

An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs

Renganathan Sundararajan
John S. Conery

University of Oregon

Abstract

Static global analyses based on abstract interpretation have been used to derive properties of programs. The analyses differ in the expressiveness of the abstract domains and the precision and efficiency of domain operations. We extend an abstract domain defined by Jacobs and Langen and present new abstract domain operations to derive *freeness*, *groundness*, and *sharing* of variables in logic programs. Analysis of non-trivial programs shows that our method is more precise and more efficient than previous proposals.

Keywords: logic programming, abstract interpretation, groundness analysis, sharing analysis.

Proc. 12th Conference on Foundations of Computer Science and Theoretical Computer Science
(New Delhi, Dec. 18–20), 1992, pp. 203–216.

Lecture Notes in Computer Science 629, Springer-Verlag, Heidelberg, Germany.

1 Motivation, Research Review and Contributions

Most of the recent work in abstract interpretation of programming languages has concentrated on languages that have a large declarative component. Static analyses of logic programs were initially proposed by Bruynooghe [1], Debray [6], Jones and Sondergaard [12] and Mellish [16], and there have been a number of other important contributions to this field [1, 4, 5, 10, 14, 15, 17, 18, 20].

Our use of abstract interpretation is in a compiler for independent AND-parallel logic programs [8, 3]. In programs based on this model, two body goals may be solved in parallel if they do not bind variables in arguments to conflicting terms. The simplest solution to this problem is to test at runtime to make sure two goals have no unbound variables in common. However, run-time tests for independence of terms can be very costly: $\mathcal{O}(mn)$ where m and n correspond to the size of terms.

Three types of static analysis may help us determine at compile time that two goals will be independent:

groundness Obviously, if a term is ground (contains no unbound variables) then it is independent of any other term.

sharing Two nonground terms are independent if they have no variables in common, i.e. no variable occurs in both terms.

freeness Even if a variable occurs as an argument in two goals, the goals may be solved in parallel if one goal does not bind the variable to a nonvariable term; essentially the variable is a free variable of the body of the goal.

Several papers have been published on analyzing procedure calls for common variables. Jacobs and Langen [10] defined an abstract domain named *Sharing* to express variable sharing and independence information. Muthukumar and Hermenegildo [17] extended *Sharing* to derive freeness of variables and suggested an alternative set of abstract domain operations over *Sharing*. Both these analyses conclude that if variables X and Y may have a shared variable and Y and Z may have a shared variable, then X and Z may also have a shared variable. In the abstract interpretation scheme of Jones and Sondergaard, sharing is non-transitive [12]. Treating it as transitive introduces imprecision in sharing propagation. Codish [2] used the abstract domains of Jones and Sondergaard to formally rederive an abstract unification algorithm for the domains. The abstract domains of [12] are less expressive than *Sharing* (see [10] for examples) but sharing propagation is better since it is not transitive. A brief look at the abstract unification algorithms of [12, 2] will show that the time complexity is dominated by transitive closures and hence $\mathcal{O}(n^3)$ in the number of variables in the input terms.

Two projects related to freeness analysis are a static analysis, proposed by Debray and Warren for deriving input and output modes of procedures, which can detect the occurrence of free variables in some cases [6], and a detailed analysis by Winsborough which detects more cases of free variable occurrences [21].

In this paper, we extend Jacobs and Langen's *Sharing* domain to derive groundness, freeness, and sharing of variables more precisely than previous schemes. Although the worst case time complexity of our abstract unification algorithm is exponential in the input size (as are those of [10] and [17]), the actual behavior of our algorithm on realistic programs is better than previous algorithms. The exponential component of our abstract domain operation is executed

rarely, only when terms with repeated variable subterms occur. Such occurrences are not frequent in practical programs, whereas computing transitive closures is an integral part of sharing propagation in [12] and [2].

To substantiate our claim, we defined and implemented a domain-independent abstract interpreter for logic programs [19] and instantiated it with the abstract domains and operations of Jones and Sondergaard [12], Jacobs and Langen [10], and our own extended version of *Sharing*. Analyses of non-trivial programs showed that our method captures variable sharing and independence information more accurately than previous proposals. These analyses also confirm our belief that the efficiency our algorithm (at least for the examples we tested) is much better than other algorithms. Furthermore, analysis times for the complex domains and operations proposed here compare very favorably with the times reported for much simpler analyses, for example Warren, Hermenegildo, and Debray’s global flow analysis [20].

2 Abstract Interpretation

2.1 Definitions and Notation

In abstract interpretation, as in concrete interpretation, we are concerned with the values of variables. An abstract interpreter will bind variables to values from an abstract domain, where each element is an abstraction of one or more values from the corresponding concrete domain. We use $\text{vars}(A)$ to denote the set of variables in A , where A may be a term, body, head, or an abstract domain.

The notion of binding a variable is captured by a mapping from variables to terms. A *substitution* $\theta : \text{Subst} = \text{Var} \rightarrow \text{Term}$ is a set of bindings $\{v_i \mapsto t_i, \dots\}$ where the v_i are distinct logical variables and the t_i are terms and the v_i do not occur in any t_j . The *domain* of a substitution is the set of variables occurring on the left hand sides:

$$\text{dom}(\theta) = \{v_i \mid v_i \mapsto t_i \in \theta \wedge t_i \neq v_i\}$$

and the *range* is the set of terms on the right hand sides:

$$\text{range}(\theta) = \{t_i \mid v_i \mapsto t_i \in \theta \wedge t_i \neq v_i\}.$$

The definition of substitution can be extended from $\text{Var} \rightarrow \text{Term}$ to $\text{Term} \rightarrow \text{Term}$. $t\theta$ is the term obtained by applying substitution θ to a term t , i.e. replacing every variable v_i of t with the corresponding t_i for which there is a binding $v_i \mapsto t_i \in \theta$.

The *composition* of substitutions, denoted $\theta_j \circ \theta_i$, is a function $\lambda x.(x \theta_i) \theta_j$ where $x \in \text{Term}$. Composition associates to the right: $\theta_i \circ \theta_j \circ \theta_k = \theta_i \circ (\theta_j \circ \theta_k)$. The domain of $\theta_i \circ \theta_k$ is $\text{dom}(\theta_i) \cup \text{dom}(\theta_j)$. Composition allows us to define a natural pre-ordering of substitutions: $\theta_1 \leq \theta_2$ iff $\exists \theta_3 \in \text{Subst}$ such that $\theta_2 = \theta_3 \circ \theta_1$.

We consider only *idempotent* substitutions, that is $\theta \circ \theta = \theta$ for any substitution θ . What this implies about the structure of a substitution is that no variable that occurs on the left hand side of some binding also occurs in a term on the right hand side, i.e. $\text{dom}(\theta) \cap \text{vars}(\text{range}(\theta)) = \emptyset$.

A substitution θ is a *unifier* of terms t_1 and t_2 if $t_1\theta = t_2\theta$. The most general unifier (*mgu*) of two terms is a unifier which is least with respect to \leq (modulo variable renaming). Define a function $\text{mgu} : \text{Atom} \times \text{Atom} \rightarrow \text{Subst} \cup \{\text{fail}\}$ that yields the most general unifier of two atoms if it exists and returns *fail* otherwise. We assume that the variables in a program have been renamed so that a variable does not occur in more than one clause. Hence, we can refer unambiguously to the set of program variables $P\text{var}$.

In an abstract interpretation framework it is customary to define a core semantics for a language, leaving some semantic domains and functions unspecified. An interpretation of the core semantics supplies the missing domains and functions. Each domain left unspecified in the core semantics must be a complete lattice, or at least a complete partial order (cpo) [11]. The abstract domains must be of finite height to ensure termination.

A modeling relation between an abstract domain $Asub$ and the corresponding standard domain $\mathcal{P}(Subst)$ is given by two continuous functions. An abstraction function α maps a standard domain to an abstract domain, and a concretization function γ goes in the other direction, mapping an abstract domain to a concrete domain:

$$\begin{aligned}\alpha &: \mathcal{P}(Subst) \rightarrow Asub \\ \gamma &: Asub \rightarrow \mathcal{P}(Subst)\end{aligned}$$

For any $\Theta \in \mathcal{P}(Subst)$ and $\Phi \in Asub$ we have the following *soundness* property:

$$\begin{aligned}\Phi &= \alpha(\gamma(\Phi)) \\ \Theta &\subseteq \gamma(\alpha(\Theta))\end{aligned}$$

For every term \mathbf{x} and function \mathbf{f} of a concrete domain there will correspond a term x and function f of the abstract domain. The result of the abstract application $f(x)$ must yield an abstract object y such that $\gamma(y)$ must describe the set of objects of the standard domain obtained by applying \mathbf{f} to \mathbf{x} . In other words, we have a constraint that for every \mathbf{x} in the standard domain, $\mathbf{f}(\mathbf{x}) \in \gamma(f(\alpha(\mathbf{x})))$.

2.2 Abstract Interpretation of Logic Programs

Before describing abstract interpretation of logic programs, we will review the standard (concrete) interpretation [13]. A goal G , consisting of a set of subgoals $\{a_i, \dots, a_n\}$, is solved in the context of a program P and a current substitution θ (initially empty). A subgoal a_m is selected using a *computation rule*; then a *search rule* is used to find a clause $b \leftarrow b_1, \dots, b_k$ in P such that b unifies with a_m using substitution θ_i (after renaming the variables of the clause). The body literals replace a_m in the goal and the process continues after composing θ_i with the current substitution, i.e. the new set of goals is $(G - \{a_m\} \cup \{b_1, \dots, b_k\})\theta_i \circ \theta$. This process repeats until the set of goals is empty, in which case the initial goal has been successfully solved, or there is a subgoal that does not unify with the head of any of the clauses. When the set of goals is empty, we call the current substitution an *answer substitution*.

Abstract interpretation of a logic program starts with a program P and a set of *entry points*. The entry points can be examples of actual queries together with an initial abstract substitution for the variables in the query, or the entry points may be query patterns. The current substitution is given a different name depending on the state of computation. If a goal is about to be unified with the head of clause, the current substitution is known as the *call substitution* of the goal. If a goal has just been successfully unified with the head of clause, the unifier is the *entry substitution* for the clause; note that the entry substitution is also the call substitution for the first literal in the body. If the last literal in a clause body has just been solved, the current substitution is known as the *exit substitution* for the clause. Restricting the exit substitution of a clause to the non-local variables (variables that do not occur in the clause) and then composing it with the call substitution of the goal gives us the *success substitution* of the clause. The set

of success substitutions of all the clauses whose heads unified with a goal G is the set of answer substitutions of G .

The goal of abstract interpretation is to collect information about each clause in the program, in the form of an exit substitution relative to a given call substitution. For a unit clause, the exit substitution is the same as the entry substitution, i.e. the call substitution composed with the mgu of the head and the call. For a nonunit clause, we need to compute the success substitution of the body literals, using the success substitution of literal i as the call substitution of literal $i + 1$. The success substitution of the last literal in the clause is the exit substitution of the clause. Although our eventual goal is to gather information about programs for independent AND-parallel execution, in which goals are not executed from left to right, for this paper we will use our abstract domains in an abstract interpreter that does a strict left-to-right execution. Our goal here is to compare the precision of the analysis afforded by the different domains. Thus our abstract interpreter uses the success substitution of body literal i as the call substitution of literal $i + 1$. An abstract interpreter that is a front-end to a compiler which generates a goal ordering based on an abstract interpretation is the subject of a current project.

Since a program may contain recursive predicates, abstract interpretation involves fixpoint computation. Standard algorithms for fixpoint computations and their variations can be found in [7, 5, 19, 10, 17], among others. In an abstract interpretation over domains of finite height, we can solve the current set of goals in depth-first order without sacrificing completeness.

3 Abstract Domain A_{Sub}

Our abstract domain A_{Sub} is a triple $\langle Free, Repeat, Sharing \rangle$. Each component of A_{Sub} represents a particular property of a set of concrete substitutions Θ :

- *Sharing* is the same as Jacobs and Langen's domain of the same name. It is a set $\{S_0, \dots, S_n\}$ where each S_i is a set of variables which *may* be unified to terms in the concrete domain that have variables in common by some substitution $\theta \in \Theta$.
- *Free* is a set of variables that are *definitely* free (not bound to non-variable terms) in all $\theta \in \Theta$.
- *Repeat* identifies the set of variables that are bound to concrete terms that *may* have subterms in which a variable occurs more than once.

Information about which variables may be bound to terms with repeated variable subterms allows us to derive a precise abstract unification algorithm which does not treat sharing as a transitive relation.

We now give formal definitions of the components of the abstract domain A_{Sub} . Definitions 1 through 3 are due to Jacobs and Langen [10].

Elements of the abstract domain *Sharing* will contain sets of variables that may have shared subterms in the concrete interpretation:

Definition 1 Abstract Domain *Sharing*

Sharing: $\mathcal{P}(\mathcal{P}(\text{Var}))$

$\sqsubseteq_{\text{Sharing}} = \subseteq$

$X_1 \sqcup X_2 = X_1 \cup X_2$, for $X_1, X_2 \in \text{Sharing}$

□

The partial ordering reflects the fact that *Sharing* is intended to express possible, as opposed to definite, sharing among the variables in any execution.

A substitution θ may have a binding $v \mapsto t$ where t contains a variable u . In this case we say u occurs through v . The function $occ(\theta, u)$ is the set of all variables through which u can occur given substitution θ :

Definition 2

$$occ : Subst \times Var \rightarrow \mathcal{P}(Var)$$

$$occ(\theta, u) = \{v \in dom(\theta) \mid u \in vars(v\theta)\} \quad \square$$

Note that $occ(\theta, u)$ is a set of variables that have a common variable subterm, namely u .

We can use occ to define a function $shared(\theta)$ that gives us all possible sets of variables that may have shared subterms in substitution θ . A set of variables $\{x_1, \dots, x_n\}$ is in $shared(\theta)$ when x_1, \dots, x_n are bound respectively to terms t_1, \dots, t_n in θ and there is at least one common variable that occurs in the terms t_1, \dots, t_n :

Definition 3

$$shared : Subst \rightarrow Sharing$$

$$shared(\theta) = \{occ(\theta, u) \mid u \in Var\} \quad \square$$

Example 1 Let $\theta_0 = \{W \mapsto f(A, C, A), X \mapsto g(A, C), Y \mapsto B, Z \mapsto h(A, B, C, D)\}$.

A quick glance shows that A occurs through the bindings of W , X , and Z . By Definition 2, $W\theta_0 = f(A, C, A)$, $X\theta_0 = g(A, C)$, and $Z\theta_0 = h(A, B, C, D)$; also $A \in vars(W\theta_0)$, $A \in vars(X\theta_0)$, and $A \in vars(Z\theta_0)$, so $occ(\theta_0, A) = \{W, X, Z\}$, as expected.

Applying occ to all variables in θ_0 gives us the value of $shared$:

$$shared(\theta_0) = \{\emptyset, \{W, X, Z\}, \{Y, Z\}, \{Z\}\} \quad \square$$

When a variable x is bound to a ground term in θ , no variable occurs through x and hence x does not appear in any set in $shared(\theta)$. By a slight abuse of terminology, we call such variables “ground variables,” meaning they are variables that are bound to ground terms in the concrete substitution. By extension, if \emptyset is the only element of $sharing(\theta)$, then all variables in $dom(\theta)$ are ground in θ .

Next we discuss our extensions, the domains *Free* and *Repeat*, which together with *Sharing* form the abstract domain *ASub*.

The abstract domain *Free* will be a set of variables that will *definitely* be free at a given program point in all possible executions. In the *Sharing* domain we were satisfied to know if variables could possibly have shared subterms, so set union was used to define the lub operation in that domain. Here, set intersection is the lub operation.

Definition 4 Abstract domain *Free*

$$Free: \mathcal{P}(Var)$$

$$\forall x, y \in Free, \quad x \sqsubseteq_{Free} y \text{ iff } x \supseteq y$$

$$\forall x, y \in Free, \quad x \sqcup y = x \cap y \quad \square$$

A variable in the domain of a substitution is free if it is bound to another variable term. The following function $free(\theta)$ defines the set of variables that are free in θ :

Definition 5

$$free : Subst \rightarrow Free$$

$$free(\theta) = \{x_i \mid x_i \in dom(\theta) \wedge x_i \mapsto x_j \in \theta \wedge x_j \in Var\} \quad \square$$

Note the fact that substitutions are idempotent guarantees that no variables on the right hand sides of bindings $x_i \mapsto x_j$ can also be on the left hand side of some other binding.

Example 2 Using the substitution θ_0 of Example 1, $free(\theta) = \{Y\}$. \square

The domain *Repeat* represents those variables that are bound to terms which may have repeated variable occurrences. A larger set of substitutions may induce a larger set of variables which have repeated variable subterms. Hence, set union is the lub operation on *Repeat*.

Definition 6 Abstract domain *Repeat*

$$Repeat: \mathcal{P}(Var)$$

$$\sqsubseteq_{Repeat} = \subseteq$$

$$X_1 \sqcup X_2 = X_1 \cup X_2, \text{ for } X_1, X_2 \in Repeat \quad \square$$

Function $repeat(\theta)$ returns a set of variables that are bound to terms containing multiple occurrences of at least one variable:

Definition 7

$$repeat : Subst \rightarrow Repeat$$

$$repeat(\theta) = \{x_j \mid x_j \mapsto t_j \in \theta \wedge \exists x_k \in Var \wedge multioccurs(x_k, t_j)\} \quad \square$$

The predicate $multioccurs(v, t)$ is true if v occurs in more than one place in term t .

Example 3 $repeat(\theta_0) = \{W\}$. \square

Note that we lose information about the number of occurrences of repeated variables and the positions in which they occur. This may cause some loss of precision in sharing analysis but is not expected to cause any significant loss in most cases. Keeping this information, on the other hand, would make the domains too concrete and the analyses too costly. Also note that $\forall \theta \ repeat(\theta) \cap free(\theta) = \emptyset$.

Having introduced the domains *Free*, *Repeat* and *Sharing*, we now define *ASub* as the product of these three domains. The partial ordering on and lub of *ASub* are derived from the component domains. Domain subscripts are omitted when there can be no confusion.

Definition 8 Abstract Domain $ASub = Free \times Repeat \times Sharing$

$$\sqsubseteq_{ASub} = \sqsubseteq_{Free} \times \sqsubseteq_{Repeat} \times \sqsubseteq_{Sharing}$$

$$\sqcup_{ASub} = \sqcup_{Free} \times \sqcup_{Repeat} \times \sqcup_{Sharing} \quad \square$$

The initial abstract substitution for the top level goal or entry point with variables x_0, x_1, \dots, x_n has a *Sharing* component of the form $\{\{x_0\}, \{x_1\}, \dots, \{x_n\}\}$, i.e. all variables are independent, *Repeat* is the empty set, and *Free* is the set $\{x_0, x_1, \dots, x_n\}$, i.e. all variables are initially free.

Finally, we define the abstraction and concretization functions that are mappings between the concrete and abstract domains. The abstraction function α maps a set of substitutions to an element of *ASub*. We define α in terms of the abstraction functions $free$, $repeat$ and $shared$ of the component domains *Free*, *Repeat* and *Sharing* respectively.

Definition 9 Abstraction Function

$$\begin{aligned} \alpha : \mathcal{P}(\text{Subst}) &\rightarrow \text{ASub} \\ \alpha(\Theta) &= \bigsqcup_{\theta \in \Theta} \{(free(\theta), repeat(\theta), shared(\theta))\} \end{aligned} \quad \square$$

The abstraction of a set of substitutions Θ includes all possible sharing of variables induced by each substitution $\theta \in \Theta$, since set union is the lub operation for *Sharing*. Likewise, a variable is ground in the abstraction of Θ only if it is ground in all the substitutions $\theta \in \Theta$. The *Repeat* component of the abstraction of Θ includes a variable x if x is bound to a term with a repeated variable occurrence in at least one $\theta \in \Theta$. Finally, a variable is free in an abstract substitution Θ only if it is free in all $\theta \in \Theta$.

The definition of the concretization function γ follows naturally from the above definitions.

Definition 10 Concretization function

$$\begin{aligned} \gamma : \text{ASub} &\rightarrow \mathcal{P}(\text{Subst}) \\ \gamma(A) &= \{\theta \in \text{Subst} \mid (free(\theta) \times repeat(\theta) \times sharing(\theta)) \sqsubseteq A\} \end{aligned} \quad \square$$

It is easy to verify that our definitions of α and γ fulfill the modeling relation between *ASub* and $\mathcal{P}(\text{Subst})$ specified earlier.

4 Operations on the Abstract Domain

In abstract interpretation, as in concrete interpretation, the central operation is unification. In this section we will explain how information in the heads of a called procedure is composed with the current abstract substitution to form the abstract substitution used in the next step. We start with definitions of functions *relevant*, *closure under union* (due to [10]), *restrict*, and *update*, which will be used to construct the abstract unification function.

Our first function allows us to choose a subset of a *Sharing* that has variables in common with a given term. Given a syntactic object t (an element of the domain of concrete terms A) and a *Sharing* S , *relevant*(t, S) returns the subset of S that contains elements that have one or more variables occurring in t :

Definition 11

$$\begin{aligned} relevant : A \times \text{Sharing} &\rightarrow \text{Sharing} \\ relevant(t, S) &= \{X \in S \mid \text{vars}(t) \cap X \neq \emptyset\} \end{aligned} \quad \square$$

Example 4 Recall from Example 1 that

$$\theta_0 = \{W \mapsto f(A, C, A), X \mapsto g(A, C), Y \mapsto B, Z \mapsto h(A, B, C, D)\}$$

and

$$S = \text{sharing}(\theta_0) = \{\emptyset, \{W, X, Z\}, \{Y, Z\}, \{Z\}\}$$

The sets of variables that may share with Z are

$$relevant(Z, S) = \{\{W, X, Z\}, \{Y, Z\}, \{Z\}\} \quad \square$$

Suppose Z , bound to a nonground complex term, now becomes further instantiated as the result of composition of another substitution with the current substitution. Such further instantiations may cause some distinct subterms of Z to share. The closure under union of a *Sharing* S approximates further sharing among the variables of S that may be caused by such compositions.

Definition 12 Closure under union

The closure under union of a *Sharing* S , denoted S^* , is the smallest superset of S such that $X \in S^* \wedge Y \in S^* \Rightarrow X \cup Y \in S^*$. \square

Function *restrict* is used to restrict a given abstract substitution to the variables of a clause or to the variables in the goal's environment.

Definition 13 *restrict*

$restrict : ASub \times \mathcal{P}(Var) \rightarrow ASub$
 $restrict(\langle F, R, S \rangle, V) = \langle F \cap V, R \cap V, \{X' \mid X \in S \wedge X' = X \cap V\} \rangle$ \square

A function we will not define formally is *update*(T, V) which returns a copy of term T with the set of ground variables V replaced by constants.

We can now discuss the main abstract domain operations *abstract_unify*, *entry_substitution*, and *success_substitution*. Function *entry_substitution* abstracts the effect of head unification and is used to propagate information from the abstract environment of a goal to the abstract environment of the clause whose head matched the goal. The abstract operation *success_substitution* is used for propagating information in the other direction, i.e., it abstracts the effect of composing the exit substitution of a clause with the call substitution of a goal and restricting the result to the variables of the goal environment. Most of the work of abstracting head unification and abstracting the composition of exit and call substitutions will be done by the function *abstract_unify*. The technical details of renaming the variables of a clause is taken care of in formal definitions but omitted in the remaining discussions.

4.1 *entry_substitution*

Just before head unification, all the variables of a clause are free, pairwise independent, and not bound to terms with repeated variable subterms. The *entry_substitution* function will compute the component-wise union of this initial abstract substitution (environment) of the clause with the abstract substitution of the goal and use it as the environment in which the goal and the clause head are unified.

Definition 14

$entry_substitution : Atom \times Clause \times ASub \rightarrow ASub$
 $entry_substitution(Goal, Head :- Body, ASub_{call}) =$
let $(H, B) = rename((Head, Body))$
 $F = ASub_{call} \downarrow Free \cup vars(H) \cup vars(B)$
 $R = ASub_{call} \downarrow Repeat$
 $S = ASub_{call} \downarrow Sharing \cup \{\{X\} \mid X \in vars(H) \cup vars(B)\}$
 $Esub = abstract_unify(Goal, H, \langle F, R, S \rangle)$
in $rename^{-1}(restrict(Esub, vars(H) \cup vars(B)))$ \square

The notation $S \downarrow T$ is the projection of S onto T . The call to *abstract_unify* implements abstract head unification, and the updated abstract substitution is restricted to the clause variables and is used for solving the body goals.

4.2 *success_substitution*

Function *success_substitution* is similar to *entry_substitution*. It uses *abstract_unify* to compose the exit substitution and the call substitution by abstractly unifying the head and the goal in the environment which is the component-wise union of the abstract exit substitution of the clause and the abstract call substitution of the goal. The success substitution of a goal is then just the restriction of the resulting unified substitution to the variables in the environment of the goal.

Definition 15

$$\begin{aligned}
& \textit{success_substitution} : \textit{Atom} \times \textit{Clause} \times \textit{ASub} \times \textit{ASub} \rightarrow \textit{ASub} \\
& \textit{success_substitution}(\textit{Goal}, \textit{Head} \textit{:} \textit{Body}, \textit{ASub}_{\textit{call}}, \textit{ASub}_{\textit{exit}}) = \\
& \textbf{let} \quad (\textit{H}, \textit{B}, \textit{ASub}_{\textit{exit}}) = \textit{rename}'((\textit{Head}, \textit{Body}, \textit{ASub}_{\textit{exit}})) \\
& \quad \textit{ASub}_{\textit{both}} = \textit{ASub}_{\textit{exit}} \cup \textit{ASub}_{\textit{call}} \\
& \quad \textit{ASub}_{\textit{succ}} = \textit{abstract_unify}(\textit{Goal}, \textit{H}, \textit{ASub}_{\textit{both}}) \\
& \textbf{in} \quad \textit{restrict}(\textit{ASub}_{\textit{succ}}, \textit{Pvar})
\end{aligned}$$

□

4.3 *abstract_unify*

The *abstract_unify* function updates an abstract substitution $\textit{ASub}_{\textit{in}}$ for a head and a goal to reflect the effects of their unification. It first unifies the head and the goal in an empty environment to obtain an idempotent substitution which can be thought of as a set of equations $E : \{x_1 = t_1, \dots, x_n = t_n\}$ [9]. These equations represent the unifications to be performed in the *current* environment whose abstraction is $\textit{ASub}_{\textit{in}}$. All we have to do is to safely abstract the effect of each of these equations on the abstract substitution. In the rest of the paper, we will refer to an mgu as a set of equations and each binding in the mgu as an equation.

Since variables bound to ground terms are not free, do not have subterms with shared variables, and do not have repeated variable occurrences, it is convenient to identify the ground variables defined by the abstract substitution and eliminate them from the abstract substitution and the unification equations. The function *propagate_frs* (defined later) will use the remaining equations $x_i = t_i$ to update the abstract substitution. Since *propagate_frs* performs a safe but imprecise approximation in the presence of variables bound to terms with repeated variable subterms, explicit groundness propagation and elimination of ground variables from the abstract substitution improves the precision of our analysis.

Definition 16

$$\begin{aligned}
& \textit{abstract_unify} : \textit{Atom} \times \textit{Atom} \times \textit{ASub} \rightarrow \textit{ASub} \\
& \textit{abstract_unify}(\textit{Goal}, \textit{Head}, \textit{ASub}_{\textit{in}}) = \\
& \textbf{let} \quad \theta = \textit{mgu}(\textit{Goal}, \textit{Head}) \\
& \quad \textit{Ground}_0 = (\textit{vars}(\textit{Goal}) \cup \textit{vars}(\textit{Head})) \setminus \textit{vars}(\textit{ASub}_{\textit{in}} \downarrow \textit{Sharing}) \\
& \quad \textit{Ground}_1 = \textit{Ground}_0 \cup \{X \mid V_i \mapsto T_i \in \theta \wedge V_i \in \textit{Ground}_0 \wedge X \in \textit{vars}(T_i)\} \\
& \quad \textit{Ground} = \textit{Ground}_1 \cup \{V_i \mid V_i \mapsto T_i \in \theta \wedge \textit{vars}(T_i) \subseteq \textit{Ground}_1\} \\
& \quad \theta_0 = \{x_i \mapsto t_i \mid x_i \mapsto t'_i \in \theta \wedge x_i \notin \textit{Ground} \wedge t_i = \textit{update}(t'_i, \textit{Ground})\} \\
& \quad \textit{Sharing}_0 = \textit{ASub}_{\textit{in}} \downarrow \textit{Sharing} \setminus \textit{relevant}(\textit{Ground}, \textit{ASub}_{\textit{in}} \downarrow \textit{Sharing}) \\
& \quad \textit{Repeat}_0 = \textit{ASub}_{\textit{in}} \downarrow \textit{Repeat} \setminus \textit{Ground} \\
& \quad \textit{Free}_0 = \textit{ASub}_{\textit{in}} \downarrow \textit{Free} \setminus \bigcup \textit{relevant}(\textit{Ground}, \textit{ASub}_{\textit{in}} \downarrow \textit{Sharing}) \\
& \textbf{in} \quad \textit{propagate_frs}((\textit{Free}_0, \textit{Repeat}_0, \textit{Sharing}_0), \theta_0)
\end{aligned}$$

□

Example 5 Unify goal $p(X, f(Y), Z, A)$ with head $p(f(Q, P), f(f(R, 2)), f(T, S), P)$, given current abstract substitution $\langle Free, Repeat, Sharing \rangle$ where:

$$Free = \{P, Q, R, S, T\}$$

$$Repeat = \{Z, A\}$$

$$Sharing = \{\emptyset, \{Y\}, \{Y, Z\}, \{Y, B\}, \{A\}, \{P\}, \{Q\}, \{R\}, \{S\}, \{T\}\}$$

Unifying the head and the goal yields equations

$$\{X = f(Q, P), Y = f(R, 2), Z = f(T, S), A = P\}$$

X is ground because it does not appear in the *Sharing* component of the abstract environment. Any term equated to X must also be ground, so $X = f(Q, P)$ implies that Q and P are also ground, and further A is also ground since $A = P$. Thus the set of ground variables is $\{A, P, Q, X\}$. We can use this information to refine the abstract substitution by removing the sets $\{A\}$, $\{P\}$, and $\{Q\}$ from the *Sharing* component, A from the *Repeat* component and P and Q from the *Free* component. The updated abstract substitution is:

$$Free = \{R, S, T\}$$

$$Repeat = \{Z\}$$

$$Sharing = \{\emptyset, \{Y\}, \{Y, B\}, \{Y, Z\}, \{R\}, \{T\}, \{S\}\}$$

The remaining unification equations are $\{Y = f(R, 2), Z = f(T, S)\}$. □

We have considered the effect of grounding variables through head unification. What is left is the approximation of the effect of the remaining equations on the abstract substitution. This is done by the function *propagate_frs*, which is discussed next.

4.4 *propagate_frs*

The main part of the abstract unification algorithm, *propagate_frs*, updates the current abstract substitution by safely approximating the effects of each equation generated by head-goal unification in the empty environment.

Definition 17

$$propagate_frs : ASub \times Subst \rightarrow ASub$$

$$propagate_frs(\langle F, R, S \rangle, \theta) =$$

$$\mathbf{if} \ \theta = \{\} \ \mathbf{then} \ \langle F, R, S \rangle$$

$$\mathbf{else}$$

```

let  ( $V_i = T_i$ )  $\in$   $\theta$ 
       $\theta' = \theta \setminus \{V_i = T_i\}$ 
       $A = \text{relevant}(V_i, S)$ ,  $B = \text{relevant}(T_i, S)$ 
       $B' = \text{if } V_i \in R \text{ then } B^* \text{ else } B \text{ fi}$ 
       $A' = \text{if } (\text{vars}(T_i) \cap R \neq \emptyset \vee \exists X_j \text{ s.t. } \text{multioccurs}(X_j, T_i))$ 
        then  $A^*$  else  $A$  fi
       $S' = (S \setminus (A \cup B)) \cup \text{pairwise union of } A' \text{ and } B'$ 
       $R' = R \cup (\text{if } V_i \in R \text{ then } \bigcup \text{relevant}(T_i, S) \text{ else } \emptyset \text{ fi})$ 
       $R'' = R' \cup (\text{if } (\text{relevant}(T_i, S) \cap R \neq \emptyset \vee \exists X_j \text{ s.t. } \text{multioccurs}(X_j, T_i))$ 
        then  $\bigcup \text{relevant}(V_i, S)$  else  $\emptyset$  fi)
       $R''' = R'' \cup \text{relevant}(\bigcup\{X \cap Y \mid X \in A', Y \in B'\}, S)$ 
       $F' = \text{Free} \setminus (\text{if } V_i \notin F \text{ then } \text{relevant}(S', T_i) \text{ else } \emptyset \text{ fi})$ 
       $F'' = F' \setminus (\text{if } (T_i \notin \text{Var} \vee T_i \notin F') \text{ then } \text{relevant}(S', V_i) \text{ else } \emptyset \text{ fi})$ 
in   $\text{propagate\_frs}((F'', R''', S'), \theta')$ 
fi

```

□

An outline of a proof of correctness of *abstract_unify* is given in the Appendix.

To propagate information about sharing, repeating, and freeness for an equation, we begin with the *Sharing* component. After the unification defined by the equation $v_i = t_i$, any variables relevant to the current value of v_i may share with any variable relevant to the current value of t_i . Assume that v_i and t_i do not have repeating variable sub-terms according to the current abstract substitution. We compute the pairwise union of each possible pair of sets of variables from $\text{relevant}(v_i, \text{Sharing})$ and $\text{relevant}(t_i, \text{Sharing})$ and add these to *Sharing* (since this information supersedes the old relevant sets, they are removed).

Next we consider how repeated variables impacts sharing. The effect of an equation $v_i = t_i$ depends on whether or not v_i (or a variable in t_i) is already bound to some term which has one or more repeated variable occurrences. This leads to four possibilities, depicted in Figure 1. A broken edge from v_i to t_i represents the fact that the equation unifies the two terms. A solid edge from v_i to itself indicates that v_i may have a repeated variable occurrence in the current substitution; similarly a solid edge from a node t_i to itself means the term t_i has one or more variables that occur more than once in the term. For both types of nodes, the variables we are concerned with are those that occur in the term and the variables it may be aliased to, i.e. the relevant variables as defined by the current *Sharing*. In the figure, the set (X_1, \dots, X_n) associated with a variable node are all the variables in $\text{relevant}(v_i, \text{Sharing})$, and the variables Y_1, \dots, Y_k are the variables in $\text{relevant}(t_i, \text{Sharing})$.

We have already considered case (a). Consider case (b) in the figure. Some Y_i has a repeated subterm, as indicated by the solid arc. The equation $v_i = t_i$ implies that some of the variables X_1, \dots, X_n may now depend on each other as the result of the repeated variable somewhere in t_i . To handle this case, we compute the closure under union of $\text{relevant}(v_i, \text{Sharing})$ and add the pairwise union of this set and $\text{relevant}(t_i, \text{Sharing})$ to *Sharing*. Case (c) is symmetric to case (b), the only difference being that the repeated variable is one of the X_i and the set to close under union and add to *Sharing* is $\text{relevant}(t_i, \text{Sharing})$. Finally, in case (d), since there may be repeated variable subterms in v_i and t_i , both the sets $\text{relevant}(v_i, \text{Sharing})$ and $\text{relevant}(t_i, \text{Sharing})$ are closed under union and their pairwise union is added to *Sharing*. As an example of case (c), consider equation $Z = f(T, S)$ of Example 5. Since Z may have a repeated

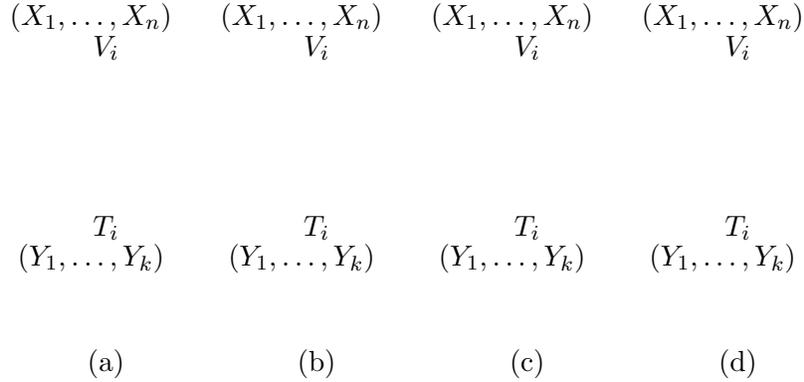


Figure 1: Sharing, Freeness and Repeatedness Propagation

variable occurrence, T and S may be brought to share. This is accounted for by taking the closure under union of $\{\{T\}, \{S\}\} = \{\{T\}, \{S\}, \{T, S\}\}$.

When updating *Repeat* for an equation $v_i = t_i$, we note that if v_i may be bound a term which has repeated variable subterms, then t_i now also has repeated variable subterms. Since we do not know which of the variables of v_i repeat, and since we are deriving *potential* repeatedness information, we have to conclude that any subterm of t_i may also have repeated variable subterms. Returning again to the equation $Z = f(T, S)$ and the case where we know Z has repeated variable subterms, we conservatively conclude that T and S may also be bound to terms with repeated variable subterms and add them to *Repeat*.

Even when neither v_i nor t_i has a repeated variable subterm, the equation $v_i = t_i$ may cause two previously distinct variable subterms to become aliased. Consider the equation $P = R$, given substitution $\{P \mapsto (T, U), Q \mapsto (T, U), R \mapsto (U, N)\}$ and the sharing component of its abstraction $\{\{P, Q\}, \{P, R\}, \{R\}\}$. Two previously distinct components T and U in the value of P are now aliased to each other and P , Q , and R will now have a repeated variable occurrence. Such cases can arise only when there is at least one variable (U in this situation) that occurs through both v_i and t_i . We must conclude that v_i , all the variables in t_i , and variables that share with them may be bound to a term with repeated variable subterms.

Finally, to propagate freeness information, we need to remove variables from *Free* that are unified with ground terms or are unified with other variables that are not free.

5 Comparison to Other Approaches

To compare the precision and efficiency of our abstract domain with others, we designed and implemented a domain-independent abstract interpreter which accepts as parameters an abstract domain and its operations. We analyzed five different programs with our domain (*SR*), the domain of Jacobs and Langen (*JL*), and the domain of Jones and Sondergaard (*JS*). The five programs were

- *Grammar*: Generates and recognizes a very small set of English sentences (15 clauses).

Program	No of Pairs of Shared Variables			Analysis Times, in secs using SICStus Prolog 0.7		
	SR	JS	JL	SR	JS	JL
<i>Grammar</i>	6	6	6	0.9	5.3	0.9
<i>Bid</i>	0	0	5	4.3	5.6	4.5
<i>Deriv</i>	0	0	0	21.1	25.8	20.2
<i>Read</i>	23	29	42	47.6	123.9	47.6
<i>RdTok</i>	37	41	60	129.1	568.8	415.8

Table 1: Precision and Efficiency of 3 systems. Analysis Times on a HP 9000/835.

- *Bid*: Computes the opening bid for a bridge hand (51 clauses).
- *Deriv*: Performs symbolic differentiation and simplifies the derivative (61 clauses).
- *Read*: Warren and O’Keefe’s public domain Prolog parser (87 clauses).
- *Rdtok*: O’Keefe’s public domain tokenizer for Prolog (54 clauses).

Table 1 shows the results of our experiment. The precision of an analysis is measured in terms of the number of pairs of program variables reported to be shared by a method. Since the analyses are aimed at deriving *potential* sharing, an analysis is more precise when it derives a smaller set of such variables, i.e. it reports only those variables that do in fact share. Our analyzer, SR, found the smallest number of pairs of sharing variables in all cases. JS was a close second, and JL was not as precise as the other two. In the case of two widely used practical programs, the number of pairs of sharing variables derived by JL was larger by about 82% and 62% than that of SR and 45% and 46% larger than those of JS.

Our method is the most efficient, taking less time to analyze all but one program. Contrary to popular belief that more precise analyses are necessarily less practical, the improved precision of our analysis actually contributes to faster execution. JS, although almost as precise as ours, is two to three times slower in the case of *Read* and about five times slower in the case of *Rdtok* and *Grammar*. This is mainly due to the transitive closures computed by JS when approximating unification. Computing transitive closures is a central part of Jones and Sondergaard’s technique for propagating sharing and cannot be avoided. In our domain, the operation *closure under union* has a higher worst case time complexity (exponential in the number of elements in the set, as opposed to $\mathcal{O}(n^3)$ for transitive closure) but it is avoided most of the time because of the expressiveness of our abstract domain and the matching precision of our abstract domain operations. The fact that Jacobs and Langen’s analysis was less precise than ours is also reflected in the analysis times. JL is slower by three and a half times in the case of *Rdtok* where its precision is the lowest.

The analysis times of our abstract interpreter compare favorably with those reported in the literature. For example, the *MA*³ and *Ms* interpreters required 60.2 seconds using Quintus Prolog-2.2 on a Sun3/50 to analyze *Read* [20], whereas ours took 47.6 seconds using SICStus 0.7 on a somewhat faster HP9000/835 system. There is, however, a point worth noting. Our abstract interpreter, written in Prolog, interprets the program to be analyzed. The WAM (Warren Abstract Machine) code of our abstract interpreter is itself interpreted by the WAM emulator of SICStus Prolog. Hence, our analyzer pays the overhead of double interpretation. The *Ms* interpreter transforms the programs to be analyzed into new programs [20] which get executed

directly by the underlying Prolog system and thus avoids the overheads. Despite the overhead, our abstract interpreter's analysis times are comparable.

6 Conclusions

The feasibility of data flow analysis of logic programs based on abstract interpretation has been well established. In this paper we showed that it is possible to define more informative and efficient analyses through the use of more precise information in the abstract domain. Although the worst case behavior of our algorithm is exponential in the number of shared variables, the observed behavior on practical programs is very efficient.

References

- [1] BRUYNNOOGHE, M. A Framework for the Abstract Interpretation of Logic Programs. Tech. Rep. CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.
- [2] CODISH, M., AND YARDENI, E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *Proceedings of the Eighth International Conference on Logic Programming* (1991).
- [3] CONERY, J. S. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Boston, MA, 1987.
- [4] DEBRAY, S. K. Static analysis of parallel logic programs. In *1988 Joint Conference on Logic Programming* (1988), pp. 711–733.
- [5] DEBRAY, S. K. Static inference of modes and data dependencies. *ACM Transactions of Programming Languages and Systems* 11, 3 (July 1989), 418–450.
- [6] DEBRAY, S. K., AND WARREN, D. S. Automatic mode inference for prolog programs. In *Proceedings of the 1986 Symposium on Logic Programming* (1986).
- [7] HECHT, M. S. *Flow Analysis of Computer Programs*. Programming Language Series. North-Holland, New York, 1977.
- [8] HERMENEGILDO, M. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, University of Texas at Austin, August 1986.
- [9] J-L. LASSEZ, M.J. MAHER, K. M. *Unification Revisited*. Morgan Kaufmann Publishers, Inc, 1988, ch. 15, pp. 587–626.
- [10] JACOBS, D., AND LANGEN, A. Accurate and efficient approximation of variable aliasing in logic programs. In *Proceedings of the North American Conference on Logic Programming* (1989), pp. 154–165.
- [11] JONES, N., AND MYCROFT, A. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of 13th POPL* (1986), ACM.

- [12] JONES, N., AND SONDERGAARD, H. A semantics based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood Limited, 1987, ch. 6, pp. 123–142.
- [13] LLOYD, J. W. *Foundations of Logic Programming*, first ed. Springer-Verlag, Berlin, 1984.
- [14] MANILLA, H., AND UKKONEN, E. Flow analysis of prolog programs. In *4th IEEE Symposium on Logic Programming* (september 1987), IEEE Computer Society.
- [15] MARRIOTT, K., AND SONDERGAARD, H. Bottom-up dataflow analysis of normal logic programs. In *5th International Conference on Logic Programming* (August 1988).
- [16] MELLISH, C. S. Abstract interpretation of prolog programs. In *Third International Conference on Logic Programming* (july 1986), pp. 463–475.
- [17] MUTHUKUMAR, K., AND HERMENEGILDO, M. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming* (1991).
- [18] SONDERGAARD, H. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of European Symposium on Programming* (1986), Springer-Verlag.
- [19] SUNDARARAJAN, R. An introduction to abstract interpretation of logic programs. Tech. Rep. CIS-TR 91-05, Dept of Computer and Information Science, University of Oregon, Eugene, Oregon, U.S.A. 97403, 1991.
- [20] WARREN, D. H. D., HERMENEGILDO, M., AND DEBRAY, S. K. On the practicality of global flow analysis of logic programs. In *1988 Joint Conference on Logic Programming* (1988), pp. 684–699.
- [21] WINSBOROUGH, W. *Automatic, Transparent Paralellization of Logic Programs at Compile Time*. PhD thesis, Dept. of Computer Science, University of Wisconsin-Madison, Sept. 1988.

Appendix A: Correctness

The correctness of our analysis can be established by proving the correctness and monotonicity of the *entry_substitution* and *success_substitution* functions and the following main theorem about *abstract_unify*.

Theorem 1

$$\{mgu(h\theta, g\sigma)\} \subseteq \gamma(\text{abstract_unify}(g, h, \alpha(\theta) \cup \alpha(\sigma))) \quad \text{where } \text{dom}(\theta) \cap \text{dom}(\sigma) = \emptyset \quad \square$$

The condition $\text{dom}(\theta) \cap \text{dom}(\sigma) = \emptyset$ reflects the fact that the clause head h and the goal g are renamed apart before unification. To prove the above, it is enough show that

$$\alpha(\{mgu(h\theta, g\sigma)\}) \sqsubseteq \alpha(\gamma(\text{abstract_unify}(g, h, \alpha(\theta) \cup \alpha(\sigma))))$$

with the same proviso that $\text{dom}(\theta) \cap \text{dom}(\sigma) = \emptyset$. From our definitions of α and γ it can be shown that $\alpha(\gamma(A)) = A$, which simplifies the above equation to

$$\alpha(\{mgu(h\theta, g\sigma)\}) \sqsubseteq \text{abstract_unify}(g, h, \alpha(\theta) \cup \alpha(\sigma))$$

The above condition can be stated in terms of the component domains and their partial orderings, as follows (*LHS* and *RHS* mean the left and right hand sides of the above equation):

$$\begin{aligned} \text{LHS}\downarrow\text{Free} &\supseteq \text{RHS}\downarrow\text{Free} \\ \text{LHS}\downarrow\text{Repeat} &\subseteq \text{RHS}\downarrow\text{Repeat} \\ \text{LHS}\downarrow\text{Sharing} &\subseteq \text{RHS}\downarrow\text{Sharing} \end{aligned}$$

The following outline supports the above three conditions.

1. For any θ , $\text{ground_variables}(\theta) \cap \text{Free}(\theta) = \emptyset$ and $\text{ground_variables}(\theta) \cap \text{Repeat}(\theta) = \emptyset$ and $\text{ground_variables}(\theta) \cap \text{vars}(\text{Sharing}(\theta)) = \emptyset$ from the definitions of *Free*, *Repeat* and *Sharing*.
2. For any θ , it can be proved from the definition of the function *shared* that the set of variables $\text{dom}(\theta) \setminus \text{vars}(\text{shared}(\theta))$ are mapped to ground terms by θ .
3. Steps three and four of *abstract_unify* correctly derive the set of definitely ground variables.
4. Conclude from the above steps [1-3] that *propagate_frs* is called with a correct refinement of the abstract substitution and the mgu.
5. The proof of correctness of the function *propagate_frs* is (i) by induction on the number of equations in the mgu of the goal and the head and (ii) by induction on the structure of terms. We discussed the conservative derivation of potential sharing among and the repeatedness of variables based on the structure of terms (Figure 1) and the repeated occurrences of their variables.