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

Renganathan Sundararajan
e-mail: renga@cs.uoregon.edu

CIS-TR-91-06 (Revised)
October 91

Abstract

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

Although simple data flow analyses of logic programs have been shown to be practical by other researchers, it is believed that more informative analyses such as the one presented in this paper are impractical. We show that this is not necessarily so. We defined and implemented an efficient and domain-independent abstract interpreter for computing Minimal Function Graph semantics of logic programs and instantiated it with the abstract domain and corresponding operations presented in this paper. The analysis times compare very favorably with those reported in the literature for much simpler global flow analyses of logic programs. This research is part of a larger effort aimed at efficient parallel execution of logic programs.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

*This research was partly supported by a Tektronix Fellowship

1 Overview

Although the formal theory of abstract interpretation was developed originally for analyzing programs in procedural languages [7], most of the recent work in the field has concentrated on languages that have a large declarative component. We assume the reader is familiar with the fundamentals of horn clause logic programming, including resolution. If not, please refer to [16]. In the logic programming world, abstract interpretation frameworks for static analyses were initially proposed by Bruynooghe [3], Jones and Sondergaard [15] and Mellish [19]. There have been a number of contributions [2, 4, 8, 9, 13, 17, 18, 20, 22, 25] to this field. Some of the proposals do bottom-up analyses and others top-down. Top-down analyses model SLD-resolution and hence aim at deriving assertions about execution states at various program points reached in a conventional top-down execution of logic programs. Bottom-up analyses model the $\mathbf{T_P}$ semantics [1, 16].

1.1 Motivation

Abstract Interpretation based analyses for inferring groundness and independence of variables are useful for eliminating or minimizing run-time checks for groundness and/or independence of variables in parallel execution models that support Independent-And Parallelism (IAP) [6, 12]. In IAP models, two goals in the body of a clause cannot be executed in parallel unless the variables that occur in them are pair-wise independent, i.e, are bound to terms that do not share any common variable sub-terms. Run-time tests for independence of variables can be very costly; $\mathcal{O}(mn)$ where m and n are the number of nodes in the tree representation of terms that the variables are bound to. If we can detect at compile-time that there is no possibility of sharing among the variables of two literals, we can dispense with the tests.

Very often programmers use a technique called *difference lists* to convert linear time concatenation of lists into a constant time operation. The following quicksort example illustrates this.

```
quicksort(X, Y) :-                                        ..... 1
    quicksort(X, Y-[]).
quicksort([], Sorted-Sorted).                             ..... 2
quicksort([X|Unsorted], Sorted-SoFar) :-                  ..... 3
    split(Unsorted, X, Small, Large),
    quicksort(Small, Sorted-[X|LargeSorted]),
    quicksort(Large, LargeSorted-SoFar).
:- quicksort([5,2,3,9], Sorted).
```

Although the two recursive calls in the body of clause 3 share a variable, *LargeSorted*, only the second call will instantiate it. The first recursive call will not bind it to a non-variable term, i.e, leave it *free*. The method described in [10] can detect the occurrence of free variables in some cases and Winsborough [26] provides a more complicated method for detecting more cases of free variable occurrences. Detection of free variables allows us to parallelize the two recursive calls by a simple source to source program transformation.

1.2 Brief Research Review and Contributions

Jacobs [13] defined an abstract domain, *Sharing*, to express variable sharing and independence information. Muthukumar [21] extended the domain *Sharing* to derive *freeness* of variables and suggested an alternative set of abstract domain operations over *Sharing*. In both cases, if variables X and Y may share and Y and Z may share, they *will have to* conclude that X and Z may also share. As Jones and Sondergaard point out [15], sharing is a non-transitive relation. Treating it as transitive introduces imprecision in sharing propagation.

Jones and Sondergaard presented an abstract interpretation scheme in [15] for sharing analysis. Codish [5] used the abstract domains of Jones and Sondergaard and formally rederived an abstract unification algorithm. The abstract domains of [15] are less expressive than Jacobs' (see [13] for examples) but sharing propagation is better since they don't treat sharing as transitive. A brief look at the abstract unify algorithms of [15, 5] will show that their time complexity is dominated by transitive closures (of another relation called *link*) and hence $\mathcal{O}(n^3)$.

Static analysis proposed by Debray [10] for deriving input, output modes of procedures can detect the occurrence of free variables in some cases. Winsborough [26] provides a more complicated method for detecting more cases of free variable occurrences.

In this paper, we extend the domain *Sharing* to derive the *freeness, sharing, repeatedness* and *groundness* of variables more precisely than previous proposals. Although the worst case time complexity of our abstract unification algorithm is exponential in the input size, (as those of [13] and [21]) we show that the expected behavior of our algorithm is much better than that of previous algorithms for the following reasons. The exponential component of our abstract domain operation is executed rarely, only when terms with repeated variable sub-terms occur. Such occurrences are not frequent in practical programs, whereas computing transitive closures is an integral part of sharing propagation in [15] and [5].

To substantiate our claim, we defined and implemented a *domain independent* abstract interpreter for computing Minimal Function Graph semantics [14] of logic programs [23] (that do not include *assert*, *retract*, *call* and other non-logical and meta-logical features). We instantiated this interpreter with the abstract domains and operations of Jones and Sondergaard [15], Jacobs and Langen [13], and our own. Analyses of *non-trivial*, and often used programs show that our analysis captures variable sharing and independence information more accurately than previous proposals. These analyses also confirm our belief that the execution time of our algorithm (at least for the examples we tested) is much less than other algorithms, including the polynomial time algorithms of [15, 5]. Furthermore, analysis times for the complex domains and operations proposed here compare very favorably with the times reported in the literature for *much simpler* analyses.

1.3 Definitions

A substitution $\sigma : Subst = Var \to Term$ is a mapping $\{v_i \mapsto t_i, \ldots\}$ where the v_i are distinct, $\mathrm{dom}(\sigma) = \{v_i \mid v_i \mapsto t_i \in \sigma \wedge t_i \neq v_i\}$, $\mathrm{range}(\sigma) = \{t_i \mid v_i \mapsto t_i \in \sigma \wedge t_i \neq v_i\}$ and v_i do not occur in any of the t_i. *Var* is the set of all variables and *Term* is the set of all first order terms. The definition of substitution can be extended homomorphically from $Var \to Term$ to $Term \to Term$. The composition of substitutions, denoted by $\sigma_j \circ \sigma_i$ is defined as $\lambda\, x.\, \sigma_j(\sigma_i\, x)$ where $x \in Term$, and $\circ$ associates to the right. The domain of the composition is $dom(\sigma_i) \cup dom(\sigma_j)$. Substitutions, the way we have defined them, are idempotent. In this paper, we deal with idempotent substitutions only and use $\sigma, \sigma' \cdots \theta, \theta' \cdots$ to denote substitutions, $|\sigma|_V$ for the restriction of $\mathrm{dom}(\sigma)$ to V.

There is a natural pre-ordering of substitutions: $\sigma_1 \leq \sigma_2$ iff $\exists \sigma_3 \in Subst$ such that $\sigma_2 = \sigma_3 \circ \sigma_1$, i.e, σ_1 is more general than σ_2. A unifier σ of two terms t_1, t_2 is a substitution such that $\sigma t_1 = \sigma t_2$, i.e, the term obtained by applying the substitution σ_1 to t_1 is syntactically identical to the term obtained by applying σ_2 to t_2. The most general unifier of two terms is a unifier which is least with respect to $\leq$ (modulo variable renaming). Define a function $mgu : Atom \times Atom \to Subst \cup \{fail\}$ that yields the most general unifier of two atoms if there is one and returns *fail* otherwise. In the rest of the paper we assume that the variables in a program have been renamed so that a variable does not occur in more than one clause. Hence we can refer unambiguously to the set of program variables *Pvar*. In the rest of the paper, we use *vars(A)* to denote the set of variables in A which may be a term, body, head, or an abstract domain (to be defined later).

2 Abstract Interpretation

Cousot and Cousot [7] formalized the idea that static analyses of programs can be viewed in terms of abstract interpretation. Their original work was done in the context of analysis of flow chart programs which have nodes, states associated with the nodes and program points between nodes. For logic programs, the counterpart to states is substitutions; literals correspond to nodes and a program point is between two literals in the body of a clause. In the abstract interpretation framework, it is customary to define a core semantics for the language, leaving some semantic domains and semantic functions (such as unification) unspecified. An interpretation I of the core semantics supplies the missing domains and semantic functions. A collecting semantics, which is an interpretation of the core semantics, associates with each program point a set of all substitutions that may prevail at that point during any execution of the goal(s). The collecting semantics is like the standard semantics for the language except that it gathers and preserves the association between program points and the sets of substitutions encountered at these points during an execution.

All analyses are done using abstract interpretations which are approximations of the collecting semantics. An interpretation I of the core semantics is an abstract interpretation if the domains and semantic functions of I are abstractions of their counterparts in the collecting semantics and if I safely models the collecting semantics. The domains left unspecified in core semantics must be either a complete lattice or at least a complete partial order (cpo) [14]. In an abstract interpretation, the domains are also required to be of finite height to ensure termination. The notion of soundness or safety is stated in terms of modeling relations. A modeling relation between the abstract domain *Asub* and the corresponding standard domain $\mathcal{P}(Subst)$ is given by two continuous functions α and γ

$$\alpha : \mathcal{P}(Subst) \rightarrow Asub, \qquad \gamma : Asub \rightarrow \mathcal{P}(Subst)$$

such that for any $\Theta \in \mathcal{P}(Subst)$ and $\Phi \in Asub$

$$\Theta \subseteq \gamma(\alpha(\Theta)) \quad \text{and} \quad \Phi = \alpha(\gamma(\Phi)).$$

The function α is called the abstraction function and γ the concretization function. Furthermore, each abstract function f over an abstract domain corresponding to a concrete function $\mathtt{f}$ over the standard domain must obey the following constraint. For every $\mathtt{x}$ in the standard domain, $\mathtt{f}(\mathtt{x}) \subseteq \gamma(f(\alpha(\mathtt{x})))$. That is, the abstracted function f when applied to the abstraction of an object $\mathtt{x}$ of the standard domain must yield an an abstract object y such that its *(y's)* concretization must contain *(describe)* the set of objects of the standard domain obtained by applying $\mathtt{f}$ to $\mathtt{x}$.

2.1 Abstract Interpretation Framework for Logic Programs

We briefly describe the standard interpretation of logic programs before discussing the proposed abstract interpretation. We are given a program P, a goal G = $\{a_i, \ldots, a_n\}$ and a current substitution θ. A subgoal a_m is selected using a *fair computation rule* [16] and removed from the set of subgoals $\{a_i, \ldots a_n\}$. A matching clause j is selected using a *search rule* [16], the variables of clause j are renamed, the head h_j of a clause j is unified with $\theta\, a_m$. If the unification succeeds, the body literals $b_{j1}, \ldots, b_{jk}$ are added to the set of goals to be solved and the new current substitution is $\theta' \circ \theta$, where θ' is the mgu of the head and the goal. The above process is repeated until it terminates because the set of goals is empty or there is a subgoal θa such that it does not unify with the head of any of the clauses. If the set of goals becomes empty, then the current substitution is one answer substitution corresponding to the initial substitution.

Abstract interpretation of a logic program starts with a program or module P and a set of entry points. These entry points can be examples of actual queries together with an initial abstract substitution for the variables in the query. Alternatively, the entry points may be query patterns.

At any point during computation there will be a current substitution. Call it θ. The current substitution is given different names depending on the state of computation. If goal a is about to be unified with the head of clause i, the current substitution (to be applied to the goal) is known as the *call substitution* of goal a. If a has just been successfully unified with the head of clause i (which has been renamed), the unifier is the *entry substitution* for clause i and also the *call substitution* for the first literal in its body. If the last literal in the body has just been solved, the current substitution is known as the *exit substitution* of clause i.

Restricting the exit substitution of a clause i to the non-local variables and then composing it with the call substitution of a gives us the *success substitution* of clause i. In section 3, we use the function *entry_substitution* to safely approximate head unification and produce an initial environment for the body of clause i and the function *success_substitution* to safely approximate the composition of exit substitution of clause i with the call substitution of goal a. The least upper bound (lub) of the set of success substitutions of all the clauses whose heads unified with goal a is called *the* success substitution of goal a. The operation lub is domain dependent.

Given an entry substitution for a clause, its exit substitution is computed as follows. If it is a unit clause, the exit substitution is the same as the entry substitution. Otherwise, compute the success substitution of the body literals sequentially, using the success substitution of literal i as the call substitution of literal $i+1$. The success substitution of the last literal in the clause is the exit substitution of the clause.

In an abstract interpretation over domains of finite height, we can solve the current set of goals $\theta\{a_i, \ldots a_n\}$ depth-first without destroying completeness. Since a program may contain recursive predicates, abstract interpretation involves fixpoint computation. Please see [11, 23, 9, 20] for details.

In next sub-section, we define our abstract domain *Asub* and then discuss the abstract operations *entry_substitution success_substitution* and *abstract_unify* in the following section.

2.2 Abstract Domain *Asub*: *Free* × *Repeat* × *Sharing*

The abstract domain *Asub* is the product of the domains *Free*, *Repeat* and *Sharing*. We first define each of these domains and then present abstraction and concretization functions.

Definition 1

free: *Subst* $\rightarrow \mathcal{P}(Var)$

$free(\sigma) = \{x_i \mid x_i \in dom(\sigma) \wedge x_i \mapsto x_j \in \sigma \wedge x_j \in Var)$ □

Example 1

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

Then $free(\sigma) = \{\mathrm{Y}\}$. □

Definition 2 Abstract domain *Free*

Free: $\mathcal{P}(\mathrm{Var})_\perp$

The partial ordering on *Free* is defined to be

$\perp \sqsubseteq_{Free} x \qquad \forall x \in Free$

$x \sqsubseteq_{Free} y \qquad iff\, x \supseteq y \;\; \forall x, y \in Free \wedge x, y \neq \perp$

and the least upper bound operation is defined as

$\sqcup X = \perp \qquad - \textit{if } X = \{\perp\}$

$\sqcup X = \bigcap(X \setminus \{\perp\}) \qquad - \textit{otherwise}$ □

The lub operation on *Free* is set intersection since we wish to derive *definite freeness* of variables.

Definition 3

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

$repeat(\theta) = \{\ x_j \mid x_j \mapsto t_j \in \theta\ \ \wedge\ \ \exists x_k \in Var$ s.t. x_k occurs more than once in $t_j)\}$ $\square$

$repeat(\theta)$ returns a set of variables that are mapped by θ to terms containing multiple occurrences (i.e, more than one) of at least one variable. It can be shown that $repeat(\theta) \cap free(\theta) = \emptyset,\ \forall\theta$. In the example given above, $repeat(\sigma) = \{W\}$. Note that we lose information about the positions where a repeated variable occurs, in this case, argument positions 1 and 3 of the principal functor *f*.

Definition 4 Abstract domain *Repeat*

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

$\sqsubseteq_{repeat} = \subseteq$ and $X_1 \sqcup X_2 = X_1 \cup X_2$, for $X_1, X_2 \in Repeat$ $\square$

The lub operation on *Repeat* is set union since we want to conservatively estimate the set of variables bound to terms that *may* have repeated occurrences of a variable sub-term.

The following definitions relate to the domain *Sharing*. Definitions [5-9] are due to Jacobs and Langen [13]. A *Sharing* is a set of sets of variables. It expresses the sharing among and independence of variables caused by a substitution. We say that a variable u occurs through a variable v in a given substitution σ, if u occurs in a term bound to v in σ. A set of variables is in a $sharing(\sigma)$ iff there is at least one common variable occurring through all of them in σ. In other words, two variables A, B may share in a substitution σ iff there is a set $X \in sharing(\sigma)$ such that $A \in X$ and $B \in X$. A and B are independent (in σ) otherwise. Further, a variable $A \in dom(\sigma)$ is ground iff it is not a member of any set in $sharing(\sigma)$.

Definition 5

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

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

Definition 6

$sharing : Subst \rightarrow Sharing$

$sharing(\sigma) = \{occ(\sigma, u) \mid u \in Var\}$ $\square$

Definition 7 Abstract Domain *Sharing*

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

$\sqsubseteq_{Sharing} = \subseteq$ and $X_1 \sqcup X_2 = X_1 \cup X_2$, for $X_1, X_2 \in Sharing$ $\square$

Using the σ defined in example 1, $sharing(\sigma) = \{\emptyset, \{W, X, Z\}, \{Y, Z\}, \{Z\}\}$. The variables {A,C} in the range of σ occur through the variables W, X, and Z in σ; {B} occurs through Y, and Z; {D} through Z. It can be shown that $free(\sigma) \subseteq vars(sharing(\sigma))$ and $repeat(\sigma) \subseteq vars(sharing(\sigma))$.

Let a component of a variable X be a non-empty subset of the set of variables in a term t that X is bound to in a substitution σ. The abstract domain *Sharing* represents a variable by its components and collects together in a set all the variables that *share* the same component(s). Thus each set I in a *Sharing* is a set of variables that *share* the same component. Furthermore, we can think of each set I of variables in a *Sharing* as representing a particular component of each of the variables in I. In the above example, variable Z has three components {A,C}, {B} and {D} represented by the sets {W,X,Z}, {Y,Z}, {Z} in *sharing*(σ). It shares {A,C} with W and X, {B} with Y, and {D} with none.

Any variable E which is not in *vars*(*range*(σ)) occurs through the empty set of variables, hence $\emptyset \in$ *sharing*(σ). When a variable is independent of all other variables, it is in a set by itself. Likewise, when a variable X is bound to a ground term in σ, no variable occurs through X and hence X does not appear in any set in *sharing*(σ). By extension, if $\emptyset$ is the only element of *sharing*(σ), then all variables in *dom*(σ) are ground in σ. Note that X and Y are independent since no variable occurs through both of them, and grounding Z grounds W, X, Y but not *vice-versa.* The partial ordering reflects the fact that a *Sharing* is intended to express *possible* (as opposed to *definite*) sharing among the variables.

The following definition of multioccurs determines if a term T may have repeated variable sub-terms given the sharing abstraction of a (set of) substitution(s). A term T may have repeating variable sub-terms if a variable occurs more than once in term T or two distinct variable sub-terms of T may share in the given abstraction.

Definition 8

$multioccurs : Term \times Sharing \rightarrow Bool$
$multioccurs(t, SS)$ *iff* ($\exists X \in vars(t)$ *s.t.* X occurs more than once in t) or
($\exists X, Y \in vars(t),\ X \neq Y,\ \exists S \in SS$ *s.t.* $\{X, Y\} \subseteq S$) □

Definition 9

$relevant : A \times Sharing \rightarrow Sharing$
$relevant(t, S) = \{X \in S \mid vars(t) \cap X \neq \emptyset\}$ □

The function *relevant* takes a term or a set t and a *sharing* S and returns a set of those elements of S that represent a component of a variable in t, i.e, the sets of variables that may share with a variable in t.

Definition 10 Closure under union

The closure under union of a *sharing* S, denoted by S^* is the smallest super set of S such that $X \in S^* \wedge Y \in S^* \Rightarrow X \cup Y \in S^*$. □

The closure under union of a *sharing* S approximates further sharing *among the variables of* S that may be caused by $\sigma_1 \circ \sigma_2$ where σ_1 is any substitution and *sharing*(σ_2) $\sqsubseteq S$.

Definition 11 *Strongly Connected*

sc: Var $\times$ *Var* $\times$ *Sharing* $\rightarrow$ *Bool*
sc(x_i, x_j,S) iff $x_i, x_j \in vars(S) \wedge relevant(x_i, S) = relevant(x_j, S)\}$ □

It can be shown that *sc* is an equivalence relation and it partitions the variables of a *Sharing* S into equivalence classes. All variables $x_1, \cdots, x_n$ in an equivalence class I are bound to (possibly different) terms $t_1, \cdots, t_n$ in the substitution σ where $sharing(\sigma) = S$, such that $vars(t_1) = \cdots = vars(t_n)$. This follows from the definitions of $relevant(x_i, S)$, $sharing(\sigma)$ and $occ(\sigma, u)$. We now define *Asub* as the product of the domains *Free, Repeat* and *Sharing.* The partial ordering on and lub of *Asub* are derived from the component domains.

Definition 12 $Asub = Free \times Repeat \times Sharing$

$$\sqsubseteq_{Asub} \quad = \quad \sqsubseteq_{Free} \times \sqsubseteq_{Repeat} \times \sqsubseteq_{Sharing}$$
$$\sqcup_{Asub} \quad = \quad \sqcup_{Free} \times \sqcup_{Repeat} \times \sqcup_{Sharing} \qquad \square$$

Hereafter we will omit the domain subscripts from $\sqsubseteq$ and $\sqcup$ when it is clear from the context.

Definition 13 Abstraction Function

$\alpha : \mathcal{P}(Subst) \rightarrow Asub$

$$\alpha(\Theta) = \bigsqcup_{\theta \in \Theta} \{(free(\theta), repeat(\theta), sharing(\theta))\} \qquad \square$$

Our definitions of the component domains and their associated orderings make the above abstraction function *safe* in the sense that

- the abstraction of a set of substitutions Θ includes all possible sharing of variables induced by each substitution $\theta \in \Theta$, since set union is the lub operation for *Sharing.* In other words, variables X and Y occur together in a set in the sharing component of the abstract substitution if there is at least one substitution $\theta \in \Theta$ such that $vars(\theta X) \cap vars(\theta Y) \neq \emptyset$.

- a variable X is ground only if it is ground in all the substitutions, i.e, it does not occur in any set in $sharing(\theta)$, $\forall\, \theta \in \Theta$.

- the repeat component of *Asub* includes a variable x_i if, *in at least one substitution* it is bound to a term with a repeated variable occurrence.

- a variable is free in the abstract substitution only if it is free in *all* the substitutions abstracted.

Definition 14 Concretization function

$\gamma : Asub \rightarrow \mathcal{P}(Subst)$

$$\gamma(A) = \{\sigma \in Subst \mid (free(\sigma) \times repeat(\sigma) \times sharing(\sigma)) \sqsubseteq A\} \qquad \square$$

It is easy to verify that our definitions of α and γ fulfill the modeling relation between *Asub* and $\mathcal{P}(Sub)$ specified in section 2. The next function *restrict* is used to restrict an *Asub* to a set of variables.

Definition 15 *restrict*

$restrict : Asub \times \mathcal{P}(Var) \rightarrow Asub$

$$restrict((F, R, S), V) = (F \cap V, R \cap V, \{X' \mid X \in S \ \wedge \ X' = X \cap V\}) \qquad \square$$

3 Operations on Abstract Domains

We now define the functions *entry_substitution*, *success_substitution*, and *abstract_unify*. We use $\downarrow$ for projection and tuple notation for injection. For example, $Asub_{call}\downarrow Repeat$ refers to the *Repeat* component of $Asub_{call}$.

Definition 16 *entry_substitution*: $Atom \times Clause \times Asub \rightarrow Asub$

```
entry_substitution(Goal, Head :- Body, $Asub_{call}$) =
    let  $(H,B)$  $=$  $rename((Head, Body))$                                   1
         $F$      $=$  $Asub_{call}\downarrow Free \cup vars(H) \cup vars(B)$          2
         $R$      $=$  $Asub_{call}\downarrow Repeat$                                 3
         $S$      $=$  $Asub_{call}\downarrow Sharing \cup \{\{X\} \mid X \in vars(H) \cup vars(B)\}$   4
         $ESub$   $=$  $abstract\_unify(Goal, H, (F, R, S))$                    5
    in   $rename^{-1}(restrict(Esub, vars(H) \cup vars(B)))$                  6
```

□

The above function first creates an initial environment (*F*,*R*,*S*) from the call substitution and the renamed clause. *abstract_unify* updates the initial environment to approximate head unification and the result *Esub* is restricted to the variables of the clause and then inverse of *rename* applied. Let $\theta : Pvar \rightarrow Var$ be a renaming substitution which maps program variables to variables that do not occur in a program and θ^{-1} be its inverse. Then *rename* is a homomorphic extension of θ to $Term \rightarrow Term$ and $rename^{-1}$ is a homomorphic extension of θ^{-1} to $Asub \rightarrow Asub$. Step one renames clause variables to avoid conflicts with goal variables. Union of the set of free variables of the call substitution and the set of renamed clause variables gives the set of free variables just before unification. The set of variables bound to terms with repeated variable sub-terms (just before head unification) is the same as that of call substitution, since all renamed clause variables are free at this point (step three). Before head unification, all renamed clause variables are independent of each other (step four). *Esub* is the substitution induced by *Goal* when abstractly unified with the renamed head *H* in the environment *(F,R,S)*. In step six, *Esub* is restricted to renamed clause variables and then inverse of the renaming function applied.

Definition 17 *success_substitution*: $Atom \times Clause \times Asub \times Asub \rightarrow Asub$

```
success_substitution(Goal, Head :- Body, $Asub_{call}$, $Asub_{exit0}$) =
    let  $(H, B, Asub_{exit})$  $=$  $rename'((Head, Body, Asub_{exit0}))$   1
         $Asub_{both}$         $=$  $Asub_{exit} \cup Asub_{call}$              2
         $Asub_{succ0}$        $=$  $abstract\_unify(Goal, H, Asub_{both})$    3
    in   $restrict(Asub_{succ0}, Pvar)$                                     4
```

□

Success_substitution is very similar to *entry_substitution*. We use the $Asub_{exit}$ (after consistent renaming) as the abstract substitution for the clause variables. After abstract unification is used to propagate information from the clause to the goal, the result is restricted to program variables (i.e, variables in the environment of the goal). *rename*$'$ renames the variables in *(Head, Body, Asub)* consistently to non-program variables. We now define *abstract_unify* and *propagate_frs*.

Definition 18 *abstract_unify*: $Atom \times Atom \times Asub \rightarrow Asub$

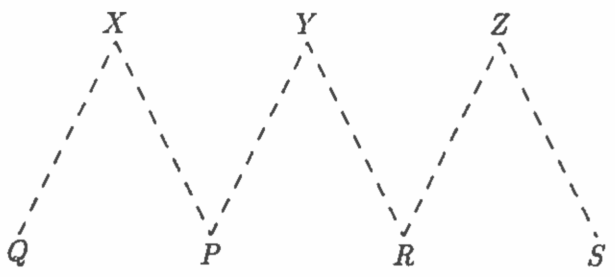
```
  abstract_unify(Goal, Head, $Asub_{in}$) =
    let $\theta$          = $mgu(Goal, Head)$                                                          1
        $Ground_0$        = $(vars(Goal) \cup vars(Head)) \setminus vars(Asub_{in}{\downarrow}Sharing)$                 2
        $Ground_1$        = $Ground_0 \cup \{X \mid V_i \mapsto T_i \in \theta \wedge V_i \in Ground_0 \wedge X \in vars(T_i)\}$   3
        $Ground$          = $Ground_1 \cup \{V_i \mid V_i \mapsto T_i \in \theta \wedge vars(T_i) \subseteq Ground_1\}$          4
        $\theta_0$        = $\{x_i \mapsto t_i \mid x_i \mapsto t_i' \in \theta \wedge x_i \notin Ground \wedge t_i = update(t_i', Ground)\}$   5
        $Sharing_0$       = $Asub_{in}{\downarrow}Sharing \setminus relevant(Ground, Asub_{in}{\downarrow}Sharing)$         6
        $Repeat_0$        = $Asub_{in}{\downarrow}Repeat \setminus Ground$                                     7
        $Free_0$          = $Asub_{in}{\downarrow}Free \setminus \bigcup relevant(Ground, Asub_{in}{\downarrow}Sharing)$      8
    in $propagate\_frs((Free_0, Repeat_0, Sharing_0), \theta_0)$                                          9
```

□

Head unification is approximated by *abstract_unify*. First, the mgu θ of *Goal* and *Head* is obtained and then groundness information is derived. The set of ground variables *Ground* is used to refine *Free*, *Sharing* and *Repeat* components of $Asub_{in}$, and the mgu θ. Since *propagate_frs* performs an imprecise but safe approximation of sharing in the case of variables which may be bound to terms with repeated variable sub-terms, identification and removal of ground variables from $Asub_{in}{\downarrow}Repeat$ improves the precision of the analysis.

- Recall that if a goal (head) variable does not occur in the sharing component of the abstract substitution for the goal (head), then it is ground in all the corresponding concrete substitutions (see the discussion following definition 7). Step two computes the initial set $Ground_0$ of goal and head variables that are ground in all concretizations of $Asub_{in}{\downarrow}Sharing$.

- Steps three and four compute the set of ground variables implied by $Asub_{in}{\downarrow}Sharing$ and θ. Since substitutions are idempotent, a substitution can be viewed as a bipartite graph [15]. Let $\theta = \{X \mapsto f(Q,P), Y \mapsto g(P,R), Z \mapsto h(R,S)\}$ be represented by the following graph. Further assume that $Ground_0 = \{X, R\}$. Step three corresponds to propagating information downwards in the graph which yields $Ground_1 = \{X, P, Q, R\}$. Step four corresponds to upwards information propagation, resulting in $Ground = \{X, Y, P, Q, R\}$.



- Step five refines θ to reflect the groundness of variables. We haven't formally defined the function *update: Term* $\times$ $\mathcal{P}$*(Var)* $\rightarrow$ *Term*. Informally, *update* replaces ground variables in a term T by constants.

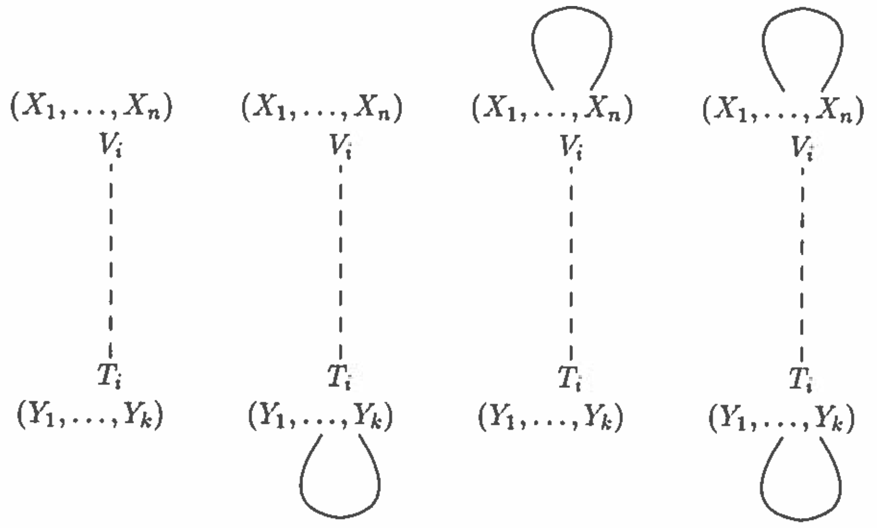
- Step six updates $Asub_{in}\downarrow Sharing$ to reflect the groundness of variables in *Ground*. Likewise, variables bound to ground terms are removed from $Asub_{in}\downarrow Repeat$, the set of variables that may be bound to terms with repeating variable sub-terms (step seven). In step eight, ground variables and other variables that may share with the ground variables are removed from the set of free variables. Function *propagate_frs* is discussed next.

Definition 19 *propagate_frs: Asub* $\times$ *Subst* $\rightarrow$ *Asub*

```
propagate_frs((F, R, S), σ) =
    if  σ = {} then (F, R, S)
    else
      let  $Binding = V_i \rightarrow T_i \in \sigma, \quad \sigma' = \sigma \setminus \{Binding\}$
           $A = relevant(V_i, S), \quad B = relevant(T_i, S)$
           $B' =$ if $V_i \in R$ then $B^*$ else $B$ fi
           $A' =$ if $(vars(T_i) \cap R \neq \emptyset \vee multioccurs(T_i, S))$ then $A^*$ else $A$ fi
           $S' = (S \setminus (A \cup B)) \cup$ pairwise union of $A'$ and $B'$
           $R' = R \cup$ (if $V_i \in R$ then $\bigcup B$ else $\emptyset$ fi)
           $R'' = R' \cup$ (if $(vars(T_i) \cap R \neq \emptyset \vee multioccurs(T_i, S'))$
                       then $\bigcup A$ else $\emptyset$ fi)
           $R''' = R'' \cup (\bigcup\{X \cap Y \mid X \in A', Y \in B'\})$
           $F' = Free \setminus$ (if $V_i \notin F$ then $\bigcup relevant(T_i, S')$ else $\emptyset$ fi)
           $F'' = F' \setminus$ (if $(T_i \notin Var \vee T_i \notin F')$ then $\bigcup relevant(V_i, S)$ else $\emptyset$ fi)
      in   $propagate\_frs((F'', R''', S'), \sigma')$
    fi
```

□

In the following discussion, we refer to the current abstract substitution as $Asub_{cur}$ and its components *Free, Repeat* and *Sharing* as F, R and S respectively. Let A be the set of sets in *sharing* S that are *relevant* to V_i (i.e, each set X in A represents some component of V_i that V_i may share with other variables in X) and B be the one relevant to T_i.



(a) (b) (c) (d)

Each binding $V_i \mapsto T_i$ in σ (the mgu of the head and the goal) propagates *sharing*, *repeatedness* and *freeness* as follows. According to $Asub_{cur}$, V_i and T_i may or may not have repeating variable sub-terms, leading to four possibilities as illustrated below. A solid edge from V_i (or T_i) to itself indicates that θV_i (or θT_i) may have a repeated variable occurrence, where $\theta \in \gamma(Asub_{cur})$. Broken edges represent unification. θV_i is represented as a generic term $(X_1, \ldots X_n)$ where $X_1, \ldots X_n$ are the variables that occur in θV_i and likewise θT_i is represented as $(Y_1, \ldots Y_k)$, without loss of generality.

The binding $V_i \mapsto T_i$ may cause three types of sharing: sharing among the variable sub-terms of V_i (and hence among other variables which share with V_i); sharing among the variable sub-terms of T_i (and hence among other variables which share with T_i); and sharing between any sub-term of V_i on the one hand and any of T_i on the other. The effects of these three types of sharing are safely approximated as follows.

(a) Neither θV_i nor θT_i has a repeated variable sub-term. This means that the only sharing caused by the binding $V_i \mapsto T_i$ is the sharing between any sub-term of V_i and any sub-term of T_i and this is approximated by the pairwise union of A and B.

(b) θT_i may have a repeated variable sub-term, but θV_i doesn't. The repeated variable(s) in θT_i may cause some sub-terms of θV_i to share, which is approximated by taking the closure under union of the sharing sets relevant to V_i $(= A')$. However, no further sharing among the sub-terms of θT_i is caused by the binding $V_i \mapsto T_i$. Finally, the sharing between any sub-term of V_i and that of T_i is approximated by the pairwise union of A' and B.

(c) is symmetric to **(b)**.

(d) can be derived analogously.

We now discuss the propagation of *repeatedness* due to the binding $V_i \mapsto T_i$. If θV_i may have a repeating variable sub-term, then we have to conclude that all variables in θT_i may have the same property and so does any other variable which may share with T_i. Thus $R' = R \cup \bigcup relevant(T_i, S)$. Likewise, if there is a repeating variable in θT_i (before unifying θT_i with V_i, indicated by $vars(T_i) \cap R \neq \emptyset$) or two variable subterms of T_i are brought to share due to the unification of θT_i and V_i, then V_i and other variables that may share with V_i may now have repeating variable sub-terms. Hence, $R'' = R' \cup \bigcup relevant(V_i, S)$.

The last step in the propagation of repeatedness safely approximates the case where two previously distinct variable sub-terms of θV_i or θT_i may now be aliased to each other. Consider the substitution $\theta' = \{P \mapsto (T, U), Q \mapsto (T, U), R \mapsto (U, N)\}$ and the sharing component of its abstraction $Asub_{\theta'}{\downarrow}Sharing = \{\{P, Q\}, \{P, R\}, \{R\}\}$. Suppose that P is now unified with R. The new sharing $\{\{P,Q,R\}, \{P,R\}\}$ is obtained by taking the pairwise union of $relevant(P, Asub_{\theta'}{\downarrow}Sharing) = \{\{P, Q\}, \{P, R\}\}$ (call it A') and $relevant(R, Asub_{\theta'}{\downarrow}Sharing) = \{\{R\}, \{P, R\}\}$ (call it B'). Two previously distinct components T and U of P are now aliased to each other and P and Q will have a repeated variable occurrence. Such cases can occur only when $X \cap Y \neq \emptyset$ where $X \in A'$ and $Y \in B'$. Each variable in the intersection of X and Y may (but not necessarily) have two of their previously distinct components aliased. We safely conclude that each such variable may now be bound to a term with repeating variable sub-terms.

Freeness propagation can be reasoned similarly. If V_i is not free, then any variable that occurs in T_i may not be free and so do variables that share with T_i. Likewise, if T_i is not a variable or a variable that is not free, then V_i and any other variable that shares with V_i in θ may not bee free. Note that if V_i is free but T_i is not, then any free variables that occur in T_i will remain free after unifying θV_i with θT_i.

3.1 Correctness

Correctness (or safety) of our analysis can be established by proving the correctness of the functions *entry_substitution*, *success_substitution* and the following main theorem about *abstract_unify*. We provide only sketches of the proof.

Theorem 1
$\{mgu(\theta h, \sigma g)\} \subseteq \gamma(abstract\_unify(g, h, \alpha(\theta) \cup \alpha(\sigma)))$ where $dom(\theta) \cap dom(\sigma) = \emptyset$ $\square$

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

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

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

$$\alpha(\{mgu(\theta h, \sigma g)\}) \sqsubseteq abstract\_unify(g, h, \alpha(\theta) \cup \alpha(\sigma))$$

The above correctness condition can be stated in terms of the component domains and the partial ordering on them, as follows. (We use *LHS* and *RHS* to mean the left and right hand sides of the above equation.)

$$\begin{array}{lcl} LHS{\downarrow}Free & \supseteq & RHS{\downarrow}Free \\ LHS{\downarrow}Repeat & \subseteq & RHS{\downarrow}Repeat \\ LHS{\downarrow}Sharing & \subseteq & RHS{\downarrow}Sharing \end{array}$$

We provide below a sketch of the proofs of the above correctness condition.

1. For any θ, $ground\_variables(\theta) \cap Free(\theta) = \emptyset$
2. For any θ, $ground\_variables(\theta) \cap Repeat(\theta) = \emptyset$
3. For any θ, $ground\_variables(\theta) \cap vars(Sharing(\theta)) = \emptyset$
4. For any θ, it can be proved from the definition of *sharing* that the set of variables $dom(\theta) \setminus vars(sharing(\theta))$ are mapped to ground terms by θ. (Step two of *abstract_unify* function.)
5. Steps three and four of *abstract_unify* correctly derive the set of definitely ground variables. (See the discussion using bipartite graph representation of the unifier of the goal and the head.)
6. Conclude from the above steps [1-5] that *propagate_frs* is called with a correct refinement of the abstract substitution and the mgu.
7. Proof of correctness of the function *propagate_frs* is based on induction on the number of bindings in the mgu of the goal and the head and on induction on the structure of terms. We discussed the conservative derivation of potential sharing among and the repeatedness of variables based on the structure of terms and the repeated occurrences of variables in them.

3.2 Comparison with Other Approaches

As mentioned in the introduction, we designed and implemented a domain-independent abstract interpreter in Prolog that can be parametrized by abstract domains and operations on them. The details of MFG semantics underlying the interpreter and its efficient implementation will be reported elsewhere. We instantiated our interpreter with the domains and operations proposed by ourselves, Jacobs and Langen [13] and Jones and Sondergaard [15] in order to compare their precision and efficiency. In the following, we refer to these analyses as *SR, JL* and *JS* respectively. The following five programs were used to test the precision and efficiency of the three methods.

- *Grammar*: Generates and recognizes a set of English sentences. (15 clauses)
- *Bid*: Computes opening bid for a bridge hand. (51 clauses)

		No of Pairs of Shared Variables		
Program	**Size**	**SR**	**JS**	**JL**
Grammar	15	6	6	6
Bid	51	0	0	5
Deriv	61	0	0	0
Read	87	23	29	42
RdTok	54	37	41	60

Table 1: Precision of Analysis: Comparison of SR, JA and JS systems

- *Deriv*: Performs symbolic differentiation and simplifies the derivative. (61 clauses)
- *Read*: Public domain parser for Prolog by D.H.D Warren and Richard O'Keefe. (87 clauses)
- *Rdtok*: Public domain tokenizer for Prolog by Richard O'Keefe. (54 clauses)

These programs (except the first one) are of moderate size and the last two are used by many Prolog compilers. The precision of the analysis is measured in terms of the number of pairs of program variables reported to be *sharing* by each method for each of the benchmarks, in Table 1. Since the analyses are aimed at deriving *potentially sharing* variables, an analysis is more precise when it derives a smaller set of such variables. Our analyzer, SR, found the smallest number of pairs of sharing variables in all cases. JS was a close second. JL was not as precise as the other two. In the case of two widely used practical programs, the number of pairs of sharing variables derived by JL was larger by about 82% and 62% than that of SR and 45% and 46% larger than those of JS. We will see that, contrary to popular belief that more precise analyses are necessarily less practical or more time consuming, the improved precision of our analysis does actually contribute to faster execution.

We now compare the execution times of the three analyzers for the same programs, in table 2. These times represent median values over several runs. SR takes the least time to analyze all benchmark programs. JS, although almost as precise as ours, is three times slower in the case of *Read* and about six times slower in the case of *Rdtok* and *Grammar*. This is mainly due to the transitive closures computed by JS when approximating unification. Computing transitive closures is a central part of Jones and Sondergaard's technique for propagating sharing and can't be avoided. In our analysis of most programs, the operation *closure under union* (whose worst case time complexity is exponential in the number of elements in the set and thus worse than $\mathcal{O}(n^3)$ for transitive closure) is avoided most of the time because of the expressiveness of our abstract domain and the matching precision of our abstract domain operations. The fact that Jacobs and Langen's analysis was less precise than ours is also reflected in the analysis times. JL is slower by about three times in the case of *Rdtok* where its precision is the lowest and is slightly slower in the case of *Read*.

The analysis times of our abstract interpreter compare very favorably with those reported in the literature such as MA^3 and *Ms*[25]. For example, the MA^3 and *Ms* inter-

Program	Size	Analysis Times, in secs		
		SR	JS	JL
Grammar	15	1.7	11.9	1.7
Bid	51	5.4	6.3	5.4
Deriv	61	20.7	28.2	20.7
Read	87	55.2	183.6	57.0
Rdtok	54	66.7	404.9	178.1

Table 2: Analysis Times on a HP 9000/835, using Sicstus Prolog 0.6

preters took 60.2 seconds using Quintus Prolog-2.2 on a Sun3/50 and 68.3 seconds using SB-Prolog-2.3.2 on a Sun3/50 for analyzing *Read*, whereas ours took less time, 55.2 seconds using Sicstus-0.6 on a somewhat faster machine HP9000/835. There are, however, two points to note:

- The times taken by MA^3 and *Ms* [25] are for deriving modes for each procedure and thus collect information at one program point per procedure. Our analysis times are for deriving *sharing, freeness* and *groundness* information at *all program points*. In our approach, each clause has *n+1* program points, where *n* is the number of literals in the body.

- Our abstract interpreter, written in Prolog, interprets the program to be analyzed. The WAM code (Warren Abstract Machine [24]) of our abstract interpreter itself gets interpreted by the WAM emulator of Sicstus Prolog. Hence, our analyzer pays the overhead of double interpretation. The *Ms* interpreter transforms the programs to be analyzed into new programs [25] which get executed directly by the underlying Prolog system and thus avoids the overheads of double interpretation. Despite the double interpretation overheads, our abstract interpreter's analysis times compare very favorably.

4 Conclusions and Future Research

The feasibility of abstract interpretation based data flow analyses of logic programs has been established for simple mode analyses [25]. We showed that it is possible to define more *informative* and *efficient* analyses. We demonstrated that our proposed abstract domains and operations are more precise than other proposals and at the same time more efficient. It is clear from the analysis of some of the widely used Prolog programs that our extensions to the abstract domain proposed by Jacobs and Langen and our new abstract domain operations derive *significantly more precise information, more efficiently.* The precision of our analysis can be improved by keeping track of the position of repeated variable occurrences. But the feasibility of such an extension and the improvements in precision obtainable are to be explored.

References

[1] Apt, K. R., and Emden, M. V. Contributions to the theory of logic programming. *J.A.C.M 29* (1982), 841–862.

[2] Bruynooghe, M. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming 10*, 2 (February 1991), 91–124.

[3] Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B. Abstract interpretation: Toward the global optimization of prolog programs. In *Proceedings of the IEEE Symposium on Logic Programming* (1987), pp. 192–203.

[4] Bruynooghe, M., and Jenssen, G. An instance of abstract interpretation integrating type and mode inferencing. In *1988 Joint Conference on Logic Programming* (1988), pp. 684–699.

[5] Codish, M., and Yardeni, E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *Proceedings of the Eighth International Conference on Logic Programming* (1991).

[6] Conery, J. S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, MA, 1987.

[7] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice-theoretic model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of 4th POPL* (1977), ACM, pp. 238–252.

[8] Debray, S. K. Static analysis of parallel logic programs. In *1988 Joint Conference on Logic Programming* (1988), pp. 711–733.

[9] Debray, S. K. Static inference of modes and data dependencies. *ACM Transactions of Programming Languages and Systems 11*, 3 (July 1989), 418–450.

[10] Debray, S. K., and Warren, D. S. Automatic mode inference for prolog programs. In *Proceedings of the 1986 Symposium on Logic Programming* (1986).

[11] Hecht, M. S. *Flow Analysis of Computer Programs.* Programming Language Series. North-Holland, New York, 1977.

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

[13] Jacobs, D., and Langen, A. Accurate and efficient approximation of variable aliasing in logic programs. In *Proceedings of the North American Conference on Logic Programming* (1989), pp. 154–165.

[14] Jones, N., and Mycroft, A. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of 13th POPL* (1986), ACM.

[15] Jones, N., and Sondergaard, H. A semantics based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood Limited, 1987, ch. 6, pp. 123–142.

[16] Lloyd, J. W. *Foundations of Logic Programming*, first ed. Springer-Verlag, Berlin, 1984.

[17] Manilla, H., and Ukkonen, E. Flow analysis of prolog programs. In *4th IEEE Symposium on Logic Programming* (september 1987), IEEE Computer Society.

[18] Marriott, K., and Sondergaard, H. Bottom-up dataflow analysis of normal logic programs. In *5th International Conference on Logic Programming* (August 1988).

[19] Mellish, C. S. Abstract interpretation of prolog programs. In *Third International Conference on Logic Programming* (july 1986), pp. 463–475.

[20] Muthukumar, K., and Hermenegildo, M. Determination of variable dependence information through abstract interpretation. In *Proceedings of the North American Conference on Logic Programming* (1989), pp. 166–185.

[21] Muthukumar, K., and Hermenegildo, M. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming* (1991).

[22] Sondergaard, H. An application of abstract interpretation of logic programs: Occur check reduction. In *Proccedings of European Symposium on Programming* (1986), Springer-Verlag.

[23] Sundararajan, R. An introduction to abstract interpretation of logic programs. Tech. Rep. CIS-TR 91-05, Dept of Computer and Information Science, University of Oregon, Eugene, Oregon, U.S.A. 97403, 1991.

[24] Warren, D. H. D. An abstract prolog instruction set. Tech. Rep. 309, SRI, Menlo Park, CA, 1983.

[25] Warren, D. H. D., Hermenegildo, M., and Debray, S. K. On the practicality of global flow analysis of logic programs. In *1988 Joint Conference on Logic Programming* (1988), pp. 684–699.

[26] Winsborough, W. *Automatic, Transparent Paralellization of Logic Programs at Compile Time.* PhD thesis, Dept. of Computer Science, University of Wisconsin-Madison, Sept. 1988.