Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation

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

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

Abstract

In this paper, abstract interpretation algorithms are described for computing the sharing as well as the freeness information about the run-time instantiations of program variables. An abstract domain is proposed which accurately and concisely represents combined freeness and sharing information for program variables. Abstract unification and all other domain-specific functions for an abstract interpreter working on this domain are presented. These functions are illustrated with an example. The importance of inferring freeness is stressed by showing (1) the central role it plays in non-strict goal independence, and (2) the improved accuracy it brings to the analysis of sharing information when both are computed together. Conversely, it is shown that keeping accurate track of sharing allows more precise inference of freeness, thus resulting in an overall much more powerful abstract interpreter.

1 Introduction

The technique of abstract interpretation [7] has been studied in the context of flow analysis of logic programs giving rise to a number of frameworks and applications ([2], [13], [8], [17], [18], [3], [5], [12], [6], [11] ...). A shortcoming of many previously proposed approaches (specially when targeted at the optimization of parallel execution) has been the lack of accurate inference of program variable sharing and freeness information. In an earlier paper [14], algorithms were proposed for performing abstract unification which, when combined (in the spirit of Bruynooghe’s framework) with the top-down driven abstract interpretation algorithm presented in [16], can be used to obtain accurate variable sharing and groundness information for a program and a given query. This combined information is termed simply as sharing in this paper. However, knowledge of the sharing information alone does not allow the determination of the freeness of program variables in the subgoal: i.e. sharing only tells whether two variables can be potentially aliased or whether a
variable is bound to a ground term. However, sharing does not distinguish between
a variable which is just bound to another variable and one which is bound to a
complex term. Such a variable is said to be free in the former case and non-free in
the latter. It turns out that freeness information is very useful for at least two rea-
sons. First, the information itself is vital in the detection of non-strict independence
among goals, a condition which allows efficient parallelization of programs, and also
in the optimization of unification, goal ordering, avoidance of type checking, gen-
eral program transformation, etc. Second, by computing this freeness information
in combination with the sharing it is possible in turn to obtain much more accurate
sharing information. Conversely, keeping accurate track of sharing also allows more
precise inference of freeness. The overall effect is thus a more precise analysis than
if two separate analyses were performed. These two points are further illustrated
in the following two subsections (1.1,1.2) and in the descriptions of the algorithms.
The rest of the paper proceeds as follows: section 2 reviews some basic concepts in
abstract interpretation. Section 3 then presents our abstraction framework. Section
4 presents the abstract unification algorithm for this framework. Section 5 illus-
trates the abstract unification algorithm through an example. Section 6 explains the
synergistic interaction between sharing and freeness and, finally, section 7 presents
our conclusions.

1.1 Interaction between sharing and freeness
Consider the following clause used in a matrix multiplication program:

\[ \text{multiply([VO|Rest], V1, [Result|Others]):-} \]
\[ \text{vmul(VO,V1,Result), multiply(Rest, V1, Others).} \]

In a typical use of the multiply/3 predicate, multiply/3 takes a matrix (first
argument) and a vector (second argument) and places the product of these two in
the third argument i.e., when this clause is called, the first and the second arguments
of multiply/3 are bound to ground terms and its third argument is a free variable.
Using this freeness information about the third argument it is possible to infer
that the variables Result and Others are free and independent (i.e. they do not share),
when this clause is called. This makes it possible, for example, to simplify
the code generated for the unification of this third argument, and also to conclude
that the atoms \text{vmul(VO,V1,Result)} and \text{multiply(Rest, V1, Others)} will be
independent goals - i.e., executable in parallel in an Independent And-Parallel (IAP)
system without an independence check. It is important to note that this could not
be done without the freeness information: if the third argument were not known to
be free, the variables Result and Others could be potentially aliased to each other
and, therefore, the two subgoals in the body of multiply/3 could be executed in
parallel only after an indepence check. Other ways in which sharing and freeness
interact will be clear in the descriptions of the abstract unification functions.

1.2 Freeness and non-strict Independence
The idea behind independent and-parallelism is to execute in parallel goals which
are independent, in the sense that they cannot affect each other's search space.
In traditional (strict) independent and-parallelism only goals which do not share
variables are executed in parallel. Two run-time goals \( g_1 \) and \( g_2 \) are thus defined
to be strictly independent if \( \text{vars}(g_1) \cap \text{vars}(g_2) = \emptyset \). It turns out that sharing
information is sufficient to infer this property in many cases. However, as pointed
out before, inference of variable freeness can improve the accuracy of the sharing
and groundness information. But, most importantly, freeness information is vital
in the detection of non-strict goal independence [9, 19], a concept which extends
the applicability of independent and-parallelism to a much larger set of goals (and
thereby achieve increased parallelism) by allowing them to share variables, provided
they don’t “compete” for the bindings of such variables. The condition of “no competition” for bindings translates into a series of requirements that some variables be independent (which can be determined as before) and others be free (“nv-bound”) before and after the execution of the parallel goals. It is in order to ensure that this latter condition holds that the freeness analysis is required. Compile-time analysis is especially important in non-strict independence because some of the information required cannot easily be obtained at run-time.

2 Abstract Interpretation of Logic Programs

Abstract interpretation is an elegant and useful technique for performing a global analysis of a program in order to compute, at compile-time, characteristics of the terms to which the variables in that program will be bound at run-time for a given class of queries. In principle, such an analysis could be done by an interpretation of the program which computed the set of all possible substitutions (collecting semantics) at each step. However, these sets of substitutions can in general be infinite and thus such an approach can lead to non-terminating computations. Abstract interpretation offers an alternative in which the program is interpreted using abstract substitutions instead of actual substitutions. An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain. The set of all possible abstract substitutions for a clause represents an “abstract domain” (for that clause) which is usually a complete lattice or cpo of finite height – such finiteness required, in principle, for termination of fixpoint computation. The ordering relation for this partial order is herein represented by “≤.” 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, $u(x) \leq \gamma(u'(\alpha(x)))$.

The input to the abstract interpreter is a set of clauses (the program) and set of “query forms” i.e., names of predicates which can appear in user queries and their abstract substitutions. The goal of the abstract interpreter is then to compute the set of abstract substitutions which can occur at all points of all the clauses that would be used while answering all possible queries which are concretizations of the given query forms. It is convenient to give different names to abstract substitutions depending on the point in a clause to which they correspond. Consider, for example, the clause $h :- p_1, \ldots, p_n$. Let $\lambda_i$ and $\lambda_{i+1}$ be the abstract substitutions to the left and right of the subgoal $p_i$, $1 \leq i \leq n$ in this clause. See figure 1(b).

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

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one being to essentially follow a top-down strategy starting from the query forms.\footnote{More precisely, this strategy can be seen as a top-down driven bottom up computation.} A purely bottom-up analysis scheme is also possible ([8], [1], [12], [4]). The following description is based on the top-down framework of Bruynooghe [2].

In a similar way to the concrete top-down execution, the abstract interpretation process can 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 \( \lambda_{\text{call}} \) 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 \( \beta_1^{\text{entry}}, \ldots, \beta_m^{\text{entry}} \) for these clauses. Compute their exit substitutions \( \beta_1^{\text{exit}}, \ldots, \beta_m^{\text{exit}} \) as explained below. Compute the success substitutions \( \lambda_1^{\text{success}}, \ldots, \lambda_m^{\text{success}} \) corresponding to these clauses. The success substitution \( \lambda^{\text{success}} \) is then the least upper bound (LUB) of \( \lambda_1^{\text{success}}, \ldots, \lambda_m^{\text{success}} \). 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 \( h \vdash p_1, \ldots, p_n \) whose body is non-empty and an entry substitution \( \lambda_1 \), \( \lambda_1 \) is the call substitution for \( p_1 \). Its success substitution \( \lambda_2 \) is computed as above. Similarly, \( \lambda_3, \ldots, \lambda_{n+1} \) are computed. Finally, \( \lambda_{n+1} \) is obtained, which is the exit substitution for this clause. See figure 1(b). For a unit clause (i.e. whose body is empty), its exit substitution is the same as its entry substitution.

Based on this framework, we had described an efficient top-down driven (and abstract domain independent) abstract interpretation algorithm in [16]. In addition to the abstraction and concretization functions, the following abstract domain-specific functions – which together help perform abstract unification – need to be described in order to make the abstract interpreter complete:

- **call_to_entry:** this function computes the entry substitution for a clause \( C \) given a subgoal \( S_g \) (which unifies with the head of \( C \)) and the projection of its call substitution,

- **exit_to_success:** this function computes the projection of the success substitution for a subgoal \( S_g \) given its call substitution and the exit substitution for a clause \( C \) whose head unifies with \( S_g \).

- **lub, project, extend:** these functions respectively compute the LUB of two abstract substitutions, project an abstract substitution on a subgoal and extend the projection of an abstract success substitution on a subgoal to all the variables of the clause in which the subgoal occurs.

In the next section, we introduce an abstract domain which can be used to describe both sharing and freeness. In the following section, we describe the above functions for this abstract domain. Subsequently, we illustrate these functions with the help of an example.
3 Abstraction Framework

The representation of abstract substitutions used herein is described in this section. In the framework proposed abstract substitutions are elements of $D_a = \varphi(Pvar) \times \varphi(Pvar \rightarrow \{G, F, NF\})$. Each abstract substitution is therefore a 2-tuple. Intuitively, the first element holds the sharing information, while the second holds the freeness information. Accordingly, "sharing" and "freeness" are used to designate the corresponding elements of the tuple such that $D_a = D_a_{sharing} \times D_a_{freeness}$. For example, $\lambda_{call}$, the call substitution for a subgoal, is $\lambda_{call} = (\lambda_{sharing\_call}, \lambda_{freeness\_call})$. The two components of $D_a$ are further described in the following sections.

3.1 Sharing Component

The sharing component provides information about potential aliasing and variable sharing among the program variables (as well as groundness). Its structure is the same as in our earlier paper [14] and that of Jacobs and Langen [10]. However, for the sake of keeping this paper self-contained, we give a brief description of the domain for sharing and some definitions and results that are used in section 4.

The sharing component of the abstract substitution for a clause is defined to be a set of sets of program variables in that clause. Informally, a set of program variables appears in the sharing component if the terms to which these variables are bound share a variable.

More formally, a (concrete) substitution for the variables for 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 the functors in the given program and query. We consider only idempotent substitutions.

The function $Occ$ takes two arguments, $\theta$ (a substitution) and $U$ (a variable in $Uvar$) and produces the set of all program variables $X \in Pvar$ such that $U$ occurs in $vars(X\theta)$. The domain of a substitution $\theta$ is written as $dom(\theta)$. The instantiation of a term $t$ under a substitution $\theta$ is denoted as $t\theta$ and $vars(t\theta)$ denotes the set of all variables in $t\theta$.

**Definition 2 (Occ)**

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

The sharing component of the abstraction of a substitution $\theta$ is defined as:

**Definition 3 (Abstraction(sharing) of a substitution)**

$$A_{sharing}(\theta) = \{Occ(\theta, U) | U \in Uvar\}$$

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

**Definition 4 (pos)**

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

Given a subgoal $pred(u_1, \ldots, u_n)$ and the sharing component $\lambda_{\_share}$ of an abstract substitution, the function $P(pred(u_1, \ldots, u_n), \lambda_{\_share})$ computes the dependencies among the argument positions of this subgoal due to $\lambda_{\_share}$. This is expressed as a subset of the powerset of $\{1, \ldots, n\}$

**Definition 5 (P)**

$$P(pred(u_1, \ldots, u_n), \lambda_{\_share}) = \{pos(pred(u_1, \ldots, u_n), S) | S \in \lambda_{\_share}\}$$
Definition 6 (Closure under union) For a set of sets $S_S$, the closure $S_S^*$ of $S_S$ is the smallest superset of $S_S$ that satisfies: $S_1 \in S_S^* \land S_2 \in S_S^* \Rightarrow S_1 \cup S_2 \in S_S^*$.

The following theorem describes an important result which is used in section 4. The reader is referred to [16] for its proof.

Theorem 3.1 Let $\lambda_{\text{share\_call}}$ and $\lambda_{\text{share\_success}}$ be respectively the sharing components of the abstract call and success substitutions of a subgoal $Sg$. Let $\beta_{\text{share\_entry}}$ be the sharing component of the entry substitution of a clause $C$ due to the unification of its head with $Sg$. Then the following statements are true:

- $\lambda_{\text{share\_success}} \subseteq \lambda_{\text{share\_call}}^*$
- $\mathcal{P}(\text{head}(C), \beta_{\text{share\_entry}}) \subseteq (\mathcal{P}(Sg, \lambda_{\text{share\_call}}))^*$

3.2 Freeness Component

The freeness component of an abstract substitution for a clause gives the mapping from its program variables to an abstract domain $\{G, F, NF\}$ of freeness values i.e. $D_{\text{freeness}} = \chi(\text{Var} \rightarrow \{G, F, NF\})$. $X/G$ means that $X$ is bound to only ground terms at run-time. $X/F$ means that $X$ is free, i.e., it is not bound to a term containing a functor. $X/NF$ means that $X$ is potentially non-free, i.e., it can be bound to terms which have functors. During the process of performing abstract unification, we use a set of temporary freeness values of the form $NF(e)$ (where $e$ is a normalized unification equation). After abstract unification is performed, these values are changed to $NF$. $X/NF(e)$ means that $X$ was free prior to unification by the equation $e \equiv X = f(t_1, \ldots, t_n)$ but became non-free due to the equation $e$.

The important consequence of this is that it does not introduce any new sharing between the variables in $\text{vars}(f(t_1, \ldots, t_n))$ nor does it change their freeness values. Suppose, subsequently, that equation $e' \equiv X = \text{Term}$ (where $e \neq e'$) is processed. Now, the freeness values of $X$ and all variables in $\text{vars}(f(t_1, \ldots, t_n))$ and $\text{Term}$ are changed from $NF(e)$ to $NF$. The three freeness values are related to each other by the following partial order: $\perp \subseteq F \subseteq NF$, $\perp \subseteq G \subseteq NF$.

More formally, the freeness value of a term is defined as follows:

Definition 7 (Abstraction(freeness) of a Term)

\[
\mathcal{A}_{\text{freeness}}(\text{Term}) =
\begin{cases}
  \text{G} & \text{if } \text{vars}(\text{Term}) = \emptyset \\
  \text{F} & \text{if } \text{vars}(\text{Term}) = \{Y\} \land \text{Term} \equiv Y \\
  \text{NF} & \text{else}
\end{cases}
\]

3.3 Integration of the Sharing and Freeness Components

In some sense, one can think of Sharing and Freeness as orthogonal components of an abstract substitution in that the former gives the aliasing information while the latter provides the typing information and so one may be tempted to think that they do not interact with each other. On the contrary, and as mentioned before, there is a symmetric interaction between the two components in the abstract unification algorithms, the presence of the sharing component increasing the precision of the information in the freeness component derived by the analysis and vice versa. This is further illustrated in section 6. Moreover, the two components $\lambda_{\text{sharing}}$ and $\lambda_{\text{freeness}}$ of an abstract substitution $\lambda$ are related to each other by the following condition: $X \notin \text{vars}(\lambda_{\text{sharing}}) \Leftrightarrow X/G \in \lambda_{\text{freeness}}$

Definition 8 (Abstraction of a set of substitutions)

$\alpha(\Theta) = (\cup_{\theta \in \Theta} \mathcal{A}_{\text{sharing}}(\theta), \{X/F \mid F \sup_{X/\text{Term} \in \Theta, \theta \in \Theta} \mathcal{A}_{\text{freeness}}(\text{Term})\})$
Definition 9 (Concretization of an abstract substitution)
\[
\gamma(A_{\text{subst}}) = \{ \theta \mid \theta \text{ is a substitution}, A_{\text{sharing}}(\theta) \subseteq A_{\text{sharing}} \}
\]
where \( A_{\text{subst}} = (A_{\text{sharing}}, A_{\text{freeness}}) \)

The set inclusion relation in the concrete domain induces a partial order on the abstract substitutions, i.e., \( \lambda_1 \subseteq \lambda_2 \) iff \( \gamma(\lambda_1) \subseteq \gamma(\lambda_2) \). It can be easily shown that \( \lambda_1 \subseteq \lambda_2 \) iff the following conditions are satisfied: (1) \( \lambda_1_{\text{sharing}} \subseteq \lambda_2_{\text{sharing}} \) and (2) \( X/Fs1 \in \lambda_1_{\text{freeness}} \land X/Fs2 \in \lambda_2_{\text{freeness}} \Rightarrow Fs1 \subseteq Fs2 \).

The function \( \text{lub} \) computes the least upper bound of two abstract substitutions \( A_{\text{subst}1} \) and \( A_{\text{subst}2} \) by taking the least upper bound of their \( \text{sharing} \) and \( \text{freeness} \) components.

Definition 10 (\( \text{lub} \))
\[
\text{lub}(A_{\text{subst}1}, A_{\text{subst}2}) = (A_{\text{subst}1_{\text{share}}} \cup A_{\text{subst}2_{\text{share}}},
\text{lub}_{\text{freeness}}(A_{\text{subst}1_{\text{freeness}}}, A_{\text{subst}2_{\text{freeness}}}))
\]
where \( (A_{\text{subst}1_{\text{share}}}, A_{\text{subst}1_{\text{freeness}}}) = A_{\text{subst}1} \)
and \( (A_{\text{subst}2_{\text{share}}}, A_{\text{subst}2_{\text{freeness}}}) = A_{\text{subst}2} \)

The function \( \text{lub}_{\text{freeness}} \) computes the least upper bound of the \( \text{freeness} \) components of two abstract substitutions \( A_{\text{subst}1} \) and \( A_{\text{subjst}2} \).

Definition 11 (\( \text{lub}_{\text{freeness}} \))
\[
\text{lub}_{\text{freeness}}(A_{1_{\text{freeness}}, A_{2_{\text{freeness}}}}) =
\{ X/Fs \mid X/Fs1 \in A_{1_{\text{freeness}}, X/Fs2 \in A_{2_{\text{freeness}}, Fs} \leftarrow \text{if} \ (Fs1 = Fs2) \text{ then } Fs1 \text{ else } NF \}
\]

4 Algorithms for Computing Abstract Entry Substitution and Abstract Success Substitution

In this section, we present algorithms for computing the abstract entry substitution \( \text{(callLto_entry)} \) and the projection of the abstract success substitution of a subgoal \( \text{(exitLto_success)} \). The notation for the variables used in these algorithms is described in figure 2. We also describe functions for some basic operations that deal with our abstract domain like \text{project} \ and \text{extend}. Unless otherwise noted, all substitutions referred to in the rest of this paper are abstract substitutions.

The top-level function, \( \text{callLto_entry} \), takes as its input the arguments \( \lambda \) (the projection of the call substitution on the subgoal), \( Sg \) (the subgoal), and \( C \) (the clause whose head has the same functor as \( Sg \) and whose entry substitution is to be computed)\(^4\) and returns \( \beta_{\text{entry}} \) (the entry substitution for the clause \( C \)). The following gives an intuitive description of the basic steps in this function:

1. First, the unification equation \( Sg = \text{head}(C) \) is simplified into a set of irreducible equations by the function \( \text{simplify.equations} \).

2. Starting with the given freeness values of the variables in \( Sg \) and the freeness values of all the variables in \( C \) being \( F \), we perform abstract unification using the function \( \text{abs.unify} \). \( \text{abs.unify} \) performs two important functions: (1) propagate groundness, (2) and propagate freeness. This function, along with the function \( \text{partition} \) forms the core of our algorithm.

\(^4\)It is assumed that \( Sg \) and \( \text{head}(C) \) are unifiable, otherwise the values of both the entry substitution and the success substitution are \( \bot \). Also, it is assumed that the variables in \( C \) are renamed so that \( \text{vars}(Sg) \cap \text{vars}(C) = \emptyset \).
\( \lambda_{\text{call}}, \lambda_{\text{success}} \) - Call and Success substitutions for the subgoal \( S_g \)

\( \lambda, \lambda' \) - Projections of the Call and Success substitutions on the subgoal \( S_g \)

\( \beta_{\text{entry}}, \beta_{\text{exit}} \) - Entry and Exit substitutions for the Clause \( C \) when its head is unified with subgoal \( S_g \)

\( X, X' \) - program variables in \( S_g \) or \( C \)

\( V \) - \( \{X/F | X \in \text{vars}(S_g) \text{ or } X \in \text{vars}(C)\} \)

\( F_s, F_s', F_s'' \) - variables from the domain \( \{G, F, NF, NF(e)\} \)

\( E, E' \) - sets of unification equations

\( e, e' \) - unification equations

\( S_g_{\text{share}} \) - updated sharing information about the variables in \( S_g \) after unification

\( S, S', P, P_1, P_2 \) - sets of program variables

\( SS \) - set of sets of program variables

\( \_ \) - "don't care value" for a variable

Figure 2: Notation for the variables

3. Since some program variables might have become ground due to abstract unification, \( S_g_{\text{share}} \), the updated sharing information for variables in \( S_g \), is computed using the function \( \text{update}\_\text{sharing} \).

4. Using the sharing information in \( E \), the set of simplified equations obtained by abstract unification and in \( S_g_{\text{share}} \), the (updated) sharing information in \( \lambda_{\text{call}} \), a conservative estimate of the sharing information in \( \beta_{\text{entry}} \) is computed by the functions \( \text{powerset}\_\text{of}\_\text{set}\_\text{of}\_\text{sets} \) and \( \text{partition} \).

5. Finally, \( \beta_{\text{entry}} \) is computed by computing its components \( \beta_{\text{share}}_{\text{entry}} \) (which is obtained by pruning \( \beta_{\text{share}} \) so that it agrees with the sharing information in \( \lambda_{\text{call}} \)) and \( \beta_{\text{freeness}}_{\text{entry}} \) (using the function \( \text{project}\_\text{freeness} \)).

Definition 12 (\textbf{call}\_to\_\textbf{entry})

\[
\text{call}\_\text{to}\_\text{entry}(\lambda, S_g, C) = (\beta_{\text{share}}_{\text{entry}}, \beta_{\text{freeness}}_{\text{entry}})
\]

where

\[
\beta_{\text{share}}_{\text{entry}} = \{\{X\} | X \in \text{body}(C), X \notin \text{head}(C)\}
\]

\[
\cup \{S | S \in \beta_{\text{share}}, \text{pos}(\text{head}(C), S) \in (P(S_g, S_g_{\text{share}}))\}
\]

and

\[
\beta_{\text{freeness}}_{\text{entry}} = \text{project}\_\text{freeness}(C, \text{collapse}\_\text{non}\_\text{free}(V))
\]

and

\[
\beta_{\text{share}} = \text{powerset}\_\text{of}\_\text{set}\_\text{of}\_\text{sets}(\text{project}\_\text{share}(\text{head}(C)),
\]

\[
\text{partition}(V, E, S_g_{\text{share}}))
\]

and

\[
S_g_{\text{share}} = \text{update}\_\text{sharing}(V, \lambda_{\text{share}})
\]

and

\[
(V, E) = \text{abs}\_\text{unify}(\lambda_{\text{freeness}}, \cup \{X/F | X \in \text{vars}(C)\},
\]

\[
\text{simplify}\_\text{equations}(\{S_g = \text{head}(C)\})
\]

and

\[
(\lambda_{\text{share}}, \lambda_{\text{freeness}}) = \lambda
\]

The other top-level function \( \text{exit}\_\text{to}\_\text{success} \), is quite similar to \( \text{call}\_\text{to}\_\text{entry} \) in the sense that it also performs abstract unification\(^5\). It takes as its input arguments \( \beta_{\text{exit}} \) (the exit substitution of the clause \( C \)), subgoal \( S_g \), clause \( C \), and \( \lambda \) and computes \( X' \), the projection of the success substitution on \( S_g \). The following are the salient differences in the basic steps between this function and the function \( \text{call}\_\text{to}\_\text{entry} \).

\(^5\)An implementation of these functions would take advantage of the fact that the abstract unification which is performed for \( \text{call}\_\text{to}\_\text{entry} \) is almost the same as the one for \( \text{exit}\_\text{to}\_\text{success} \). Hence it would save the result of abstract unification performed for \( \text{call}\_\text{to}\_\text{entry} \) and reuse it when computing \( \text{exit}\_\text{to}\_\text{success} \).
• First, \( \beta_{\text{exit}} \) is projected on \( \text{head}(C) \) using the function \( \text{project}(\text{head}(C), \beta_{\text{exit}}) \).

• The function \( \text{abs_unify} \) makes use of the freeness values from \( \beta_{\text{exit}} \) in addition to \( \lambda \).

• The function \( \text{partition} \) makes use of the sharing information in \( S_g \text{share} \) as well as \( \beta \text{share}' \).

• Finally, \( \lambda \text{share}' \) is computed by pruning \( \text{Sup}_\lambda \text{share}' \) so that it agrees with the sharing information in both \( \lambda_{\text{call}} \) and \( \beta_{\text{exit}} \).

**Definition 13 (exit_to_success)**

\[
\text{exit_to_success}(\beta_{\text{exit}}, S_g, C, \lambda) = (\lambda \text{share}', \lambda \text{freeness}')
\]

where

\[\lambda \text{share}' = \{ S | S \in (\text{Sup}_\lambda \text{share}' \lor \lambda \text{share}) \}\]

and \( \lambda \text{freeness}' = \text{project_freeness}(S_g, \text{collapse_non_free}(V)) \)

and \( \text{Sup}_\lambda \text{share}' = \text{powerset_of_set_of_sets}(\text{project_share}(S_g, \text{partition}(V, E, S_g \text{share} \lor \beta \text{share}'))) \)

and \( S_g \text{share} = \text{update_sharing}(V, \lambda \text{share}) \)

and \( (V, E) = \text{abs_unify}(\lambda \text{freeness} \lor \beta \text{freeness}', \text{simplify_equations}((\{ S_g = \text{head}(C) \}))) \)

and \( \beta' = (\beta \text{share}', \beta \text{freeness}') = \text{project}(\text{head}(C), \beta_{\text{exit}}) \)

and \( (\lambda \text{share}, \lambda \text{freeness}) = \lambda \)

The function \( \text{simplify_equations} \) takes as its input \( E \), the set of unification equations and recursively simplifies them until all equations are of the form \( X = \text{Term} \).

**Definition 14 (simplify_equations)**

\[
\text{simplify_equations}(E) =
\begin{cases}
  \emptyset & \text{if } \exists e \in E. e \equiv f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n) \\
  \text{simplify_equations}(E \cup \{ t_1 = u_1, \ldots, t_n = u_n \} - \{ e \}) & \text{else }
\end{cases}
\]

The function \( \text{abs_unify} \) takes as input \( V \) (set of freeness value assignments for the variables in \( S_g \) and \( \text{head}(C) \)) and \( E \) (the set of normalized unification equations obtained from \( S_g = \text{head}(C) \)) and computes \( (V', E') \) where \( V' \) and \( E' \) are the updated values of \( V \) and \( E \) after abstract unification is performed.

Assume that \( E = \{ e_1, \ldots, e_n \} \), where \( e_i \) is of the form \( X = Y \) or \( X = f(t_1, \ldots, t_m) \). This function performs fixpoint computation on the ordered pair \( (V, E) \). During each iteration, each \( e_i, i = 1, \ldots, n \) is visited using the function \( \text{au_nify} \). After all the equations have been visited, it is checked if any freeness value or equation has changed during the current iteration. If so, the fixpoint computation is continued, otherwise it outputs \( (V, E) \).

**Definition 15 (abs_unify)**

\[
\text{abs_unify}(V, E) =
\begin{cases}
  \text{au_nify}(V, E, \emptyset) & \text{if } \text{au_nify}(V, E, \emptyset) = (V, E) \\
  (V, E) & \text{then } (V, E) \\
  \text{else } \text{abs_unify}(V', E') & \text{where } (V', E') = \text{au_nify}(V, E, \emptyset)
\end{cases}
\]

The function \( \text{au_nify} \) has three input parameters: \( V, E, E' \). \( V \) is the same as in \( \text{abs_unify} \), \( E \) and \( E' \) are sets of normalized unification equations. This function is invoked by the function \( \text{abs_unify} \) with \( E' = \emptyset \) and performs one iteration (of abstract unification) by visiting each of the equations in \( E \).

During each step, an equation \( e_i \in E \) is removed from \( E \). The freeness values in \( V \) are updated using this equation. \( e_i \) is added to \( E' \) if and only if all the variables in this equation have not become ground at this step.
Definition 16 (aunify)
\[
aunify(V, \{X = Term\} \cup E, E') =
\begin{align*}
\text{if } (X/G) \in V \text{ then } & \aunify(V - \{Y/G\}Y \in \text{vars}(Term)) \cup \{Y/G\}Y \in \text{vars}(Term), E, E' \\
\text{if } \text{vars}(Term) = \emptyset \text{ or } & \forall Y \in \text{vars}(Term). (Y/G) \in V \\
\text{then } & \aunify(V - \{X/G\}, E, E') \\
\text{if } Term \equiv Y \text{ and } & (X/F) \in V \text{ and } (Y/F) \in V \\
& \text{then } \aunify(V, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv Y \text{ and } & (X/NF(e)) \in V \text{ and } (Y/NF(e')) \in V \\
& \text{then } \aunify(V - \{Y/NF(e)\} \cup \{Y/NF(e')\}, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv Y \text{ and } & (X/NF(e)) \in V \text{ and } (Y/NF(e)) \in V \\
& \text{then } \aunify(V - \{Y/NF(e)\} \cup \{Y/NF(e')\}, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv Y \text{ and } & (X/NF(e)) \in V \text{ and } (Y/NF(e)) \in V \\
& \text{then } \aunify(V - \{X/NF(e)\} \cup \{X/NF\} \cup \{Y/NF\}, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv Y \text{ and } & (X/NF) \in V \text{ and } (Y/NF) \in V \\
& \text{then } \aunify(V, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv Y \text{ and } & (X/NF) \in V \text{ and } (Y/NF) \in V \\
& \text{then } \aunify(V, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv f(t_1, \ldots, t_n) \text{ and } & (X/F) \in V \\
& \text{then } \aunify(V - \{X/F\} \cup \{X/NF(X = Term)\}, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv f(t_1, \ldots, t_n) \text{ and } & (X/NF(X = f(t_1, \ldots, t_n)) \in V \\
& \text{then } \aunify(V, E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv f(t_1, \ldots, t_n) \text{ and } & (X/NF(e)) \in V \text{ and } e \neq X \equiv Term \\
& \text{then } \aunify(V - \{X/NF(e)\} \cup \{Y/NF(Y \in \text{vars}(Term)) \cup \{Y/NF\} \cup \{Y/NF\}Y \in \text{vars}(Term), E, E' \cup \{X = Term\}) \\
\text{if } Term \equiv f(t_1, \ldots, t_n) \text{ and } & (X/NF) \in V \\
& \text{then } \aunify(V - \{Y/NF\}Y \in \text{vars}(Term), E, E' \cup \{X = Term\}) \\
\end{align*}
\]
\aunify(V, \emptyset, E) = (V, E)

Some program variables may become ground after abstract unification is performed. The function update_sharing takes as input \(V\) (freeness values of variables after abstract unification) and \(\lambda\share\) and computes \(S_\lambda\share\) (the updated sharing information for variables in \(S_\lambda\)) as per the information in \(V\). Consider a set \(S \in \lambda\share\). \(S\) is added to \(S_\lambda\share\), if and only if \(S\) does not have a variable which is ground according to \(V\).

Definition 17 (update_sharing)
\[
\text{update_sharing}(V, SS) = \{S \mid S \in SS, \forall X \in S. X/G \notin V\}
\]

Along with \aunify, the function partition forms the “core” of our abstract unification algorithm. Its three input parameters are \(V, E\) and \(\lambda\share\). \(V\) and \(E\) are as before, while \(\lambda\share\) gives the sharing information among the variables in \(S_\lambda\) and/or \(\text{head}(C)\). Making use of these three input values, this function computes the partitions of the “connection graph” for the variables in \(S_\lambda\) and \(\text{head}(C)\).

First, we consider the cases when \(E \neq \emptyset\) and therefore an equation \(X \equiv \text{Term}\) is in \(E\).

\[\text{• If } (X/NF(X = \text{Term})) \in \, V, \text{ then this equation introduces sharing only}\] between

\[\text{Most of the other algorithms, including the algorithm that we had published earlier [14], introduce a sharing between } Y \text{ and } Y' \text{ and thus lose precision. In this case, we are able to avoid this pitfall only because we have the additional freeces information.}\]
\(X\) and an \(Y \in \text{vars}(\text{Term})\) and not between \(Y\) and \(Y'\) where \(Y, Y' \in \text{vars}(\text{Term})\).
- If \((X/NF) \in V\), then sharing is introduced not only between \(X\) and \(Y\) but also between \(Y\) and \(Y'\) for all \(Y, Y' \in \text{vars}(\text{Term})\).
- If \((X/F) \in V\), then \(\text{Term} \equiv Y\) and therefore a sharing is introduced between \(X\) and \(Y\).

We next consider the case when \(E = \emptyset\) i.e. all unification equations have been processed. If an \((X/Fs) \in V\) is such that \(Fs = G\) (i.e. \(X\) is ground) or \(X\) is in \(\text{vars}(\text{Share})\) (i.e. \(X\)'s partition already exists), then nothing is done. Otherwise, a new partition containing only \(X\) is added.

**Definition 18 (partition)**

\[
\text{partition}(V, \{X = \text{Term}\} \cup E, \text{Share}) = \\
\begin{cases} 
(X/NF(X = \text{Term})) \in V \\
\text{then partition}(V, E, \text{Share}) - \{P | P \in \text{partition}(V, E, \text{Share}), \\ 
X \in P \text{ or } \text{vars}(\text{Term}) \cap P \neq \emptyset \} \cup \{P_1 \cup P_2 | P_1, P_2 \in \text{partition}(V, E, \text{Share}), X \in P_1 \text{ or } \text{vars}(\text{Term}) \cap P_2 \neq \emptyset \} \\
\end{cases}
\]

\[\text{partition}(V, \emptyset, \text{Share}) = \{\{X\} | X/Fs \in V, Fs \neq G, X \notin \text{vars}(\text{Share})\} \cup \text{Share}\]

**Definition 19 (powerset_of_set_of_sets)**

\[\text{powerset_of_set_of_sets}(SS) = \bigcup \forall \in SS \nu(\forall)\]

The function \(\text{collapse_non_free}\) is needed because we use more than three values for freeness viz, \(G, F, NF, NF(e)\) while performing abstract unification, but subsequently, we use only three freeness values \(G, F, NF\) for the variables. Essentially, this function converts all \(NF(e)\) to \(NF\).

**Definition 20 (collapse_non_free)**

\[\text{collapse_non_free}(V) = \{X/Fs | X/Fs' \in V, Fs' \text{ if } (Fs' = NF(e)) \text{ then } NF \text{ else } Fs'\}\]

The inputs to the function \(\text{project}\) are \(\text{Term}\) (which could be an atom or a clause) and \(\text{Asubst}\) (abstract substitution). The output of this function is the projection of \(\text{Asubst}\) on \(\text{Term}\).

**Definition 21 (project)**

\[\text{project}(\text{Term}, (\text{Asubst}_{\text{share}}, \text{Asubst}_{\text{freeness}})) = \]
\[\text{project}_{\text{share}}(\text{Term}, \text{Asubst}_{\text{share}}), \]
\[\text{project}_{\text{freeness}}(\text{Term}, \text{Asubst}_{\text{freeness}})\]

The function \(\text{project}_{\text{share}}\), projects \(\text{Asubst}_{\text{share}}\) (the \(\text{freeness}\) component of \(\text{Asubst}\)) on \(\text{Term}\).

**Definition 22 (project_{share})**

\[\text{project}_{\text{share}}(\text{Term}, \text{Asubst}_{\text{share}}) = \]
\[\{S | S = (S' \cap \text{vars}(\text{Term})), S' \in \text{Asubst}_{\text{share}}\}\]
The function \texttt{project-freeness} projects \texttt{Asubst\textsubscript{freeness}} (the \textit{freeness} component of \texttt{Asubst}) on \texttt{Term}.

\begin{definition}[\texttt{project\_freeness}]
\texttt{project\_freeness}(\texttt{Term}, \texttt{Asubst\textsubscript{freeness}}) =\{X/Fs \mid X/Fs \in \texttt{Asubst\textsubscript{freeness}}, X \in \texttt{vars(Term)}\}.
\end{definition}

Given the inputs \texttt{Sg} (the subgoal), \texttt{\lambda\textsubscript{call}} (the call substitution for \texttt{sg}), and \texttt{\lambda'} (the projection of the success substitution on \texttt{Sg}), the function \texttt{extend} computes \texttt{\lambda\textsubscript{success}} (the success substitution for \texttt{Sg}).

The \textit{freeness} component of \texttt{\lambda\textsubscript{success}} is computed by taking in the freeness values of the variables in \texttt{Sg} from \texttt{\lambda'}. The freeness values of the other variables in the clause of \texttt{Sg} (which have not become \textit{ground} due to the execution of \texttt{Sg}) are computed as follows: If either the freeness of \texttt{X} is \texttt{NF} in \texttt{\lambda\textsubscript{call}} or \texttt{X} and another variable \texttt{Y} which occurs in \texttt{Sg} are potentially aliased (according to \texttt{\lambda\textsubscript{share}} \texttt{\textsubscript{success}}) and the freeness of \texttt{Y} is \texttt{NF}, then the freeness of \texttt{X} is \texttt{NF}, otherwise it is \texttt{F}.

The \textit{sharing} component of \texttt{\lambda\textsubscript{success}} is computed as follows: Consider the sets in the \textit{sharing} component of \texttt{\lambda\textsubscript{call}} whose variables do not appear in \texttt{Sg}. These are not obviously affected by the execution of \texttt{Sg} and hence are added to the \textit{sharing} component of \texttt{\lambda\textsubscript{success}}. The remaining sets have variables that do appear in \texttt{Sg} and hence we consider the closure of these sets under unión and add those sets whose projections appear in the \textit{sharing} component of \texttt{\lambda'}.

\begin{definition}[\texttt{extend}]
\texttt{extend}(\texttt{Sg}, \texttt{\lambda\textsubscript{call}}, \texttt{\lambda'}) = (\texttt{\lambda\textsubscript{share}} \texttt{\textsubscript{success}}, \texttt{\lambda\textsubscript{freeness}} \texttt{\textsubscript{success}})

where
\[\texttt{\lambda\textsubscript{freeness}} \texttt{\textsubscript{success}} = \texttt{\lambda\textsubscript{freeness'}} \cup \{X/G \mid X \in \texttt{vars(\lambda\textsubscript{share} \textsubscript{call})} - \texttt{vars(\lambda\textsubscript{share} \textsubscript{success})}\} \cup \{X/F \mid X \in \texttt{vars(\lambda\textsubscript{share} \textsubscript{success})} - \texttt{vars(\lambda\textsubscript{freeness'})}, Fs \leftarrow \texttt{if (X/NF \in \lambda\textsubscript{freeness} \textsubscript{call}) \texttt{\lor (3Y \in S, X \in \lambda\textsubscript{share} \textsubscript{call}, Y/NF \in \lambda\textsubscript{freeness'}) then NF else F}\}\}

and
\[\texttt{\lambda\textsubscript{share}} \texttt{\textsubscript{success}} = \{S \mid S \in \{S' \mid S' \in \texttt{\lambda\textsubscript{share} \textsubscript{call}}, S' \cap \texttt{vars(Sg)} \neq \emptyset\} \cup \{S \mid S \in \texttt{\lambda\textsubscript{share} \textsubscript{call}}, S \cap \texttt{vars(Sg)} = \emptyset\}\}

and
\[\texttt{(\lambda\textsubscript{share} \textsubscript{call}, \lambda\textsubscript{freeness} \textsubscript{call})} = \texttt{\lambda\textsubscript{call}}\]

and
\[\texttt{(\lambda\textsubscript{share'}, \lambda\textsubscript{freeness'})} = \texttt{\lambda'}\]

\end{definition}

\begin{proposition}
Given a subgoal \texttt{Sg} whose abstract call substitution is \texttt{\lambda\textsubscript{call}} and a clause \texttt{C} whose head unifies with \texttt{Sg}, let \texttt{\beta\textsubscript{entry}} be the abstract entry substitution for \texttt{C} as computed by the function \texttt{call\_to\_entry}. Then, \texttt{\beta\textsubscript{entry}} is a safe approximation in the following sense: In the concrete interpretation, let \texttt{\Omega\textsubscript{entry}} be the set of entry substitutions for clause \texttt{C} computed from \texttt{Sg}'s set of call substitutions \texttt{\gamma(\lambda\textsubscript{call})}. Then, \texttt{\Omega\textsubscript{entry}} \subseteq \texttt{\gamma(\beta\textsubscript{entry})}.
\end{proposition}

The reader is referred to [15] for a proof of this proposition, which is omitted here for lack of space.

\section{Example}

We illustrate the algorithm \texttt{call\_to\_entry} in section 4 with the aid of an example\footnote{A similar proposition about the safety of the abstract substitution \texttt{\lambda'} computed by the function \texttt{exit\_to\_success} can also be stated and proved. However, due to lack of space, we do not present it here.}. This example is rather contrived and its main function is to illustrate the mechanics of the algorithm as it deals with different cases.

\footnote{The function \texttt{exit\_to\_success} is similar to this function and therefore not illustrated.}
Focussing on the freeness variables of $X_1$, $X_2$, $X_3$, $X_4$, let the value of the problem is to compute the success substitution from its projection. Following the metric interaction and synergy that exists between these two components of the abstract substitution.

Consider a subgoal $Sg \equiv \text{pred}(X_1, X_2)$ whose call substitution is $\{[\emptyset, \{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}], \{X_1/F, X_2/F, X_3/F, X_4/F\}\}$. The value of the projection of its success substitution be $\{[\emptyset, \{X_1\}, \{X_2\}], \{X_1/NF, X_2/NF\}\}$. The problem is to compute the success substitution from its projection. Following the algorithm in section 4, we get the value of the success substitution to be $\{[\emptyset, \{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}], \{X_1/NF, X_2/NF, X_3/F, X_4/NF\}\}$. Focussing on the freeness values of $X_3$ and $X_4$, we notice that the former has the same freeness value of $F$.
before and after the execution of $S_g$, while the latter has changed from $F$ to $NF$. Why this difference in spite of the fact that both of them do not occur in $S_g$? This can be explained by the fact that $X_3$ is not aliased to any other variable in $S_g$'s call substitution while $X_4$ is potentially aliased to $X_2$. Therefore, $X_3$ is not affected by the execution of $S_g$ while $X_4$ is. It can potentially become non-free since the freeness value of $X_2$ has changed from $F$ to $NF$.

Consider an analysis wherein we have the freeness information but not the sharing information. Assume the same value for the freeness component of the projected success substitution: the freeness values of both $X_1$ and $X_2$ have changed from $F$ to $NF$. In this case, we do not know the sharing information among the four variables and hence we have to do the analysis assuming the worst case i.e. all four variables could be aliased to each other. Therefore, the freeness values of both $X_3$ and $X_4$ would be changed from $F$ to $NF$.

Thus we see that, in the absence of sharing information, we can only infer that the freeness value of $X_3$ is $NF$ rather than the more accurate value of $F$ in $S_g$'s success substitution that can be obtained by using the sharing information. Clearly, the presence of sharing information enhances the accuracy of the freeness information achievable by analysis.

### 7 Conclusions

An abstraction framework and abstract unification algorithms for combined inference at compile-time of groundness, sharing, and freeness information have been presented. The algorithms presented can be combined with a variety of abstract interpretation frameworks to provide analyses useful in the detection of non-strict independence among goals, a condition which ensures efficient parallelization of programs, and in the optimization of unification, goal ordering, avoidance of type checking, general program transformation, etc. It has been shown how such analyses gain in power from the increased precision arising from the combined inference of sharing and freeness information proposed in this paper. It will be interesting to implement this framework and study the tradeoffs between the cost of carrying around the extra information (freeness) and the increased precision it brings to the analysis.

### References


general algorithm and its correctness. Research report, Department of Pure

Static Analysis of Programs by Construction or Approximation of Fixpoints.
In Acm Symp. on Principles of Programming Languages, pages 238–252, 1977.


Press, June 1990.

Aliasing in Logic Programs. In 1989 North American Conference on Logic

Abstract Interpretation: an Experiment in Code Generation. In Sixth Interna-


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

Information at Compile-Time Through Abstract Interpretation. In 1989 North
American Conference on Logic Programming, pages 166–189. MIT Press, Oc-
tober 1989.

and Freeness of Program Variables Through Abstract Interpretation. Technical
Report STP-368-90, Microelectronics and Computer Technology Corporation
(MCC), Austin, TX 78759, December 1990.

Dependency Using Abstract Interpretation. Journal of Logic Programming,


Flow Analysis of Logic Programs. In Fifth International Conference and Sym-
posium on Logic Programming, pages 684–699, Seattle, Washington, August

[19] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence
of shared Free variables. In Fifth International Conference and Symposium on