Determination of Variable Dependence Information Through Abstract Interpretation

K. Muthukumar
M. Hermenegildo
MCC and The University of Texas at Austin
muthu@cs.utexas.edu, herme@cs.utexas.edu

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 infers 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. The algorithms are illustrated with examples.

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 [3] in their landmark paper. Later, it was shown by Bruynooghe [1], Jones and Sondergaard [12], and Mellish [14] that this technique can be extended to flow analysis of programs in logic programming languages. Numerous specific algorithms for such global analysis in logic programs have been presented ([6], [13], [16], [17], [18], ...). Some have actually been implemented as part of experimental compilers and shown to generate useful results with reasonable overhead, as reported in [18]. Conventional schemes, mostly geared towards optimizing the *sequential* execution of logic programs, generally focus on computing information about the arguments of *predicates* used in 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., the 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 generally mentioned in these methods as a problem to be dealt with in

order to preserve the correctness of the approach, rather than considering it, as we will herein, an important output of the analysis. Therefore, aliasing is treated, if at all, in a very conservative way, as, for example, in [4]. A method, based on keeping track of pairs of variables which can be potentially dependent, is informally proposed in [1] in the context of an example. However, the representation of variable dependency used is relatively weak and the corresponding abstract unification algorithms are not described.

In fact, variable sharing information can often be of the utmost importance for a compiler. For example, such information can be used for compile-time optimization of backtracking [2] and optimization of unification. 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):[1] 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 [8, 9], 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 well worth the effort 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. In addition, this information is also used to guide the selection of goals to be executed in parallel. One special characteristic of this application is that it is specially useful to have such information for all points in the program, rather than globally for each procedure.[3]

The inference of variable sharing and groundness information is the main subject of this paper. Starting with an approach for representing abstract substitutions (in the form of sharing information) suggested to us by Jacobs and Langen [10] 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 variable aliasing. The algorithms are illustrated with examples. We assume that the reader is familiar with logic programming (and Prolog to some extent)

[1] The reader is referred to [9] in these proceedings for a formal definition of IAP and related references.

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

[3] Due to the similarities in the way the search tree is explored by a program executed in IAP and by a program using sequential execution, as shown in [9], conventional abstract interpretation techniques can be applied (with only minor modifications) to programs which are to be evaluated in IAP (Debray presents in [5] an analysis framework for other types of parallelism where the properties of IAP regarding the similarity with sequential execution don't hold). In [18] we reported some results obtained from an abstract interpreter and annotator for IAP (the MA3 system) 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 [18] to be reasonably effective at generating annotations with a reduced number of 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.

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 and are the core of the paper. Section 4 describes how the abstract entry substitution for a clause is computed from the abstract call substitution. An example illustrating this algorithm is given in section 4.2. Section 5 describes how the abstract success substitution for a clause is computed from the abstract exit substitution. Section 7 illustrates our complete algorithm for abstract interpretation with a familiar example. Finally, section 8 summarizes our conclusions.

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* (in a form parallel to the 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* (α) and *concretization* (γ) 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 every x in the concrete computational domain, $u(x)$ is "contained in" $\gamma(u'(\alpha(x)))$.

The input to the abstract interpreter is a set of clauses (the program) and a 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.

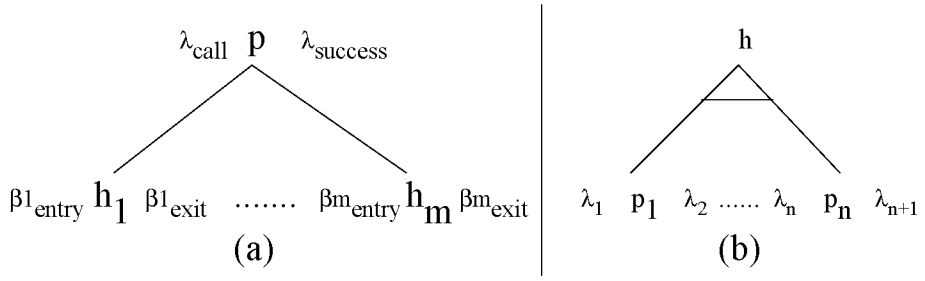

Figure 1: Illustration of the abstract interpretation process

Consider, for example, the clause h :- $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause.

Definition 1 *λ_i and λ_{i+1} are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal p_i. For this same clause, λ_1 is the* abstract entry substitution *(also represented as β_{entry}) and λ_{n+1} is the* abstract exit substitution *(also represented as β_{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. Several frameworks for doing abstract interpretation in logic programs follow along these lines. One such framework is described in detail 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 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 λ_{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 (this amounts to performing "input abstract unification" and the steps required are abstract domain-dependent). 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 each of these clauses (this amounts to performing "output abstract unification," again abstract domain-dependent). 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 h :- $p_1, \ldots, p_n$ and an entry substitution λ_1 $(= \beta_{entry})$, λ_1 is the call substitution for p_1. Its success substitution λ_2 is computed as above. Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. Finally, λ_{n+1} is obtained, which is the exit substitution for this clause. See figure 1(b).

Given this basic framework, it is clear that a particular analysis strategy needs to: (a) define an abstract domain and substitution framework, and the $\sqsubseteq$ relation, (b) 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, and (c) 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* [7] are used in [6, 18] and *stream predicates* are used in [17]. We have developed an efficient fixpoint algorithm for which, due to space limitations, it is not possible to give a precise description herein. However, this algorithm is described in detail in [15]. The basic idea behind the algorithm is as follows:

- First, the clauses which unify with p are selected. This is done using a combination of abstract and concrete unification.[4]
- Compute an *approximate* value of $\lambda_{success}$ using the non-recursive clauses for p and record this value in a *memo table*. Note that this allows faster convergence than starting with $\emptyset$ ($\perp$). Also, the memo table is used in a novel way in that it contains information about the particular goal called and its location within the program so that information is gathered for all points in all clauses, rather than on a per-procedure basis.
- Construct the subtree for p, using the approximate value of $\lambda_{success}$ 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 $\lambda_{success}$ 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 $\lambda_{success}$ again. Repeat this step until the value of $\lambda_{success}$ doesn't change, i.e., it has reached *fixpoint*.

[4] To the best of our knowledge, other abstract interpreters use only *abstract* (rather than also *concrete*) unification to "weed" out candidate clauses. This obviously decreases the precision of the information computed by them.

3 Abstraction Framework

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. The traditional approach 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 herein is somewhat different. Although we may also be interested in knowing characteristics of the the sets of terms that program variables are bound to at run-time (e.g. type information), our main objective herein is the determination of information regarding the *sharing* of variables among the sets of terms to which program variables are bound. I.e., let X and Y be program variables in a clause. We will use an abstract substitution in our abstract interpreter which can tell us whether the sets of terms that X and Y are bound to contain/share any variables. In fact, we see the two classes of information (sharing/type) as complementary: we propose using a combined abstract substitution where a conventional abstract domain is used for gathering term-type information and the techniques described in this paper are used for keeping track of variable dependence.[5] Furthermore, the knowledge of variable dependence information will increase the precision of the whole analysis. For simplicity, however, in this paper we will describe only the part of the abstract substitution responsible for variable dependency information, which, in any case, is sufficient by itself for applications such as automatic generation of IAP.

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

Let θ be a given substitution for a clause C. As defined in [9, 8], a program variable X, which is in C, is *ground* under this substitution if $var(\mathrm{X}\theta) = \emptyset$. Program variables X and Y, which are in C, are *independent* if $var(\mathrm{X}\theta) \cap var(\mathrm{Y}\theta) = \emptyset$. We say that variable V occurs in program variable X under the substitution θ if $\mathrm{V} \in var(\mathrm{X}\theta)$.

[5] Such a system, which combines the techniques presented in this paper with an abstract domain which includes limited-depth concrete terms, as in the MA3 system [18], is proposed in [15].

Below, we formally define the abstract substitution $\mathcal{A}(\theta)$ which corresponds to a concrete substitution θ, to be *a set of sets of program variables* in that clause following [10].[6] Later, we extend it to sets of substitutions. Informally, 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 θ. For the example clause mentioned previously, 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 (represented by the element $\{X, Y\}$) and (2) at least one variable occurs only in t_X (represented by the element $\{X\}$). Thus, a program variable is ground if it does not appear in any set in $\mathcal{A}(\theta)$, and two program variables are independent if 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,* θ *(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:

[6] As mentioned before, the representation that we use for abstract substitutions is essentially the same as suggested to us by Jacobs and Langen [10], from whom defs 2–7 are taken. However, our approach, developed independently, compared to the description in [11] (which we have just received at the time of preparing this final version of our paper) appears quite different: we use a combination fixpoint / top-down strategy which computes only the information that pertains to the particular set of queries to be considered, rather than for all queries. This is, in our opinion, both more useful and efficient. We compute information at all points in the program, rather than per-procedure. Also, our abstract unification is quite different and we believe that our unification algorithms compute the sharing information with considerably higher accuracy. Furthermore, we provide in [15] a concrete algorithm for performing the fixpoint computation. Also, because our approach is more along the lines of traditional analysis for logic programs it can be combined with them as suggested in this section. See [15] for a more detailed comparison.

Definition 6 (Abstraction of a set of substitutions)

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

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

Essentially, α constructs the union of the sharing information found in all substitutions in Θ. Note that, 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. 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 $\lambda_1 \subseteq \lambda_2$ and consequently 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:

- The lattice of abstract substitutions for a clause is finite and hence has a finite depth. This will be used to prove that the fixpoint computation always terminates.
- 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 λ, 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 λ. This is a consequence of the fact that every concrete substitution θ has a finite *range*. Hence, $\emptyset \in \mathcal{A}(\theta)$. From the definition of $\gamma(\lambda)$ it is clear that $\emptyset \in \lambda$.

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

Definition 8 (safe approximation) *Suppose that the concrete set of substitutions that occurs at a point in a clause is* Θ *and the abstract interpreter computes the abstract substitution at this point as* λ. λ *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* Θ, *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* λ, *it should be ground according to all substitutions in* Θ.

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.

Definition 9 *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, i.e.,*

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

Given a subgoal $pred(u_1, \ldots, u_n)$ and an abstract substitution λ, the function $\mathcal{P}(pred(u_1, \ldots, u_n), \lambda)$ computes the dependencies among the argument positions of this subgoal due to λ. 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

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

So, $\mathcal{P}$ converts the dependencies among program variables in λ to dependencies among the argument positions of the predicate $pred/n$.

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 σ and μ be two concrete substitutions. Let λ 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 σ.*

Proof: We note that

$$\begin{aligned} Occ(\mid \sigma \circ \mu \mid_{dom(\sigma)}, X) &= \bigcup_{X \in var(Y\mu)} Occ(\sigma, Y), \quad if\ X \in dom(\mu) \\ &= \bigcup_{X \in var(Y\mu)} Occ(\sigma, Y) \cup Occ(\sigma, X), \quad if\ X \notin dom(\mu) \end{aligned}$$

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

Corollary 1 *Let λ_{call} be the abstract call substitution and $\lambda_{success}$ be the abstract success substitution for a subgoal* sg. *Then $\lambda_{success} \subseteq \lambda^*_{call}$.*

Proof: Let θ_{call} and $\theta_{success}$ be the call and success substitutions for this subgoal. Then there exists a substitution μ (this is the substitution obtained by "solving" the subgoal *sg*) such that $\theta_{success} = \mid \theta_{call} \circ \mu \mid_{dom(\theta_{call})}$. Also $\mathcal{A}(\theta_{call}) = \lambda_{call}$ and $\mathcal{A}(\theta_{success}) = \lambda_{success}$. Therefore,
$\lambda_{success} = \mathcal{A}(\mid \theta_{call} \circ \mu \mid_{dom(\theta_{call})}) \subseteq \lambda^*_{call}$. □

Corollary 2 *Let λ_{call} be the abstract call substitution and $\lambda_{success}$ be the abstract success substitution for a subgoal* sg. *Then $\mathcal{P}(sg, \lambda_{success}) \subseteq (\mathcal{P}(sg, \lambda_{call}))^*$.*

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

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

Proof: Similar to the proofs of Corollaries 1 and 2. □

4 Computing 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 λ_{call} be the call substitution of the subgoal sg. Below we describe the algorithm for computing the entry substitution β_{entry} for the clause C = *hd :- body*.

1. *Projection*: Compute λ by projecting λ_{call} on to the set S_{sg}, i.e.,

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

   λ 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 S_i containing $Z = Term_Z, Z \in S_{sg} \cup S_{hd}$. Form the set $\mathcal{U}$ as follows:

   $$\mathcal{U} \leftarrow \{(Z, Set_Z) \mid Set_Z = var(Term_Z), Z \in \bigcup\nolimits_{1 \leq i \leq n} S_i\}$$

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 $Set1_Z, \ldots, Setk_Z$.

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

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 λ,

   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);

   (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* λ: $\lambda \leftarrow \{S \mid S \in \lambda, S \cap G = \emptyset\}$. This is an update of the call substitution λ 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. $E = E1 \cup E2 \cup E3$ where $E1$, $E2$, and $E3$ 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 λ to the graph G_{ST}).
   - $E2 \leftarrow \{(W, Z) \mid (W, SS) \in \mathcal{U}, S \in SS, Z \in S\}$
     $E3 \leftarrow \{(Z_i, Z_j) \mid (W, SS) \in \mathcal{U}, Z_i \in S1, Z_j \in S2, S1 \in SS, S2 \in SS, Z_i \neq Z_j\}$
     (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* β_{entry}:

$$\beta \leftarrow \bigcup_{i=1}^{r} \wp(HP_i)$$

It is clear that the entry substitution β_{entry} for the clause C is a subset of β.

10. *Prune* β *down to form* β_{entry}: β may contain some sharing information among the arguments of the subgoal predicate that is not compatible with λ. In this step, we remove such "spurious" sharing information from β. Consider $\mathcal{P}(sg, \lambda)$. This gives the sharing information among the arguments of sg due to the abstract substitution λ. 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 β. $\beta_{hd} \leftarrow \{S \mid S \in \beta, pos(hd, S) \in (\mathcal{P}(sg, \lambda))^*\}$. The entry substitution for the clause C is $\beta_{entry} = (\beta_{hd}) \cup \{\{Y_{p+1}\}, \ldots, \{Y_q\}\}$.

Proposition 2 *Let* λ_{call} *be the abstract call substitution for the subgoal* sg *and let* β_{entry} *be the abstract entry substitution for a clause* C *whose head* hd *unifies with* sg. *In the concrete interpretation, let* Ω_{entry} *be the set of entry substitutions for clause* C *computed from* sg*'s set of call substitutions* $\gamma(\lambda_{call})$. *Then,* $\Omega_{entry} \subseteq \gamma(\beta_{entry})$.

Basically, this proposition says that β_{entry}, the computed entry substitution for the clause C, is a safe approximation to its actual abstract entry substitution. A proof sketch can be found in [15].

4.2 An Example

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

sg	$pred(X_1, f(X_2, X_4), X_3, g(X_3), f(X_4, h(X_4)), X_5)$
hd (of clause C)	$pred(p(Y_1), Y_2, q(Y_3, Y_6), Y_4, f(r(Y_5), Y_6), Y_6)$
λ_{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 β_{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.

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

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* λ: $\lambda = \{\emptyset, \{X_1\}, \{X_3\}, \{X_1, X_2\}\}$
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* β *down to form* β_{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 β. To this pruned down β 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 Abstract Success Substitution

In the previous section, we described an algorithm for computing an approximation to the entry substitution β_{entry} for a clause C = *hd* :- *body*, given a subgoal *sg* (which is unifiable with *hd*) and *sg*'s call substitution λ_{call}. In this section we describe an algorithm to compute the success substitution $\lambda_{success}$ for *sg*, given the exit substitution β_{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 $\lambda_{success} = \emptyset$. Otherwise, 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 β' by projecting β_{exit} on to the set S_{hd} (the set of variables in the head *hd*), i.e.,

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

   β' is effectively all the information from β_{exit} that is used in this algorithm.

2. *Groundness propagation*: Start with the values of G,$\mathcal{U}$ and λ at the end of step 6 of the previous algorithm. Let $G2 \leftarrow \{Z \mid Z \in (S_{hd} - G), \forall S(S \in \beta' \Rightarrow Z \notin S)\}$ i.e., $G2$ contains *new* ground program variables in *hd* that were not ground according to β. 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 the next step. Else, execute the *groundness propagation* step (step 5) of the previous algorithm.

3. *Update* λ: Execute step 6 of the previous algorithm.

4. *Potential dependency graph formation*: Execute step 7 of the previous algorithm. Let $E4 \leftarrow \{(Y_i, Y_j) \mid Y_i \in S, Y_j \in S, S \in \beta'\}$. $E4$ contains the *new* sharing information obtained from β'. Update $E \leftarrow E \cup E4$.

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 λ'.

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

   - Consider $\mathcal{P}(hd, \beta_{exit})$. This gives the sharing information among the arguments of *hd* (and hence of *sg* as well) due to the abstract exit substitution β_{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 λ' 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 λ^*. Therefore, any element of λ' that is not in λ^* 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 λ_{call} and $(\lambda_{success} \cap S_{sg})$*: Partition λ_{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 *Let λ_{call} be the abstract call substitution for a subgoal* sg *which unifies with the head* hd *of a clause C and let $\lambda_{success}$ be the abstract success substitution for this subgoal computed using C and the algorithms in sections 4 and 5. In the concrete interpretation, let $\Omega_{success}$ be the set of success substitutions corresponding to the set of call substitutions $\gamma(\lambda_{call})$. Then $\Omega_{success} \subseteq \gamma(\lambda_{success})$.*

5.2 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 λ_{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* λ: $\lambda = \{\emptyset, \{X_1\}, \{X_3\}\}$
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* λ' *down to get* $\lambda_{success} \cap S_{sg}$: There are two nonempty set elements in λ', which also belong to the set λ. Therefore they are also in the set λ^*. 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 λ'.
8. *Compute* $\lambda_{success}$ *from* λ_{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 in the Computation of Success Substitution in Special Cases (Facts)

As mentioned in section 2, the algorithms described in sections 4 and 5 are used in the computation of 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 1–7 of the entry substitution algorithm (section 4), followed by
- Steps 5–8 of the success substitution algorithm (section 5).

7 Example

We illustrate the algorithms in this paper (including the fixpoint algorithm) with a familiar example, the classic quicksort program. The `append/3` predicate used here is the standard one where the third argument is the concatenation of the lists in the first two arguments.

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

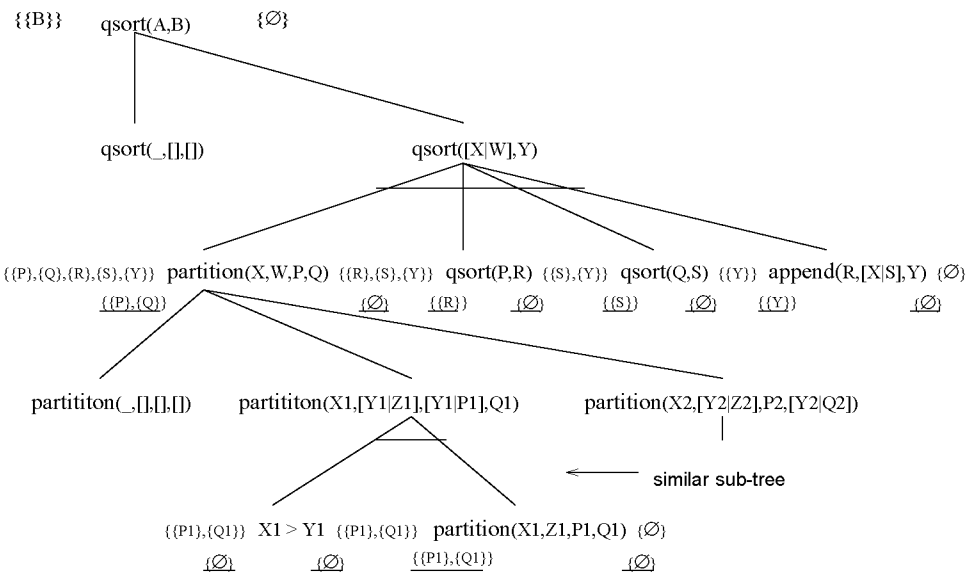

Figure 2: Abstract AND-OR tree for the quicksort program

```
append(R,[X|S],Y).

split(_,[],[],[]).
split(X1,[Y1|Z1],[Y1|P1],Q1) :- X1 >  Y1, split(X1,Z1,P1,Q1).
split(X2,[Y2|Z2],P2,[Y2|Q2]) :- X2 =< Y2, split(X2,Z2,P2,Q2).
```

Let's assume that the query for this program is `:- qsort(A,B)` with the abstract call substitution $\lambda_{call} = \{\{B\}\}$, i.e. `A` is ground but `B` is not. The abstract AND-OR tree for this program and query is shown in figure 2. 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. Initially, the memo table is empty. The predicate `qsort` is recursive and so fixpoint computation is started. The first approximation to $\lambda_{success}$ for `qsort` obtained from its nonrecursive clause is $\emptyset$. This is entered into the memo table. Further computation leads to the construction of the whole tree. 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.

Similar results can be obtained for a quicksort program which uses *difference lists*:

```
qsort(X,Y) :- qsort_dl(X,Y,[]).

qsort_dl([],X,X).
qsort_dl([P|Q],W,Z) :-
        split(P,Q,R,S),
```

```
qsort_dl(R,W,X),
qsort_dl(S,Y,Z),
X = [P|Y].
```

The predicate `split/3` used in this program is the same as the one in the simple quicksort program. For reasons of space, we do not show the AND-OR tree for this program, but it is similar to the one shown in figure 2. Note that whenever `qsort_dl` is called, its first argument is *ground* and the second and the third arguments are *independent* of each other. Hence, `split` is always called with its first and second arguments bound to ground terms. Figure 2 shows that, in such a case, `split` succeeds with its third and fourth arguments bound to ground terms. The consequence of this observation is that, the terms bound to `R` and `S` are ground and the terms bound to `W, X, Y, Z` are independent of each other, when the subgoal `qsort_dl(R,W,X)`, which occurs in the body of the recursive clause for `qsort_dl`, is called. Therefore, in an IAP implementation of this program, the subgoals `qsort_dl(R,W,X)` and `qsort_dl(S,Y,Z)` can be executed in parallel without any groundness or independence checks.

8 Conclusion

Motivated by the needs of applications such as compilation for Independent And-Parallelism, we have presented an abstract interpreter that is specifically geared towards obtaining variable dependence information (including independence and groundness of the terms they are bound to) with a high degree of precision. We have presented novel algorithms for performing abstract unification, i.e. computing entry substitutions for clauses and success substitutions for subgoals, which extensively keep track of such information. These are the essential steps in any algorithm for a top-down abstract interpreter. The techniques presented in this paper are of direct use in the compilation of logic programs for execution in systems which support Independent And-parallelism, and also in any compilation problem which can make use of information regarding variable aliasing, term groundness, and term independence. In addition, as suggested in the paper, these techniques are meant to be combined with more conventional analyses in order to increase their precision and guarantee their correctness.

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] 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.

[3] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approxima-

tion of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[4] 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.

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

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

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

[8] M. 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.

[9] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

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

[11] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. Technical Report TR-89-03, U. of Southern California, Computer Science Department, May 1989.

[12] 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.

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

[14] 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.

[15] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. Technical Report ACA-ST-232-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

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

[17] 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.

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