propagation`: The queue $L$ contains $X_4, Y_6, Y_5$ at various points during this step. After this step, $G = \{X_4, X_5, Y_5, Y_6\}$ and

$$\mathcal{U} = \{(X_1, \{\{Y_1\}\}), (Y_2, \{\{X_2\}\}), (X_3, \{\{Y_3\}\}), (Y_4, \{\{X_3\}\})\}$$

6. `Update` $\lambda$: $\lambda = \{\emptyset, \{X_1\}, \{X_3\}, \{X_1, X_2\}\}$

7. `potential dependency graph formation`: The graph $G_{ST} = (V, E)$ where, $V = \{X_1, X_2, X_3, Y_1, Y_2, Y_3, Y_4\}$ and $E = \{(X_1, X_2), (X_1, Y_1), (X_2, Y_2), (X_3, Y_3), (X_3, Y_4)\}$.

8. `Graph partitioning`: The set $S_{hd} - G$ is partitioned into two sets, $\{Y_1, Y_2\}$ and $\{Y_3, Y_4\}$.

9. Taking the union of the powersets of the above partitions, we get

$$\beta = \{\emptyset, \{Y_1\}, \{Y_2\}, \{Y_1, Y_2\}, \{Y_3\}, \{Y_4\}, \{Y_3, Y_4\}\}$$

10. `Prune` $\beta$ `down to form` $\beta_{entry}$: $\mathcal{P}(sg, \lambda) = \{\emptyset, \{1\}, \{1,2\}, \{3,4\}\}$ and $pos(hd, \{Y_1\}) = \{1\}$, $pos(hd, \{Y_2\}) = \{2\}$, $pos(hd, \{Y_1, Y_2\}) = \{1,2\}$, $pos(hd, \{Y_3\}) = \{3\}$, $pos(hd, \{Y_4\}) = \{4\}$ and $pos(hd, \{Y_3, Y_4\}) = \{3,4\}$. It is clear that $\{Y_2\}, \{Y_3\}, \{Y_4\}$ can be removed from $\beta$. To this pruned down $\beta$ we add $\{Y_7\}$ and $\{Y_8\}$ to get $\beta_{entry} = \{\emptyset, \{Y_1\}, \{Y_1, Y_2\}, \{Y_3, Y_4\}, \{Y_7\}, \{Y_8\}\}$.

# 5 Computing the Abstract Success Substitution

In the previous section, we described an algorithm for computing the entry substitution $\beta_{entry}$ for a clause C = *hd* :- *body*, given a subgoal *sg* (which is unifiable with *hd*) and *sg*'s call substitution $\lambda_{call}$. In this section we describe an algorithm to compute the success substitution $\lambda_{success}$ for *sg*, given the exit substitution $\beta_{exit}$ for the clause C, i.e., the substitution at the "rightmost" point of the clause C. This algorithm makes use of the abstract unification information computed in the previous algorithm. Also, the sets of variables $S_{sg}$ and $S_{hd}$ that are used here will be the same as in section 4.1.

If $\beta_{exit} = \emptyset$ i.e., the exit substitution is $\perp$ indicating that clause C didn't succeed, then obviously $\lambda_{success} = \emptyset$. Else, we execute the algorithm in the following section. Broadly, the various steps in this algorithm can be explained as follows:

- First we project the exit substitution on to the set of program variables in *hd* (step 1). We then check if any of these program variables is ground according to the exit substitution but was not ground according to the entry substitution. These variables became ground during the execution of the body of clause C. We propagate the groundness of these variables to the appropriate variables in *sg* (steps 2 and 3).

- We then compute the *potential* dependencies among the program variables in *sg* by forming a dependency graph as before and taking the union of the appropriate powersets of program variables in *sg* (steps 4 through 6).

- Some of these dependencies may be spurious, i.e (1) they may not agree with the call substitution of *sg* or (2) they may not agree with the dependencies among the arguments of *sg* induced by the exit substitution of the clause C. These spurious dependencies are removed (step 7).

- What we have now is the projection of the success substitution of *sg* on its program variables. This is extended to all the program variables in the clause of *sg* (step 8).

### 5.1 Algorithm

1. `Projection`: Compute $\beta'$ by projecting $\beta_{exit}$ on to the set $S_{hd}$ (the set of variables in the head $hd$), i.e.,

$$\beta' \leftarrow \{S \mid S = (S' \cap S_{hd}), S' \in \beta_{exit}\}$$

   $\beta'$ is effectively all the information from $\beta_{exit}$ that is used in this algorithm.

2. `Groundness propagation`: Start with the values of $G$,$\mathcal{U}$ and $\lambda$ at the end of step 6 of the previous algorithm. Let $G2 \leftarrow \{Z \mid Z \in S_{hd}, Z \notin G, \forall S(S \in \beta' \Rightarrow Z \notin S)\}$ i.e., $G2$ contains *new* ground program variables in $hd$ that were not ground according to $\beta$. Update $G \leftarrow G \cup G2$. Also, enqueue the elements of $G2$ to the queue $L$. This queue is used in the same manner as in the algorithm in section 4.

   If $L$ is empty, then go to step 4. Else, execute the `groundness propagation` step (step 5) of the previous algorithm.

3. `Update` $\lambda$: Execute step 6 of the previous algorithm.

4. `Potential dependency graph formation`: Execute step 7 of the previous algorithm. Let $E3 \leftarrow \{(Y_i, Y_j) \mid Y_i \in S, Y_j \in S, S \in \beta'\}$. $E3$ contains the *new* sharing information obtained from $\beta'$. Update $E \leftarrow E \cup E3$.

5. `Graph partitioning`: Let $S_{sg} - G$ be partitioned into mutually disjoint sets $SP_1, \ldots, SP_s$ such that $X_i$ and $X_j$ belong to the same partition if and only if there is a path between them in the graph $G_{ST}$.

6. `Form a first approximation to the projection of` $\lambda_{success}$ `on` $sg$:

$$\lambda' \leftarrow \bigcup_{i=1}^{s} \wp(SP_i)$$

   It is clear that $(\lambda_{success} \cap S_{sg})$ is a subset of $\lambda'$.

7. `Prune` $\lambda'$ `down to get the projection of` $\lambda_{success}$ `on` $sg$: $\lambda'$ may contain some sharing information among the arguments of the subgoal predicate that is not compatible with $\lambda$ and with $\beta'$. In this step, we remove such "spurious" sharing information from $\lambda'$.

   - Consider $\mathcal{P}(hd, \beta_{exit})$. This gives the sharing information among the arguments of $hd$ (and hence of $sg$) due to the abstract exit substitution $\beta_{exit}$ for the clause $C$. It is clear that the sharing information among the arguments of $sg$ induced by $\lambda_{success} \cap S_{sg}$ (and hence $\lambda_{success}$) has to be the same as well. Therefore, any element in $\lambda'$ that leads to an argument sharing that is not in $\mathcal{P}(hd, \beta_{exit})$ must be removed.
   - Also, as discussed in section 3 (corollaries 1 and 2), the successful execution of the subgoal $sg$ can only produce a success substitution which is a subset of $\lambda^*$. Therefore, any element of $\lambda'$ that is not in $\lambda^*$ must be removed.

These steps are summarized as follows:

$$\lambda' \leftarrow \{S \mid S \in (\lambda' \cap \lambda^*), pos(sg, S) \in \mathcal{P}(hd, \beta_{exit})\}$$

8. `Compute` $\lambda_{success}$ `from` $\lambda_{call}$ `and` $(\lambda_{success} \cap S_{sg})$: Partition $\lambda_{call}$ into two subsets $\lambda 1_{call}$ and $\lambda 2_{call}$ as follows. $\lambda 1_{call}$ contains only those elements $S$ such that $S \cap S_{sg} = \emptyset$. $\lambda 2_{call} = \lambda_{call} - \lambda 1_{call}$.

$$\lambda_{success} = \{S | (S \in (\lambda 2_{call})^*) \wedge ((S \cap S_{sg}) \in \lambda')\} \cup \lambda 1_{call}$$

We state a proposition similar to the previous one. It essentially says that $\lambda_{success}$ is a safe approximation to the actual success substitution for the subgoal $S_{sg}$.

**Proposition 3** *Given a subgoal* sg *whose abstract call substitution is* $\lambda_{call}$*, a clause* $C$ *whose head* hd *unifies with* sg *and a* safe *abstract exit substitution* $\beta_{exit}$ *(which is compatible with* $\beta_{entry}$ *as computed by the algorithm of section 4 i.e.* $\beta_{exit} \subseteq \beta_{entry}{}^*$*) for* $C$*, let* $\lambda_{success}$ *be the abstract success substitution for* sg *computed using* $C$ *and the above algorithm. Then,* $\lambda_{success}$ *is a safe approximation in the following sense: In the concrete interpretation, let* $\Omega_{success}$ *be the set of success substitutions (computed using the clause* $C$*) corresponding to the set of call substitutions* $\gamma(\lambda_{call})$ *and to exit substitutions* $\gamma(\beta_{exit})$*. Then* $\Omega_{success} \subseteq \gamma(\lambda_{success})$*.* □

**Proof (Outline):** The argument for the correctness of this proposition is very similar to the last one. $\beta'$, which contains all the relevant sharing information (due to $\beta_{exit}$) among the program variables in *hd* is correctly computed in step 1. The groundness and sharing information in $\beta'$ is then conservatively transmitted to the program variables in *sg* in steps 2 to 4. The potential dependency graph computed by the previous algorithm is enhanced by the new sharing and groundness information (if any) in $\beta'$ in step 5. In step 6, a conservative estimate of the projection of $\lambda_{success}$ on *sg* is computed. Some of the sharing information thus computed may be spurious. They may not agree with (1) the sharing information in $\lambda_{call}$ and (2) the argument sharing in *hd* due to $\beta'$. Such spurious sharing information is removed in step 7. Finally, $\lambda_{success}$ is conservatively computed in step 8. □

## 5.2 An Example

We illustrate the above algorithm by a continuation of the previous example. The subgoal $sg$, the head $hd$ (of clause $C$) and the call substitution $\lambda_{call}$ (for $sg$) are as before. Let $\beta_{exit} = \{\emptyset, \{Y_1, Y_7\}, \{Y_3, Y_4\}\}$.

1. `Projection`: $\beta' = \{\emptyset, \{Y_1\}, \{Y_3, Y_4\}\}$

2. `Groundness propagation`: From step 6 of the previous example we get $G = \{X_4, X_5, Y_5, Y_6\}$, $\mathcal{U} = \{(X_1, \{\{Y_1\}\}), (Y_2, \{\{X_2\}\}), (X_3, \{\{Y_3\}\}), (Y_4, \{\{X_3\}\})\}$ and $\lambda = \{\emptyset, \{X_1\}, \{X_3\}, \{X_1, X_2\}\}$. After the execution of this step, we get $G = \{X_2, X_4, X_5, Y_2, Y_5, Y_6\}$ and $\mathcal{U} = \{(X_1, \{\{Y_1\}\}), (X_3, \{\{Y_3\}\}), (Y_4, \{\{X_3\}\})\}$.

3. `update` $\lambda$: $\lambda = \{\emptyset, \{X_1\}, \{X_3\}\}$

4. `Potential dependency graph formation`: $G_{ST} = (V, E)$, where $V = \{X_1, X_3, Y_1, Y_3, Y_4\}$ and $E = \{(X_1, Y_1), (X_3, Y_3), (X_3, Y_4), (Y_3, Y_4)\}$.

5. `Graph partitioning`: The set $S_{sg} - G$ has two elements, $X_1$ and $X_3$ and two partitions $\{X_1\}$ and $\{X_3\}$.

6. Thus, we get $\lambda' = \{\emptyset, \{X_1\}, \{X_3\}\}$

7. `Prune` $\lambda'$ `down to get` $\lambda_{success} \cap S_{sg}$: There are two nonempty set elements in $\lambda'$, which also belong to the set $\lambda$. Therefore they are also in the set $\lambda^*$. Moreover, $pos(sg, \{X_1\}) = \{1\}$ and $pos(sg, \{X_3\}) = \{3, 4\}$. These belong to the set $\mathcal{P}(hd, \beta_{exit}) = \{\{1\}, \{3, 4\}\}$. Thus, no element is removed from $\lambda'$.

8. `Compute` $\lambda_{success}$ `from` $\lambda_{call}$ `and` $(\lambda_{success} \cap S)$: $\lambda 1_{call} = \{\emptyset, \{X_6\}\}$ and $\lambda 2_{call} = \{\{X_1\}, \{X_3\}, \{X_3, X_4\}, \{X_1, X_2, X_7\}\}$. From this, we compute $\lambda_{success} = \{\emptyset, \{X_1\}, \{X_3\}, \{X_6\}\}$.

# 6 Optimization of the Computation of Success Substitutions in Special Cases

As mentioned in section 2, the algorithms described in sections 4 and 5 can together be used to compute the success substitution of a subgoal *sg* given its call substitution and the head *hd* of a clause which unifies with *sg*. However, if it is known that this clause is a "fact" i.e., it doesn't have a body, we can eliminate some of the steps in computing *sg*'s success substitution from its call substitution. Consequently, the optimized algorithm consists of the following steps:

- Steps 1 through 7 of the entry substitution algorithm (section 4), followed by
- Steps 5 through 8 of the success substitution algorithm (section 5)

# 7 A Top-down Driven Fixpoint Computation Algorithm for Abstract Interpretation

In this section, we describe an efficient, top-down driven fixpoint computation algorithm for abstract interpretation. The goal of this algorithm is to build the abstract AND-OR tree for the given program and goal, thus computing the abstract substitutions at all points of this program.

As mentioned in section 2, in building the abstract AND-OR tree for a given program and a goal, the abstract interpreter has to repeatedly execute the *basic* step of computing the *success* substitution of a subgoal whose *call* substitution is given. Given a subgoal $p$, its call substitution $\lambda_{call}$ and clauses $C_1, \ldots, C_m$ whose heads unify with $p$, a naïve approach to executing this basic step would be to build the subtree for $p$ in a top-down fashion:

- Project $\lambda_{call}$ on to the variables in $p$ to obtain $\lambda$, the projected call substitution for $p$.

- For each clause $C_i$, compute its entry substitution using the algorithm in section 4. Compute its exit substitution by recursively computing the success substitutions for each of its subgoals in a left-to-right fashion. Compute $\lambda'_i$, the projected success substitution for $p$ from clause $C_i$, using the algorithm in section 5.

- Compute $\lambda'$, the projected success substitution for $p$ by taking the *least upper bound* (LUB) of $\lambda'_i, 1 \leq i \leq m$. Extend $\lambda'$ to $\lambda_{success}$, the success substitution for $p$.

As also mentioned in section 2, this approach may lead to problems if $p$ or one of its descendents in its subtree is a *recursive* predicate. A situation as shown in figure 2(a) may develop if $p$ is recursive, for example. In this case, a subtree for $p$ has a descendent node which has the same atom ($p$) and the same projected call substitution ($\lambda$). Obviously, this will lead to an infinite loop and $\lambda'$ will never be computed.

The goal of the fixpoint algorithm is to facilitate the computation of $\lambda'$ in such cases without going into an infinite loop. The basic idea behind this algorithm is as follows:

- Compute the *approximate* value of $\lambda'$ using the non-recursive clauses $C_1, \ldots, C_r$ for $p$ and record this value in a *memo table* [10]. Details of this memo table are described in section 7.1.

- Construct the subtree for $p$, using the approximate value of $\lambda'$ from the memo table, if necessary. Note that this computation will not enter into an infinite loop since approximate values of projected success substitutions from the memo table are used for recursive predicates.

- Update the value of $\lambda'$ using $p$'s subtree. This value is "more accurate" than the previous one. Update $p$'s subtree to reflect this change and compute the new value of $\lambda'$ again. Repeat this step until the value of $\lambda'$ doesn't change, i.e., it has reached *fixpoint*.

### 7.1 Details of the memo table

The memo table has an entry for each subgoal with a distinct atom and a distinct (projected) call substitution (modulo renaming of the variables) that occurs in the abstract AND-OR tree.[8] In the context of the fixpoint algorithm described in this paper, the main use of the memo table is to store - possibly incomplete - results (i.e. values of the projections of success substitutions) obtained from an earlier round of iteration. Each entry in this table has four fields:

1. The atom for the subgoal[9]

[8]Normally, the memo table is empty at the start of fixpoint computation. However, if the given program invokes modules which have been pre-compiled, the results of abstract interpretation for these modules can be pre-loaded into the memo table. This saves the work of performing abstract interpretation for these modules again. In addition, it can be assumed that *conceptually* the memo table is preloaded with the entries corresponding to the built-ins, which are marked as *complete*. In practice, however, because of the peculiarities of some non-logical built-ins and since the built-ins are a very important source of information they are treated specially. Note as well that the memo table is also used in the algorithm for storing dependency information. This temporary information will not be present, however, at the end of the analysis.

[9]This field actually contains information about the *unique* ID for the subgoal in the abstract AND-OR tree.

2. The projection of its call substitution on its variables ($\lambda$)

3. The projection of its success substitution on its variables ($\lambda'$)

4. Characterization of the information in the third field i.e., whether it is *complete* or *approximate* or *fixpoint*. These labels are explained in detail in section 7.3. For a nonrecursive predicate, this entry is always *complete*, but for a recursive predicate, it can take on any of the above three values.

The desired output of the algorithm, the abstract AND-OR tree for the given program and query, is *implicitly* contained in the memo table at the termination of the algorithm. Therefore, the memo table *is* the output of the algorithm as presented in section 7.3. Also, in an actual implementation of this algorithm, each entry in the memo table has an additional field indicating the clause in which the subgoal corresponding to this entry occurs and also its position within the clause. On completion of the algorithm, this information gives direct access to the abstract substitutions at all points in the given program. For reasons of space and clarity, the memo table in the algorithm presented in section 7.3 does not have this information.

## 7.2 Overview of the algorithm

This section presents the "core" of our fixpoint computation algorithm. We give a detailed description of this algorithm in section 7.3. Section 7.4 gives an outline of the proof of its correctness.

It assumes that the predicates in the given program have already been classified as *recursive* or *nonrecursive*.[10] As mentioned before, the fundamental step that is executed in this algorithm over and over again is the computation of the success substitution of $p$ given its call substitution, where $p$ is an atom that occurs as a node in the abstract AND-OR tree for the given program and goal.

If the predicate for $p$ is nonrecursive, then it is checked if the memo table has an entry corresponding to (1) the atom for this subgoal (modulo renaming of variables) and (2) $\lambda$.

- If there is such an entry, the value of $\lambda'$ is obtained from this entry and $\lambda_{success}$ is computed by extending $\lambda'$ to all the variables in the clause for the subgoal.

- If there is no such entry, let $C_1, \ldots, C_m$ be the clauses whose heads unify with $p$. The entry and exit substitutions for these clauses and subsequently, $\lambda'_1, \ldots, \lambda'_m$ are computed. $\lambda'_i$ is the projection of the success substitution on its variables for $p$ computed from the exit substitution of the clause $C_i$. $\lambda'$ is computed by taking the *least upper bound* of $\lambda'_i, 1 \leq i \leq m$. A new entry in the memo table is created with the values $p, \lambda, \lambda'$, *complete* and the ID for $p$ for the five fields. $\lambda_{success}$ is computed by extending $\lambda'$ to all the variables in the clause for the subgoal.

If the predicate for $p$ is recursive, then it is checked if the memo table has an entry corresponding to $p$ and $\lambda$.

[10]The algorithm for classifying predicates as recursive or nonrecursive is described in Ullman [25].

- If there is such an entry with the last field's value being *complete*, then $\lambda_{success}$ is computed as before by extending $\lambda'$ to all the variables in the clause for the subgoal. If the last field's value is not *complete*, then we are in the middle of performing fixpoint computation for $p$. The action to be taken in this case is described in detail.[11]

- If there is no entry for $p$ and $\lambda$, let $C_1, \ldots, C_r$ be nonrecursive clauses for $p$. $\lambda'_{fixpoint}$, the approximate value of $\lambda'$ from these clauses, is computed and a new entry in the memo table is created with the values $p$, $\lambda$, $\lambda'_{fixpoint}$, *fixpoint* and the ID for $p$ for the five fields. Let $C_{r+1}, \ldots, C_m$ be the recursive clauses for $p$. $\lambda'_{r+1}$, the projection of $p$'s success substitution due to the clause $C_{r+1}$, is computed. Let the least upper bound of $\lambda'_{fixpoint}$ and $\lambda'_{r+1}$ be $\lambda'_{lub}$. If this is not the same as $\lambda'_{fixpoint}$, then the memo table is updated with $\lambda'_{fixpoint} := \lambda'_{lub}$. This step is repeated for $(r+2), \ldots, m$. If the value of $\lambda'_{fixpoint}$ did not change for $(r+1), \ldots, m$, then fixpoint computation can be stopped. The projection of the success substitution, $\lambda'_{fixpoint}$, that is in the third field for this entry is accurate and so can be labeled *complete*. On the other hand, if the value of $\lambda'_{fixpoint}$ did change during this step, the fixpoint computation is started again with the clause $C_{r+1}$. This step is repeated until $\lambda'_{fixpoint}$ reaches fixpoint.

The following are the main advantages of our algorithm:

- Rather than performing fixpoint computation for the *entire* abstract AND-OR tree in a naïve fashion, our algorithm *localizes* the fixpoint computation only for recursive subgoals. Elsewhere [22], we have described how this leads to a more efficient computation of the abstract AND-OR tree.

- Given a recursive predicate $p$ and its projected call substitution $\lambda$, we start the fixpoint computation by first computing the approximate value of $\lambda'$ from the non-recursive clauses for $p$. This will lead to a faster fixpoint computation than if we had started with $\lambda' = \perp$

- The input and output mode information from builtin clauses like *is/2* is used to increase the *precision* of information that can be obtained from the abstract interpreter.

- Of all the clauses which define the predicate of a subgoal $p$, only those whose heads *unify* (in the concrete domain) with $p$ are used in the computation of the success substitution $\lambda_{success}$ of $p$, given its call substitution $\lambda_{call}$.

### 7.3 Algorithm for fixpoint computation

In this section we present the algorithm for fixpoint computation. The substitutions mentioned in this algorithm are all *abstract* substitutions unless otherwise stated. Because the algorithm is abstract domain independent, certain domain-dependent functions used for unification and other abstract substitution manipulation are left undefined. These functions are described at the points of their occurrence in the algorithm. For the abstract domain described in section 3, these functions have been described in sections 4 and 5.

[11] See the description of the function *lambda_to_lambda_prime* in section 7.3.

The top-level function, *compute_abstract_and_or_tree*, takes as its input arguments the *Program, Query*, and its *call substitution* and returns the *Memo table* that was computed by the fixpoint computation algorithm. The abstract AND-OR tree for the *Program* can be easily derived from this *Memo table*. This function uses the tuple projection function $\pi_2$ which returns the *second* argument of an n-tuple.

**Definition 13 (compute_abstract_and_or_tree)**

compute_abstract_and_or_tree(P,Q,$\lambda_{call}$) =
    $\pi_2$(call_to_success(P,Q,$\lambda_{call}$,{ },{ })) $\square$

The function *call_to_success* computes the success substitution of a goal given its call substitution. Its input arguments are the *Program, Subgoal*, its *call substitution*, the *input Memo table*, and *in_ids*. It returns a 3-tuple *(success substitution, output Memo table, out_ids)*. *in_ids* and *out_ids* are sets of node IDs. More precisely, they are sets of node IDs for which incomplete (i.e. fixpoint) information from the memo table has been used to compute their success substitutions. The difference between *out_ids* and *in_ids* gives the set of node IDs in the subtree of *Subgoal* which used "incomplete" information.

This function uses two abstract domain specific functions, *project* and *extend*. *project* takes as input a *Subgoal* and its *call substitution* and computes its *projected call substitution*. *extend* takes as input a *Subgoal*, its *call substitution*, and its *projected success substitution* and returns its *success substitution*.

**Definition 14 (call_to_success)**

call_to_success(P,S,$\lambda_{call}, M_{in}, Ids_{in}$) =
    $(\lambda_{success}, M_{out}, Ids_{out})$
    where $\lambda_{success}$ = extend(S,$\lambda_{call}, \lambda'$)
    and $(\lambda', M_{out}, Ids_{out})$ = lambda_to_lambda_prime(P,S,project(S,$\lambda_{call}$), $M_{in}, Ids_{in}$) $\square$

The function *lambda_to_lambda_prime* computes the *projected success substitution* ($\lambda'$) of a subgoal $p$ given its *projected call substitution* ($\lambda$). Below, we discuss the five cases it considers:

1. If $p$ is a non-recursive subgoal that has no existing entry in the memo table, then $\lambda'$ is computed by the procedure *nr_lambda_to_lambda_prime*.

2. If $p$ is a recursive subgoal that has no existing entry in the memo table, then fixpoint computation has to be started for this subgoal. A new entry corresponding to $p$ and $\lambda$ is created in the memo table with $\lambda'$ initialized to an appropriate value to start the fixpoint computation (e.g. $\perp$). Then the function *fixpoint_compute* computes $\lambda'$ by performing fixpoint computation on $p$'s subtree. Note that for simplicity the value used in the following description of the *lambda_to_lambda_prime* function is $\perp$ and the clauses are then visited in an unspecified order. In an actual implementation, however, the non-recursive clauses are visited first, thus computing a better first approximation to $\lambda'$. Only then fixpoint computation is started. This speeds up convergence.

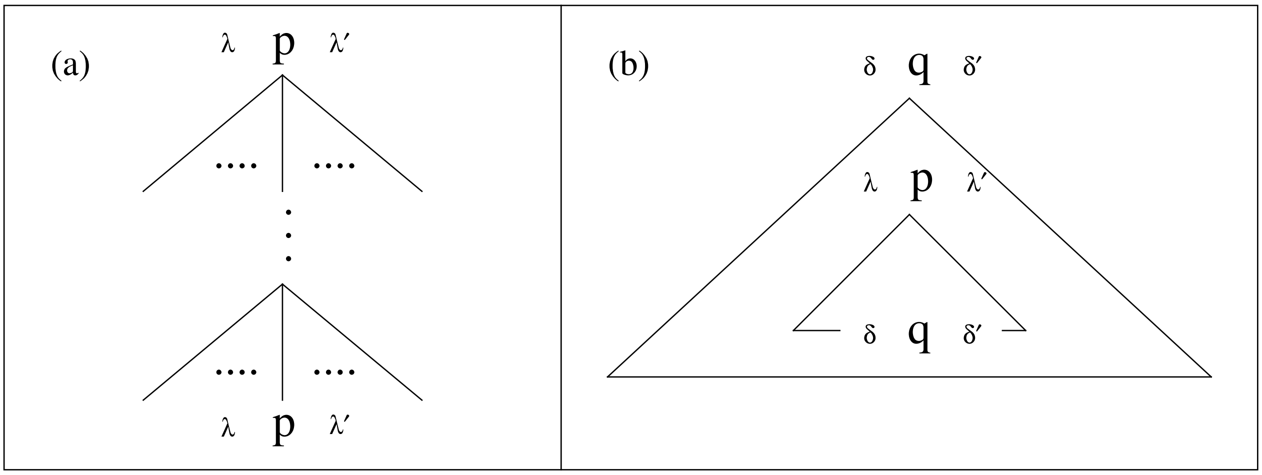


Figure 2: Some situations that arise during fixpoint computation

3. If the memo table has an entry for $p$ and $\lambda$ and this entry has the label *fixpoint*, then the current node for $p$ is the descendent of another node for $p$ with the same $\lambda$ i.e., we are in the process of computing the fixpoint for $\lambda'$ for that node. See figure 2(a). The memo table entry for $\lambda'$ is an approximation to the correct value for both the $p$ nodes. The function *id(p)* returns the unique ID of the subgoal $p$.

4. If the memo table already has a *complete* value for $\lambda'$, *out_ids* is obviously the same as *in_ids* since there are no "incomplete" nodes in the subtree for $p$.

5. If the memo table has an entry for $p$ and $\lambda$ and this entry has the label *approximate*, then the situation is as shown in figure 2(b) i.e., there are two nodes in the tree one of which is the ancestor and the other is the descendent for the current node for $p$. Both these nodes are for the same recursive predicate $q$. The fixpoint computation for $p$ has already been completed but the fixpoint computation for $q$ is not yet over. Since the fixpoint computation for $p$ made use of an *approximate* value of success substitution for $q$, the resultant $\lambda'$ is not accurate. That is why this entry has the label *approximate*. Fixpoint computation for $p$ is started again after this label is changed to *fixpoint* in the memo table. Of course, we now start with a better approximation for $\lambda'$.

**Definition 15 (lambda_to_lambda_prime)**

$$lambda\_to\_lambda\_prime(P, S, \lambda, M_{in}, Ids_{in}) =$$
$$= \begin{cases} if \ \ (S, \lambda, \_, \_) \notin M_{in} \wedge S \ is \ non\text{-}recursive \\ \quad then \ \ (\lambda', M_{out} \cup \{(S, \lambda, \lambda', complete)\}, Ids_{in}) \\ \quad where \ \ (\lambda', M_{out}) = nr\_lambda\_to\_lambda\_prime(P, S, \lambda, M_{in}, P) \\ if \ \ (S, \lambda, \_, \_) \notin M_{in} \wedge S \ is \ recursive \\ \quad then \ \ fixpoint\_compute(P, S, \lambda, M_{in} \cup \{(S, \lambda, \perp, fixpoint)\}, Ids_{in}) \\ if \ \ (S, \lambda, \lambda', fixpoint) \in M_{in} \\ \quad then \ \ (\lambda', M_{in}, Ids_{in} \cup \{id(S)\})) \\ if \ \ (S, \lambda, \lambda', complete) \in M_{in} \\ \quad then \ \ (\lambda', M_{in}, Ids_{in}) \\ if \ \ (S, \lambda, \lambda', approx) \in M_{in} \\ \quad then \ \ fixpoint\_compute(P, S, \lambda, M_{in} \cup \{(S, \lambda, \lambda', fixpoint)\} - \{(S, \lambda, \lambda', approx)\}, Ids_{in}) \ \Box \end{cases}$$

The function *nr_lambda_to_lambda_prime* computes the *projected success substitution* for a non-recursive *Subgoal* given its *projected call substitution.*

**Definition 16 (nr_lambda_to_lambda_prime)**

$$nr\_lambda\_to\_lambda\_prime(P, S, \lambda, M_{in}, Cls) =$$
$$= \begin{cases} if \quad \exists C \in Cls. \ head(C) \ unifies \ with \ S \\ \quad then \ \ (lub(\lambda', \lambda'_C), M_{out}) \\ \quad where \ \ (\lambda'_C, M_{out}, \_) = clause\_lambda\_to\_lambda\_prime(P, S, C, \lambda, M, \emptyset) \\ \quad and \ \ (\lambda', M) = nr\_lambda\_to\_lambda\_prime(P, S, \lambda, M_{in}, Cls - \{C\}) \\ else \\ \quad (\perp, M_{in}) \ \Box \end{cases}$$

The function *clause_lambda_to_lambda_prime* computes the *projected success substitution* $\lambda'_C$ for the subgoal $S$ using the clause $C$ whose head unifies with $S$. It uses the abstract domain specific functions *call_to_entry* and *exit_to_success.* The former computes the entry substitution for $C$ given the $\lambda$ for $S$. The latter computes $\lambda'_C$ for $S$ given the exit substitution for $C$.

**Definition 17 (clause_lambda_to_lambda_prime)**

clause_lambda_to_lambda_prime(P,S,C,$\lambda$, $M_{in}, Ids_{in}$) =

$(\lambda'_C, M_{out}, Ids_{out})$

where $\lambda'_C$ = exit_to_success($\beta_{exit}, S, C, \lambda$)

and $(\beta_{exit}, M_{out}, Ids_{out})$ = entry_to_exit($\beta_{entry}, M_{in}, Ids_{in}$,P,body(C))

and $\beta_{entry}$ = call_to_entry($\lambda, S, C$) $\Box$

**Definition 18 (entry_to_exit)**

$$entry\_to\_exit(\beta_{entry}, M_{in}, Ids_{in}, P, Body) =$$
$$= \begin{cases} if \quad Body = true \\ \quad then \;\; (\beta_{entry}, M_{in}, Ids_{in}) \\ if \quad Body = (Atom, As) \\ \quad then \;\; entry\_to\_exit(\beta_{int}, M_{int}, Ids_{int}, P, As) \\ \quad where \;\; (\beta_{int}, M_{int}, Ids_{int}) = call\_to\_success(P, Atom, \beta_{entry}, M_{in}, Ids_{in}) \\ if \quad Body = Atom \\ \quad then \;\; call\_to\_success(P, Atom, \beta_{entry}, M_{in}, Ids_{in}) \;\; \Box \end{cases}$$

The function *fixpoint_compute* computes the $\lambda'$ of a subgoal $S$ by performing fixpoint computation on its subtree. It does so by applying the fixpoint operator $\uparrow$ to the function *rec_l_to_lp* which traverses the subtree of $S$ once, as described below. $rec\_l\_to\_lp \uparrow \omega$ repeatedly applies the function *rec_l_to_lp* until fixpoint is reached for $\lambda'$.

**Definition 19 (fixpoint_compute)**

fixpoint_compute(P,S,$\lambda$, $M_{in}, Ids_{in}$) =
$(\lambda', M'' - \{(S, \lambda, \_, \_)\} \cup \{(S, \lambda, \lambda', Label)\}, (Ids_{in} \cup Ids_{subtree} - \{id(S)\}))$
where($M''$,Label) = update_abs_ao_tree($M', S, Ids_{subtree}$)
and $(\lambda', M', Ids_{subtree})$ = rec_l_to_lp$\uparrow$ $\omega(P, S, \lambda, (\lambda'_{init}, M_{in}, \emptyset), P)$
where$(S, \lambda, \lambda'_{init}, fixpoint) \in M_{in}$ $\Box$

The application of the fixpoint operator $\uparrow$ to the function *rec_l_to_lp* is made explicit by means of the following definition:

**Definition 20 (rec_l_to_lp$\uparrow$)**

rec_l_to_lp$\uparrow$ $0(P, S, \lambda, (\lambda', M, Ids), P) = (\lambda', M, Ids)$
rec_l_to_lp$\uparrow$ $(n+1)(P, S, \lambda, (\lambda', M, Ids), P)$ = rec_l_to_lp$(P, S, \lambda,$ rec_l_to_lp$\uparrow$ $n(P, S, \lambda, (\lambda', M, Ids), P)$,P)
rec_l_to_lp$\uparrow$ $\omega(P, S, \lambda, (\lambda', M, Ids), P) = (\cup_{n<\omega}\lambda'_n, \cup_{n<\omega}M_n, \cup_{n<\omega}Ids_n)$
where $(\lambda'_n, M_n, Ids_n)$ = rec_l_to_lp$\uparrow$ $n(P, S, \lambda, (\lambda', M, Ids), P)$ $\Box$

As mentioned before, the function *rec_l_to_lp* traverses the subtree once, computing the projected success substitution $\lambda$ from the projected call substitution $\lambda'$ for the recursive case:

**Definition 21 (rec_l_to_lp)**

$$rec\_l\_to\_lp(P, S, \lambda, (\lambda'_{in}, M_{in}, Ids_{in}), Cls) =$$
$$= \begin{cases} if \quad \exists C \in Cls.\; head(C) \; unifies \; with \; S \\ \quad then \;\; rec\_l\_to\_lp(P, S, \lambda, cl\_rec\_l\_to\_lp(P, S, \lambda, (\lambda'_{in}, M_{in}, Ids_{in}), C), Cls - \{C\}) \\ else \\ \quad (\lambda'_{in}, M_{in}, Ids_{in}) \;\; \Box \end{cases}$$

The function *cl_rec_l_to_lp* computes the projected success substitution $\lambda$ from the projected call substitution $\lambda'$ one clause at a time for the recursive case. It uses the abstract domain specific function *lub* which returns the least upper bound of two abstract substitutions.

**Definition 22 (cl_rec_l_to_lp)**

$$
\begin{array}{l}
\mathrm{cl\_rec\_l\_to\_lp}(P,S,\lambda,(\lambda'_{in}, M_{in}, Ids_{in}),C) = \\
\qquad (\lambda'_{out}, M_{out} - \{(S,\lambda,\lambda'_{in}, fixpoint)\} \cup \{(S,\lambda,\lambda'_{out}, fixpoint)\}, Ids_{out}) \\
\qquad \text{where} \quad \lambda'_{out} = lub(\lambda'_{in}, \lambda'_C) \\
\qquad \text{and} \quad\;\; (\lambda'_C, M_{out}, Ids_{out}) = clause\_lambda\_to\_lambda\_prime(P,S,C,\lambda,M_{in},Ids_{in}) \;\; \square
\end{array}
$$

Once fixpoint computation is completed for the subtree of a recursive subgoal $S$, the set of node IDs whose approximate success substitutions were used for this fixpoint computation, $Ids_{subtree}$, is examined. If this contains only the node ID for $S$, then the $\lambda'$ computed for $S$ is labeled *complete*. In this case, it is possible that some other node IDs were "dependent" on this node ID. The dependency information for these nodes is suitably updated by the function *update_depend_set*. If $Ids_{subtree}$ contains node IDs other than *id(S)*, then the $\lambda'$ obtained by the fixpoint computation is labeled *approx*. The dependency information is suitably updated in the memo table by the function *update_abs_ao_tree*.

**Definition 23 (update_abs_ao_tree)**

$$
\begin{array}{l}
update\_abs\_ao\_tree(M_{in}, S, Ids_{subtree}) = \\
= \left\{ \begin{array}{ll}
if & (Ids_{subtree} - \{id(S)\}) = \emptyset \\
 & then\;\; (update\_depend\_set(M_{in}, id(S)), complete) \\
else & \\
 & (M_{in} - \{depend\_set(id(S), \_)\} \cup \{depend\_set(id(S), (Ids_{subtree} - \{id(S)\}))\}, approx) \;\square
\end{array} \right.
\end{array}
$$

**Definition 24 (update_depend_set)**

$$
\begin{array}{l}
update\_depend\_set(M_{in}, Id) = \\
= \left\{ \begin{array}{ll}
if & \exists Id', Set.\;\; depend\_set(Id', Set) \in M_{in} \wedge Id \in Set \\
 & then\;\; update\_depend\_set(M - \{depend\_set(Id', Set)\} \cup \{depend\_set(Id', Set - \{Id\})\}, Id) \\
 & where\;\; M = \\
 & = \left\{ \begin{array}{ll}
 if & Set - \{Id\} = \emptyset \\
  & then\;\; M_{in} - \{(S', \lambda, \lambda', approx)\} \cup \{(S', \lambda, \lambda', complete)\} \\
  & where\;\; S' = id^{-1}(Id') \\
 else & \\
  & M_{in}
 \end{array} \right. \\
else & \\
 & M_{in} \;\square
\end{array} \right.
\end{array}
$$

### 7.4 Outline of the proof of correctness of the algorithm

**Proposition 4** *Given the following:*

- *an abstract domain that satisfies the conditions:*
  - *that the number of distinct (modulo renaming of variables) abstract substitutions for a clause is finite,*
  - *that they form a lattice with respect to a partial order induced by the concretization function*
- *correct, terminating procedures to compute the following:*
  - *abstract entry substitution $\beta_{entry}$ for a clause $C$ given the abstract call substitution $\lambda_{call}$ of a subgoal* sg *which unifies with the head* hd *of $C$*
  - *abstract success substitution for a subgoal* sg *given its abstract call substitution and the abstract exit substitution of a clause $C$ whose head* hd *unifies with* sg
  - *LUB of two abstract substitutions (of the same clause)*

*the fixpoint computation algorithm described above correctly computes the abstract AND-OR tree (i.e., the abstract substitutions at all points) for a given program and goal. Also, it terminates for all inputs.*
□

**Proof (Sketch):** The *correctness* of this algorithm follows from:

- the fact that it computes the abstract projected success substitution $\lambda'$ of a subgoal *sg* as the LUB of the abstract projected success substitutions $\lambda'_i$ computed from the clauses $C_i$, where $C_i, i = 1, \ldots, n$ are all clauses whose heads unify with *sg*.
- the fact that if an atom *sg* with the same projected call substitution ($\lambda$) (modulo renaming of variables) appears in different nodes of the tree, it has the same value for the projected success substitution ($\lambda'$) at these nodes

*Termination:* When the given program has no recursive predicates, it is clear that this algorithm terminates since it builds the abstract AND-OR tree in a top-down fashion and that tree cannot have two nodes with the same atom and projected call substitution (modulo renaming of variables), with one node being the descendent of the other.

When the given program has recursive predicates, the termination of this algorithm follows from:

- the fact that the subtree of a node with a recursive predicate $p$ is finite. Since $p$ can only have a finite number of *distinct* call substitutions, the subtree can only have a finite number of occurrences of nodes who have a variant of $p$ and which themselves have subtrees. All other nodes with $p$ as their predicates use the approximate value of the projected success substitution from the memo table (since they have an ancestor with the same atom and projected call substitution (modulo renaming of variables)) and hence do not have any descendent nodes.

- Given that the subtree of a node with a recursive predicate $p$ is finite, it is easy to see that the complete construction of this subtree takes only a finite number of steps. Broadly speaking, the construction of this tree proceeds as follows: First the approximate value of the projected success substitution is computed as the LUB of the projected success substitutions computed from $p$'s non-recursive clauses. Then the sub-tree is dynamically traversed in a depth-first manner and we return to the root of the subtree. At this time, the value of the projected success substitution is updated as the LUB of the old value and the value computed from $p$'s recursive clauses.

  If there is a change in this value, then the dynamic depth-first traversal is continued again. Note that this "looping" through the depth-first traversal can take place only a finite number of times, since the LUB operation is obviously monotonic and the abstract substitutions for a clause form a *finite* lattice and so the fixpoint will be reached in a finite number of steps.

  If there is no change in the value of the projected success substitution for this node, then its subtree is *complete* and so we have reached the end of fixpoint computation for this node.

□

# 8 Implementation Results

In this section, we present the results of running an implementation of an abstract interpreter which uses the fixpoint algorithm discussed in section 7.3. The goal of this abstract interpreter is to infer the groundness and independence of program variables so that run-time groundness and independence checks can be eliminated for an Independent And-Parallel execution of a given logic program. It takes as input a logic program which also contains a description of the query (or set of queries) and its (their) abstract substitution, provided by the procedure *qmode/2*.[12] It generates a memo table containing the abstract substitutions at all points of the clauses which have been used for building the abstract AND-OR tree for the given query or queries.

This implementation of the abstract interpreter is part of a parallelizing compiler for logic that has proven successful in obtaining speed-ups for a variety of logic programs [12, 14]. Normally, the results of the abstract interpreter are passed to the parallelizing compiler. However, there is an option in this system which enables it to output only the results of the abstract interpreter. Basically, the output is an annotated version of the given logic program, which contains as comments the *abstract call substitutions* of subgoals in all non-unit clauses. Lists are used in the place of sets for abstract substitutions. The results of using such an option on an example program (quicksort using difference lists) are presented in table 1. The first column gives a subgoal (along with the clause in which it occurs) and the second column gives its abstract call substitution.

For reasons of space, we do not show the abstract AND-OR tree for this program. However, we observe from table 1 that, in the body of the second clause for `qsort_dl`, after the execution of `partition(B,A,E,F)`, program variables `A, B, E, F` are ground and `C, D, G, H` are mutually

[12] If the user does not provide a query form, a general one (using a most general abstract substitution) is generated for all entry points which appear in the module declaration for the file.

| Clause / Subgoal | Abstract call substitution |
|---|---|
| `:- qmode(qsort(A,B),[[B]]).` | |
| `qsort(A,B) :-` | |
| `    qsort(A,B,[]).` | `% [[B]]` |
| `qsort([],A,A).` | |
| `qsort([A\|B],C,D) :-` | |
| `    partition(B,A,E,F),` | `% [[C],[D],[E],[F],[G],[H]]` |
| `    qsort(F,G,D),` | `% [[C],[D],[G],[H]]` |
| `    qsort(E,C,[A\|H]),` | `% [[C],[D,G],[H]]` |
| `    G=H.` | `% [[C,H],[D,G]]` |
| `partition([],A,[],[]).` | |
| `partition([A\|B],C,D,[A\|E]) :-` | |
| `    A > C,` | `% [[D],[E]]` |
| `    !,` | |
| `    partition(B,C,D,E).` | `% [[D],[E]]` |
| `partition([A\|B],C,[A\|D],E) :-` | |
| `    A=<C,` | `% [[E],[D]]` |
| `    partition(B,C,D,E).` | `% [[E],[D]]` |

Table 1: Results of abstract interpretation for the quicksort program

independent. Therefore, in an IAP implementation of this program, the subgoals `qsort_dl(F,G,D)` and `qsort_dl(E,C,[A|H])` can be executed in parallel without any groundness or independence checks. It is interesting to note that the same results would have been obtained even if the query form had been the more general "`:- qmode(qsort(A,B),[[A][B]]).`". Also, the use of the abstract interpreter (whose code is not greatly optimized) added 45% to the compilation time, which is considered a reasonable overhead. In fact, this overhead is not worse than that of previous, less precise abstract interpreters [27].

Next, we consider the simpler (from the point of view of analysis) version of quicksort which uses an explicit call to append instead of difference lists. We present the results of abstract interpretation of this program in the form of an abstract AND-OR tree, since it is much simpler than the abstract AND-OR tree of the previous program and it is more illustrative than a table. The `partition/4` predicate used here is the same as the one used in the previous program and the `append/3` predicate is the standard one.

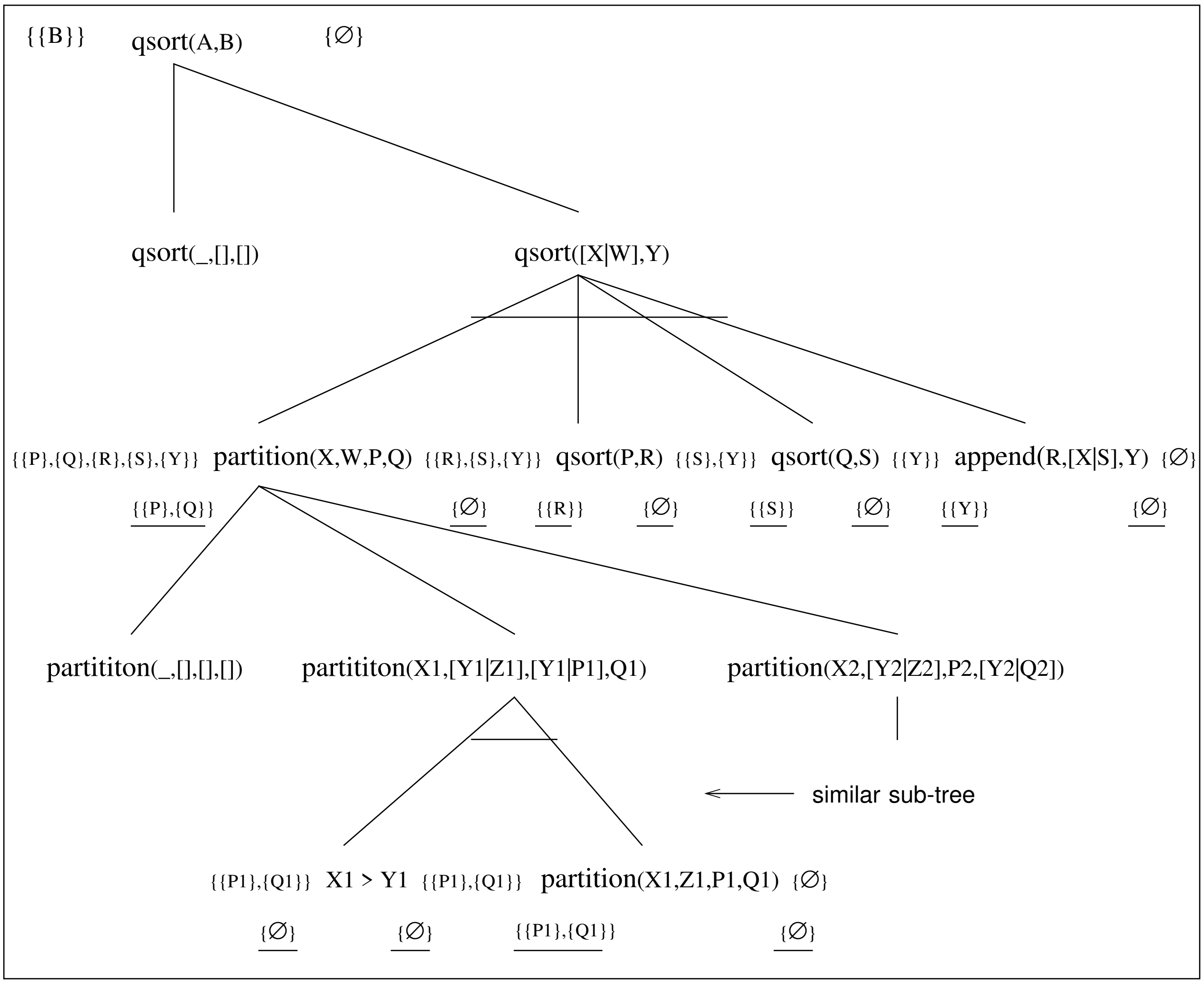


Figure 3: Abstract AND-OR tree for the quicksort program

```
:- qmode(qsort(Xs,Ys),[[Ys]]). %% query and its call substitution

qsort([],[]).
qsort([X|W],Y) :-
        partition(X,W,P,Q),
        qsort(P,R),
        qsort(Q,S),
        append(R,[X|S],Y).
```

The abstract AND-OR tree for this program and query is shown in figure 3. The root of the tree contains the OR-node `qsort(A,B)` with its call substitution to its left and its success substitution to its right. There are three AND-nodes, `qsort([X|W],Y)`, `partition(X1,[Y1|Z1],[Y1|P1],Q1)` and `partition(X2,[Y2|Z2],P2,[Y2|Q2])`. The rest are all OR-nodes. For each OR-node, the call substitution is shown on its left and the success substitution is shown on its right. If there are two adjacent siblings M and N (with M to the left of N), the success substitution for M is the call substitution for N. The projections of the call and success substitutions for a predicate are underlined and are respectively below the call and success substitutions for the predicate.

It can be seen from this tree that the terms bound to `P` and `Q` are ground and the terms bound to `R` and `S` are independent when the subgoal `qsort(P,R)`, which occurs in the body of the recursive clause for `qsort`, is called. Therefore, in an IAP implementation for this program, the subgoals `qsort(P,R)` and `qsort(Q,S)` can be run in parallel without any groundness or independence checks.

# 9 Conclusions and Future Work

Motivated by the needs of applications such as compilation for Independent And-Parallelism (IAP), we have presented an abstract interpreter that is specifically geared towards detecting groundness and independence of terms with a high degree of precision, using a novel abstract domain. We have presented efficient algorithms for computing entry substitutions for clauses and success substitutions for subgoals. These are the essential steps in any algorithm for an abstract interpreter. We have also presented a top-down directed, bottom up fixpoint computation algorithm that is independent of the abstract domain used in the interpreter. The techniques presented in this paper are of direct use in the compilation of logic programs for execution in systems which support IAP, in keeping accurate track of variable aliasing in other types of analysis, and, in general, in any compilation problem which can make use of information regarding variable sharing, groundness, and independence.

We have also presented herein some results from the implementation of an abstract interpreter which uses the algorithms discussed in this paper. This implementation is part of a parallelizing compiler for logic programs using independent and-parallelism and a run-time system which have together proved successful in obtaining speedups for a variety of logic programs [12]. Although a more detailed study of the performance of the interpreter is a subject for further research, our experiments analyzing various benchmarks have revealed that it is more accurate than previous interpreters [27] and it already plays an essential part in achieving the favourable speedup results.

At the same time we have identified ways in which the usefulness of the analysis could be increased. In particular, and in the context of IAP,[13] it would be quite advantageous to enhance the existing abstract domain to include information about the "freeness" of variables. To this end we have developed an abstract domain capable of representing freeness and dependence and developed novel abstract unification algorithms for it. The results from using this enhanced domain will be reported elsewhere.

Finally, based on our design decisions for the algorithm and our experiments with the actual implementation of the abstract interpreter we would also like to suggest a number of other avenues for further research. The structure of the abstract interpretation algorithm as described herein is such

[13] And even more so in the context of non-strict independent and-parallelism (NSIAP), a type of IAP in which goals are allowed to run in parallel even if they share variables, provided that they don't affect each other's search spaces [14].

that an implementation of it requires *double interpretation*, i.e. the given program is interpreted in the abstract domain by the abstract interpreter, which in turn is interpreted (run) by the underlying system. There is a certain degree of inefficiency in doing this and previous experiments with different abstract interpreters ([27], [8]) suggest that eliminating one of the interpretation steps can be advantageous. This can be done by performing a partial evaluation of the abstract interpreter into the program being analyzed, a step which should be done automatically. It would be interesting to compare the performances of the abstract interpretation algorithm presented using both single and double interpretation.

The abstract domain used in this paper can be enhanced to include principal functors of terms. This can improve the accuracy of the results computed by the abstract interpreter. For example, consider the following program:

```
p(X,Y) :- q(X),r(X),s(Y).
q(f(W)).
q(g(a)).
...
clauses for r(X) and s(Y).
```

Suppose that, at entry to `p`'s clause, it is known that `X` is instantiated to a term whose principal functor is `g`. In the simple abstract domain, this information cannot be used in an abstract substitution. If the entry substitution for `p(X,Y)` contains {`X,Y`}, then we can only infer that the call substitution for the subgoal `r(X)` also contains {`X,Y`}, i.e. we cannot infer that `X` is grounded and thus independent of `Y`.[14] But in the enhanced domain, since information is available regarding the principal functor for the instantiation of each program variable, we know that the principal functor of `X` is `g` at entry and so, `X` is ground after the execution of `q(X)`. This translates to a more accurate abstract substitution inferred at this program point.[15] Clearly, this also means increased work for the abstract interpreter. Furthermore, it is not necessary to limit the analysis to first-level structures, and an arbitrary depth bound can be used [27]. It would be interesting to study the tradeoff between greater accuracy and increased compilation time during the abstract interpretation phase brought about by the introduction of different levels of structure depth in the abstract domain.

## 10 Acknowledgements

We would like to thank the anonymous referees for making many useful suggestions which have helped us improve the presentation of this paper. We would also like to thank Fosca Giannotti, Francesca Rossi, Kevin Greene, and the other members of our research groups at MCC, U. of Texas, and U. of Madrid for their useful comments on earlier drafts of this paper and for their support.

[14] For example, in an IAP implementation of this program, the lack of this information would make a run-time independence check needed for the terms bound to `X` and `Y` in order to run the subgoals `r(X)` and `s(Y)` in parallel.

[15] Consequently, in an IAP implementation of this program, the subgoals `r(X)` and `s(Y)` could then be executed in parallel without a run-time independence check, thereby reducing the overhead for parallelism and obtaining a better speed-up.

## References


[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

[3] J.-H. Chang and Alvin M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *International Symposium on Logic Programming*, pages 10–22. IEEE Computer Society, July 1985.

[4] M. Corsini and G. Filè. The abstract interpretation of logic programs: A general algorithm and its correctness. Research report, Department of Pure and Applied Mathematics, University of Padova, Italy, December 1988.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Acm Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[6] S. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. Technical Report 87-24, Dept. of Computer Science, University of Arizona, August 1987.

[7] S. Debray. Static Analysis of Parallel Logic Programs. In *Fifth Int'l Conference and Symposium on Logic Programming*, Seattle,Washington, August 1988. MIT Press.

[8] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[9] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[10] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

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

[12] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[13] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. Technical Report ACA-ST-032-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

[14] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[15] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[16] D. Jacobs and A. Langen, December 1988. Personal communication / Draft.

[17] N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[18] H. Mannila and E. Ukkonen. Flow Analaysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.

[19] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[20] Kim Marriott and Harald Sondergaard. Bottom-up dataflow analysis of logic programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 733–748, Seattle,Washington, August 1988. MIT Press.

[21] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

[22] K. Muthukumar and M.V. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[23] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[24] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[25] J. D. Ullman. *Database and Knowledge-Base Systems, Vol. 1 and 2.* Computer Science Press, Maryland, 1990.

[26] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 700–710, Seattle,Washington, August 1988.

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