Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation[1]

K. Muthukumar[2]
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

[1]This research was supported in part by ESPRIT projects 2471 "PEPMA" and 5246 "PRINCE," CICYT project 361.208, and also in part by MCC.

[2]Part of the research reported in this paper was performed when this author was visiting University of Madrid (UPM).

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 reasons. 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, general 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 illustrates 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:

```
multiply([V0|Rest], V1, [Result|Others]):-
       vmul(V0,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 `vmul(V0,V1,Result)` and `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 independence 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 $vars(g_1) \cap vars(g_2) = \emptyset$. 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 "$\sqsubseteq$." Abstract substitutions and sets of concrete substitutions are related via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. In addition, each primitive operation u of the language (unification being a notable example) is abstracted to an operation u' over the abstract domain. Soundness of the analysis requires that each concrete operation u be related to its corresponding abstract operation u' as follows: for every x in the concrete computational domain, $u(x) \sqsubseteq \gamma(u'(\alpha(x)))$.

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 λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. See figure 1(b).

Definition 1 *λ_i and λ_{i+1} are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal p_i. For this same clause, λ_1 is the* abstract entry substitution *(also represented as β_{entry}) and λ_{n+1} is the* abstract exit substitution *(also represented as β_{exit}).*

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one being to essentially follow a top-down strategy starting from the query forms.[3] 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

[3] More precisely, this strategy can be seen as a *top-down driven* bottom up computation.

Figure 1: Illustration of the abstract interpretation process

with clauses whose heads are $h_1, \ldots, h_m$, p is an OR-node whose children are the AND-nodes $h_1, \ldots, h_m$ (figure 1(a)). During construction of the tree, computation of the abstract substitutions at each point is done as follows:

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

• *Computing exit substitution from entry substitution*: Given a clause h :- $p_1, \ldots, p_n$ whose body is non-empty and an entry substitution λ_1, λ_1 is the call substitution for p_1. Its success substitution λ_2 is computed as above. Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. Finally, λ_{n+1} is obtained, which is the exit substitution for this clause. See figure 1(b). 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 Sg (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 Sg given its call substitution and the exit substitution for a clause C whose head unifies with Sg.

- *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 $D_\alpha = \wp(\wp(Pvar)) \times \wp(Pvar \rightarrow \{G, F, NF\})$. 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 $D_\alpha = D_\alpha\_sharing \times D_\alpha\_freeness$. For example, λ_{call}, the call substitution for a subgoal, is $\lambda_{call} = (\lambda\_share_{call}, \lambda\_freeness_{call})$. The two components of D_α 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, θ (a substitution) and U (a variable in *Uvar*) and produces the set of all program variables $X \in Pvar$ such that U occurs in $vars(X\theta)$. The domain of a substitution θ is written as $dom(\theta)$. The instantiation of a term t under a substitution θ 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 θ is defined as:

Definition 3 (Abstraction(sharing) of a substitution)

$$\mathcal{A}_{sharing}(\theta) \quad = \quad \{Occ(\theta, U) \mid U \in Uvar\}$$

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

Definition 5 ($\mathcal{P}$)

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

Definition 6 (Closure under union) *For a set of sets SS, the closure SS^* of SS is the smallest superset of SS that satisfies: $S_1 \in SS^* \wedge S_2 \in SS^* \Rightarrow S_1 \cup S_2 \in SS^*$.*

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 λ_share_{call} and $\lambda\_share_{success}$ be respectively the sharing components of the abstract call and success substitutions of a subgoal Sg. Let β_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:*

- $\lambda\_share_{success} \subseteq \lambda\_share_{call}{}^*$
- $\mathcal{P}(head(C), \beta\_share_{entry}) \subseteq (\mathcal{P}(Sg, \lambda\_share_{call}))^*$

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. $D_\alpha\text{-}freeness = \wp(Pvar \rightarrow \{G, F, NF\})$. 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 $vars(f(t_1, \ldots, t_n))$ nor does it change their *freeness* values. Suppose, subsequently, that equation $e' \equiv X = Term$ (where $e \neq e'$) is processed. Now, the freeness values of X and all variables in $vars(f(t_1, \ldots, t_n))$ and $Term$ are changed from $NF(e)$ to NF. The three freeness values are related to each other by the following partial order: $\perp \sqsubseteq F \sqsubseteq NF$, $\perp \sqsubseteq G \sqsubseteq NF$

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

Definition 7 (Abstraction(freeness) of a Term)

$$\mathcal{A}_{freeness}(Term) = $$
$$= \begin{cases} if \quad vars(Term) = \emptyset \quad then\ G \\ if \quad vars(Term) = \{Y\} \wedge Term \equiv Y \quad then\ F \\ else \quad NF \end{cases}$$

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 $\lambda\_sharing$ and $\lambda\_freeness$ of an abstract substitution λ are related to each other by the following condition: $X \notin vars(\lambda\_sharing) \Leftrightarrow X/G \in \lambda\_freeness$

Definition 8 (Abstraction of a set of substitutions)

$$\alpha(\Theta) = (\cup_{\theta \in \Theta} \mathcal{A}_{sharing}(\theta), \{X/Fs \mid Fs = \sup_{X/Term \in \theta, \theta \in \Theta} \mathcal{A}_{freeness}(Term)\})$$

Definition 9 (Concretization of an abstract substitution)

$$\gamma(Asubst) = \{\theta \mid \theta \text{ is a substitution}, \mathcal{A}_{sharing}(\theta) \subseteq Asubst_{sharing}, \\ \forall X/Term \in \theta.\ \exists X/Fs \in Asubst_{freeness}.\ \mathcal{A}_{freeness}(Term) \sqsubseteq Fs\} \\ \textit{where } Asubst = (Asubst_{sharing}, Asubst_{freeness})$$

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

The function *lub* computes the least upper bound of two abstract substitutions $Asubst1$ and $Asubst2$ by taking the least upper bound of their *sharing* and *freeness* components.

Definition 10 (lub)

$$lub(Asubst1, Asubst2) = (Asubst1_{share} \cup Asubst2_{share}, \\ lub\_freeness(Asubst1_{freeness}, Asubst2_{freeness})) \\ \textit{where} \quad (Asubst1_{share}, Asubst1_{freeness}) = Asubst1 \\ \textit{and} \quad (Asubst2_{share}, Asubst2_{freeness}) = Asubst2$$

The function *lub_freeness* computes the least upper bound of the *freeness* components of two abstract substitutions $Asubst1$ and $Asubst2$.

Definition 11 (lub_freeness)

$$lub\_freeness(A1\_freeness, A2\_freeness) = \\ \{X/Fs \mid X/Fs1 \in A1\_freeness,\ \ X/Fs2 \in A2\_freeness, \\ Fs \leftarrow if\ \ (Fs1 = Fs2)\ \ then\ \ Fs1\ \ else\ \ NF\}$$

4 Algorithms for Computing Abstract Entry Substitution and Abstract Success Substitution

In this section, we present algorithms for computing the abstract entry substitution (*call_to_entry*) and the projection of the abstract success substitution of a subgoal (*exit_to_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 *project* and *extend*. Unless otherwise noted, all substitutions referred to in the rest of this paper are *abstract* substitutions.

The top-level function, *call_to_entry*, takes as its input the arguments λ (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 β_{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 $Sg = head(C)$ is simplified into a set of irreducible equations by the function $simplify\_equations$.

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 abs_unify. abs_unify performs two important functions: (1) propagate groundness, (2) and propagate freeness. This function, along with the function *partition* forms the core of our algorithm.

[4]It is assumed that Sg and $head(C)$ are unifiable, otherwise the values of both the entry substitution and the success substitution are $\perp$. Also, it is assumed that the variables in C are renamed so that $vars(Sg) \cap vars(C) = \emptyset$.

λ_{call},$\lambda_{success}$ - *Call* and *Success* substitutions for the subgoal Sg
λ,λ' - Projections of the *Call* and *Success* substitutions on the subgoal Sg
β_{entry},β_{exit} - *Entry* and *Exit* substitutions for the Clause C when its head is unified with subgoal Sg
X,Y - program variables in Sg or C
V - $\{X/Fs|X \in vars(Sg)\ or\ X \in vars(C)\}$
Fs,Fs1,Fs2 - variables from the domain $\{G, F, NF, NF(e)\}$
E, E' - sets of unification equations
e, e' - unification equations
Sg_share - updated sharing information about the variables in Sg 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, Sg_share, the updated sharing information for variables in Sg, is computed using the function *update_sharing*.

4. Using the sharing information in E, the set of simplified equations obtained by abstract unification and in Sg_share, the (updated) sharing information in λ_{call}, a conservative estimate of the sharing information in β_{entry} is computed by the functions *powerset_of_set_of_sets* and *partition*.

5. Finally, β_{entry} is computed by computing its components β_share_{entry} (which is obtained by pruning β_share so that it agrees with the sharing information in λ_{call}) and $\beta\_freeness_{entry}$ (using the function *project_freeness*).

Definition 12 (call_to_entry)

$$
\begin{array}{ll}
\multicolumn{2}{l}{call\_to\_entry(\lambda, Sg, C) = (\beta\_share_{entry}, \beta\_freeness_{entry})} \\
\quad where & \beta\_share_{entry} = \{\{X\}|X \in body(C), X \notin head(C)\} \\
 & \cup\{S|S \in \beta\_share, pos(head(C), S) \in (\mathcal{P}(Sg, Sg\_share))^*\} \\
\quad and & \beta\_freeness_{entry} = project\_freeness(C, collapse\_non\_free(V)) \\
\quad and & \beta\_share = powerset\_of\_set\_of\_sets(project\_share(head(C), \\
 & \qquad\qquad partition(V, E, Sg\_share))) \\
\quad and & Sg\_share = update\_sharing(V, \lambda\_share) \\
\quad and & (V, E) = abs\_unify(\lambda\_freeness \cup \{X/F|X \in vars(C)\}, \\
 & \qquad\qquad simplify\_equations(\{Sg = head(C)\})) \\
\quad and & (\lambda\_share, \lambda\_freeness) = \lambda
\end{array}
$$

The other top-level function *exit_to_success*, is quite similar to *call_to_entry* in the sense that it also performs abstract unification[5]. It takes as its input arguments β_{exit} (the exit substitution of the clause C), subgoal Sg, clause C, and λ and computes λ', the projection of the success substitution on Sg. The following are the salient differences in the basic steps between this function and the function *call_to_entry*:

[5] An implementation of these functions would take advantage of the fact that the abstract unification which is performed for *call_to_entry* is almost the same as the one for *exit_to_success.*. Hence it would save the result of abstract unification performed for *call_to_entry* and reuse it when computing *exit_to_success*.

- First, β_{exit} is projected on $head(C)$ using the function $project(head(C), \beta_{exit})$.
- The function abs_unify makes use of the freeness values from β_{exit} in addition to λ.
- The function *partition* makes use of the sharing information in Sg_share as well as β_share'.
- Finally, λ_share' is computed by pruning Sup_λ_share' so that it agrees with the sharing information in both λ_{call} and β_{exit}.

.

Definition 13 (exit_to_success)

$$
\begin{array}{ll}
\multicolumn{2}{l}{exit\_to\_success(\beta_{exit}, Sg, C, \lambda) = (\lambda\_share', \lambda\_freeness')} \\
\quad where & \lambda\_share' = \{S | S \in (Sup\_\lambda\_share' \cap \lambda\_share^*), \\
 & \qquad\qquad pos(Sg, S) \in \mathcal{P}(head(C), \beta\_share')\} \\
\quad and & \lambda\_freeness' = project\_freeness(Sg, collapse\_non\_free(V)) \\
\quad and & Sup\_\lambda\_share' = powerset\_of\_set\_of\_sets(project\_share(Sg, \\
 & \qquad\qquad partition(V, E, Sg\_share \cup \beta\_share'))) \\
\quad and & Sg\_share = update\_sharing(V, \lambda\_share) \\
\quad and & (V, E) = abs\_unify(\lambda\_freeness \cup \beta\_freeness', \\
 & \qquad\qquad simplify\_equations(\{Sg = head(C)\})) \\
\quad and & \beta' = (\beta\_share', \beta\_freeness') = project(head(C), \beta_{exit}) \\
\quad and & (\lambda\_share, \lambda\_freeness) = \lambda
\end{array}
$$

The function *simplify_equations* takes as its input E, the set of unification equations and recursively simplifies them until all equations are of the form $X = Term$.

Definition 14 (simplify_equations)

$$
simplify\_equations(E) =
$$
$$
= \begin{cases} if & \exists e \in E.\ e \equiv f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n) \\ & then\ \ simplify\_equations(E \cup \{t_1 = u_1, \ldots, t_n = u_n\} - \{e\}) \\ else & E \end{cases}
$$

The function *abs_unify* takes as input V (set of freeness value assignments for the variables in Sg and $head(C)$) and E (the set of normalized unification equations obtained from $Sg = head(C)$) 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 *aunify*. 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)

$$
abs\_unify(V, E) =
$$
$$
= \begin{cases} if & aunify(V, E, \emptyset) = (V, E) \\ & then\ \ (V, E) \\ else & abs\_unify(V', E')\ where\ (V', E') = aunify(V, E, \emptyset) \end{cases}
$$

The function *aunify* has three input parameters: V, E, E'. V is the same as in *abs_unify*, E and E' are sets of normalized unification equations. This function is invoked by the function *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)

$$
\begin{array}{l}
aunify(V, \{X = Term\} \cup E, E') = \\
= \left\{
\begin{array}{ll}
if & (X/G) \in V \;\; then \;\; aunify(V - \{(Y/\_)|Y \in vars(Term)\} \cup \\
 & \qquad\qquad\qquad \{(Y/G)|Y \in vars(Term)\}, E, E') \\
if & vars(Term) = \emptyset \; or \; \forall Y \in vars(Term).\ (Y/G) \in V \\
 & then \;\; aunify(V - \{X/\_\} \cup \{X/G\}, E, E') \\
if & Term \equiv Y \; and \; (X/F) \in V \; and \; (Y/F) \in V \\
 & then \;\; aunify(V, E, E' \cup \{X = Term\}) \\
if & Term \equiv Y \; and \; ((X/NF) \in V \; or \; (Y/NF) \in V) \\
 & then \;\; aunify(V - \{X/\_, Y/\_\} \cup \{X/NF, Y/NF\}, \\
 & \qquad\qquad\qquad E, E' \cup \{X = Term\}) \\
if & Term \equiv Y \; and \; (X/NF(e)) \in V \; and \; (Y/F) \in V \\
 & then \;\; aunify(V - \{Y/F\} \cup \{Y/NF(e)\}, E, E' \cup \{X = Term\}) \\
if & Term \equiv Y \; and \; (X/F) \in V \; and \; (Y/NF(e)) \in V \\
 & then \;\; aunify(V - \{X/F\} \cup \{X/NF(e)\}, E, E' \cup \{X = Term\}) \\
if & Term \equiv Y \; and \; (X/NF(e)) \in V \; and \; (Y/NF(e')) \in V \; and \; e \not\equiv e' \\
 & then \;\; aunify(V - \{X/NF(e), Y/NF(e')\} \cup \{X/NF, Y/NF\}, \\
 & \qquad\qquad\qquad E, E' \cup \{X = Term\}) \\
if & Term \equiv Y \; and \; (X/NF(e)) \in V \; and \; (Y/NF(e)) \in V \\
 & then \;\; aunify(V, E, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \; and \; (X/F) \in V \\
 & then \;\; aunify(V - \{X/F\} \cup \{X/NF(X = Term)\}, \\
 & \qquad\qquad\qquad E, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \; and \; (X/NF(X = f(t_1, \ldots, t_n)) \in V \\
 & then \;\; aunify(V, E, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \; and \; (X/NF(e)) \in V \;\; and \;\; e \not\equiv X = Term \\
 & then \;\; aunify(V - \{X/NF(e)\} - \{(Y/\_)|Y \in vars(Term)\} \\
 & \cup\{X/NF\} \cup \{(Y/NF)|Y \in vars(Term)\}, E, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \; and \; (X/NF) \in V \\
 & then \;\; aunify(V - \{(Y/\_)|Y \in vars(Term)\} \cup \\
 & \qquad\qquad\qquad \{(Y/NF)|Y \in vars(Term)\}, E, E' \cup \{X = Term\})
\end{array}
\right. \\
aunify(V, \emptyset, E) = (V, E)
\end{array}
$$

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 λ*_share* and computes *Sg_share* (the updated sharing information for variables in Sg) as per the information in V. Consider a set $S \in \lambda\_share$. S is added to *Sg_share*, if and only if S does not have a variable which is ground according to V.

Definition 17 (update_sharing)

$$update\_sharing(V, SS) = \{S \mid S \in SS,\ \ \forall X \in S.\ X/G \notin V\}$$

Along with *aunify*, the function *partition* forms the "core" of our abstract unification algorithm. Its three input parameters are V, E and $Share$. V and E are as before, while $Share$ gives the sharing information among the variables in Sg and/or *head*(C). Making use of these three input values, this function computes the partitions of the "connection graph" for the variables in Sg and *head*(C).

First, we consider the cases when $E \neq \emptyset$ and therefore an equation $X = Term$ is in E.

- If $(X/NF(X = Term)) \in V$, then this equation introduces sharing *only*[6] between

[6] Most of the other algorithms, including the algorithm that we had published earlier [14], introduce a sharing between Y and Y' and thus lose precision. In this case, we are able to avoid this pitfall only because we have the additional *freeness* information.

X and an $Y \in vars(Term)$ and *not* between Y and Y' where $Y, Y' \in vars(Term)$.
• If $(X/NF) \in V$, then sharing is introduced not only between X and Y but also between Y and Y' for all $Y, Y' \in vars(Term)$.
• If $(X/F) \in V$, then $Term \equiv Y$ 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 $vars(Share)$ (i.e. X's partition already exists), then nothing is done. Otherwise, a new partition containing only X is added.

Definition 18 (partition)

$$partition(V, \{X = Term\} \cup E, Share) =$$
$$= \begin{cases} if \ \ (X/NF(X = Term)) \in V \\ \quad then \ \ partition(V, E, Share) - \{P | P \in partition(V, E, Share), \\ \qquad\quad X \in P \ or \ vars(Term) \cap P \neq \emptyset\} \cup \{P_1 \cup P_2 | P_1, P_2 \in \\ \qquad\quad partition(V, E, Share), X \in P_1, vars(Term) \cap P_2 \neq \emptyset\} \\ if \ \ (X/NF) \in V \\ \quad then \ \ partition(V, E, Share) - \{P | P \in partition(V, E, Share), \\ \qquad\quad X \in P \ or \ vars(Term) \cap P \neq \emptyset\} \cup \{\bigcup P | P \in \\ \qquad\quad partition(V, E, Share), X \in P \ or \ vars(Term) \cap P \neq \emptyset\} \\ if \ \ (X/F) \in V \\ \quad then \ \ partition(V, E, Share) - \{P_1, P_2\} \cup \{P_1 \cup P_2\} \\ \qquad\quad where \ Term = Y \ and \ X \in P_1 \ and \ Y \in P_2 \ and \\ \qquad\quad P_1, P_2 \in partition(V, E, Share) \end{cases}$$

$$partition(V, \emptyset, Share) = \{\{X\} \mid X/Fs \in V, Fs \neq G, X \notin vars(Share)\} \cup Share$$

Definition 19 (powerset_of_set_of_sets)

$$powerset\_of\_set\_of\_sets(SS) = \bigcup_{S \in SS} \wp(S)$$

The function *collapse_non_free* is needed because we use more than *three* values for freeness viz, $G, F, NF, NF(e)$ while performin g 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)

$$collapse\_non\_free(V) =$$
$$\{X/Fs \mid X/Fs' \in V, \ Fs \leftarrow if \ (Fs' = NF(e)) \ then \ NF \ else \ Fs'\}$$

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

Definition 21 (project)

$$project(Term, (Asubst_{share}, Asubst_{freeness})) =$$
$$(project\_share(Term, Asubst_{share}),$$
$$project\_freeness(Term, Asubst_{freeness}))$$

The function *project_share*, projects $Asubst_{share}$ (the *sharing* component of *Asubst*) on $Term$.

Definition 22 (project_share)

$$project\_share(Term, Asubst_{share}) =$$
$$\{S | S = (S' \cap vars(Term)), S' \in Asubst_{share}\}$$

The function *project_freeness* projects $Asubst_{freeness}$ (the *freeness* component of $Asubst$) on $Term$.

Definition 23 (project_freeness)

$$project\_freeness(Term, Asubst_{freeness}) = \{X/Fs | X/Fs \in Asubst_{freeness}, X \in vars(Term)\}$$

Given the inputs Sg (the subgoal), λ_{call} (the call substitution for sg), and λ' (the projection of the success substitution on Sg), the function *extend* computes $\lambda_{success}$ (the success substitution for Sg).

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

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

Definition 24 (extend)

$$\begin{aligned}
& extend(Sg, \lambda_{call}, \lambda') = (\lambda_share_{success}, \lambda_freeness_{success}) \\
& \quad where \;\; \lambda_freeness_{success} = \lambda_freeness' \cup \{X/G | X \in vars(\lambda_share_{call}) - \\
& \qquad\quad vars(\lambda_share_{success})\} \cup \{X/Fs | X \in vars(\lambda_share_{success}) - \\
& \qquad\quad vars(\lambda_share'), Fs \leftarrow if\ (X/NF \in \lambda_freeness_{call} \vee (\exists Y \exists S.\ X \in S, \\
& \qquad\quad Y \in S, S \in \lambda_share_{success}, Y/NF \in \lambda_freeness'))\ then\ NF\ else\ F\} \\
& \quad and \;\; \lambda_share_{success} = \{S | S \in \{S' | S' \in \lambda_share_{call}, S' \cap vars(Sg) \neq \emptyset\}^*, \\
& \qquad\quad S \cap vars(Sg) \in \lambda_share'\} \cup \{S | S \in \lambda_share_{call}, S \cap vars(Sg) = \emptyset\} \\
& \quad and \;\; (\lambda_share_{call}, \lambda_freeness_{call}) = \lambda_{call} \\
& \quad and \;\; (\lambda_share', \lambda_freeness') = \lambda'
\end{aligned}$$

Proposition 1 *Given a subgoal Sg whose abstract call substitution is λ_{call} and a clause C whose head unifies with Sg, let β_{entry} be the abstract entry substitution for C as computed by the function call_to_entry.*[7] *Then, β_{entry} is a safe approximation in the following sense: In the concrete interpretation, let Ω_{entry} be the set of entry substitutions for clause C computed from Sg's set of call substitutions $\gamma(\lambda_{call})$. Then, $\Omega_{entry} \subseteq \gamma(\beta_{entry})$.*

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

5 Example

We illustrate the algorithm *call_to_entry* in section 4 with the aid of an example[8]. This example is rather contrived and its main function is to illustrate the mechanics of the algorithm as it deals with different cases.

[7] A similar proposition about the safety of the abstract substitution λ' computed by the function *exit_to_success* can also be stated and proved. However, due to lack of space, we do not present it here.

[8] The function *exit_to_success* is similar to this function and therefore not illustrated.

Subgoal Sg	$pred(f(X_1, X_2), f(X_3, X_4), X_3, X_5, X_6, X_6)$
Head of clause C	$pred(Y_1, Y_1, a, Y_2, Y_2, f(Y_1, Y_3))$
λ_share_{call}	$\{\emptyset, \{X_2\}, \{X_3\}, \{X_5\}, \{X_6\}, \{X_1, X_2\}\}$
$\lambda\_freeness_{call}$	$\{X_1/F, X_2/NF, X_3/F, X_4/G, X_5/F, X_6/F\}$

Let $vars(body(C)) = \{Y_4, Y_5\}$. In the following, we illustrate, in a step-by-step fashion, how β_{entry}, the entry substitution for the clause C is computed given the above information:

• $simplify\_equations(\{Sg = head(C)\}) = \{Y_1 = f(X_1, X_2), Y_1 = f(X_3, X_4), X_3 = a, Y_2 = X_5, Y_2 = X_6, X_6 = f(Y_1, Y_3)\}$

• The computation of (V, E) using the function *abs_unify* is long and we won't show it in full detail. Rather, we highlight some important steps which illustrate the key cases considered in the function *aunify* and also the fact that fixpoint computation is performed when the function *abs_unify* is called.

• During the first round of fixpoint computation (using the function *aunify*), the freeness value of Y_1 is changed first from F to $NF(Y_1 = f(X_1, X_2))$ and then to NF as the equation $Y_1 = f(X_3, X_4)$ is considered. This changes the freeness value of X_3 to NF. Subsequently, this value is changed to G after the equation $X_3 = a$. Finally, the freeness value of X_6 is changed to $NF(X_6 = f(Y_1, Y_3))$.

• In the second round of fixpoint computation, the freeness value of X_1 is changed from F to NF since the freeness value of Y_1 is now NF. Subsequently, the freeness value of Y_1 is changed from NF to G when the equation $Y_1 = f(X_3, X_4)$ is considered; now, the freeness values of both X_3 and X_4 are G. The freeness value of Y_2 is changed from F to $NF(X_6 = f(Y_1, Y_3))$ after the equation $Y_2 = X_6$.

• After two more rounds of using the function *aunify*, fixpoint is reached and the final value of V is $\{X_1/G, X_2/G, X_3/G, X_4/G, X_5/NF(X_6 = f(Y_1, Y_3)), X_6/NF(X_6 = f(Y_1, Y_3)), Y_1/G,\ Y_2/NF(X_6 = f(Y_1, Y_3)), Y_3 = F\}$ and the final value of E is $\{X_5 = Y_2, X_6 = Y_2, X_6 = f(Y_1, Y_3)\}$.

$Sg\_share = update\_sharing(V, \lambda\_share) = \{\emptyset, \{X_5\}, \{X_6\}\}$
$partition(V, E, Sg\_share) = \{\{X_5, X_6, Y_2, Y_3\}\}$
$project\_share(head(C), partition(V, E, Sg\_share)) = \{\{Y_2, Y_3\}\}$
$\beta\_share = \{\emptyset, \{Y_2\}, \{Y_3\}, \{Y_2, Y_3\}\}$,
$\beta\_freeness_{entry} = \{Y_1/G, Y_2/NF, Y_3/F\}$
$\mathcal{P}(Sg, Sg\_share) = \{\{4\}, \{5, 6\}\}$
$(\mathcal{P}(Sg, Sg\_share))^* = \{\{4\}, \{5, 6\}, \{4, 5, 6\}\}$.
$\beta\_share_{entry} = \{\emptyset, \{Y_2, Y_3\}\}$.
$call\_to\_entry(\lambda, Sg, C) = (\{\emptyset, \{Y_2, Y_3\}\}, \{Y_1/G, Y_2/NF, Y_3/F\})$

6 Synergistic Interaction between Freeness and Sharing

In section 1.1, we saw how freeness information could help to increase the accuracy of sharing information achievable by analysis. In this section, we show how the converse works i.e. how the presence of the sharing component leads to a more accurate estimation of the freeness component. Thereby, we demonstrate the symmetric interaction and synergy that exists between these two components of the abstract substitution.

Consider a subgoal $Sg \equiv pred(X_1, X_2)$ whose call substitution is $(\{\emptyset,\{X_1\},\{X_3\},\{X_2,X_4\}\}, \{X_1/F,X_2/F,X_3/F,X_4/F\})$. Let 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 $(\{\emptyset,\ \{X_1\},\ \{X_3\},\ \{X_2,\ X_4\}\},\ \{X_1/NF,\ X_2/NF,\ X_3/F,\ X_4/NF\})$. 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 Sg, 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 Sg? This can be explained by the fact that X_3 is not aliased to any other variable in Sg's call substitution while X_4 is potentially aliased to X_2. Therefore, X_3 is not affected by the execution of Sg 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 Sg'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

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A declarative approach to the abstract interpretation of logic programs. Technical Report TR-20/89, Dipartamento di Informatica, Universita di Pisa, 1989.

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

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

[4] M. Codish, D. Dams, and E. Yardeni. Abstract unification for the analysis of groundness and aliasing in logic programs. Technical Report TR-CS90-10, Weizmann Institute of Computer Science, August 1990.

[5] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232, October 1990.

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

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

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

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

[10] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[11] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

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

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

[14] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[15] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. Technical Report STP-368-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, December 1990.

[16] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

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

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

[19] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.