Efficient and Precise Sharing Domains for Logic Programs

Christian Fecht
Universität des Saarlandes
Postfach 151150
66041 Saarbrücken
Tel.: +49–681–302–5573
Fax : +49–681–302–3065
fecht@cs.uni-sb.de

Abstract

Sharing information between logical variables is crucial for a lot of analyses of logic programs, e.g., freeness analysis, detection of And-parallelism, and occur-check. Therefore, the development of accurate sharing domains has attracted a lot of research. The sharing domain **JL** of Jacobs/Langen, which represents substitutions by powersets of variables, is considered one of the most precise sharing domains. However, it is too inefficient in practice; lots of programs cannot be analyzed in reasonable time. Improvements of **JL**, by adding auxiliary information like linearity, suffer from the same inefficiency, too. To improve upon this situation, we systematically derived a new sharing domain **↓JL** from **JL** which represents variables by downward closed powersets of variables. We combined **↓JL** with the groundness domain **POS**. Both **JL** and the new domain **↓JL+POS** have been implemented with the help of the Prolog analyzer generator **GENA**. In order to study the impact of linearity, we also implemented the abstract domains **JL+LIN** and **↓JL+POS+LIN**. The new domains are much more efficient as their counterparts **JL** and **JL+LIN**, respectively. Even more important, they can analyze even largest real-world programs in reasonable time. Surprisingly, the new sharing domains seem to have the same precision than **JL** and **JL+LIN** in practice.

1 Introduction

Sharing analysis of logic programs aims at computing independence information for variables. Two variables X and Y are *independent* in a given substitution ϑ if instantiating X does not instantiate Y and vice versa. Instead of computing independence directly, the complementary information — *possible sharing* — is computed. Two variables X and Y share in a substitution ϑ if they are bound to terms containing at least one common variable, that is $vars(X\vartheta) \cap vars(Y\vartheta) \neq \emptyset$. Independence is important for the following reasons:

And-Parallelism An And-parallel Prolog implementation [15, 6] would try to solve a sequence (A, B) of goals by solving the goals A and B in parallel. To do this efficiently, the goals A and B have to be independent. Definite independence information can be statically computed by means of a sharing analysis [23, 16].

Freeness For a Prolog compiler the most useful information about a variable at a particular program point is that it is always bound or always free [28, 26]. Freeness is not closed under instantiation. Thus, a variable X may lose its freeness, because another variable is instantiated. In order to compute non-trivial and correct freeness information, one has to know about possible sharing between variables.

Occur-Check If a variable X is unified with a term t, one has to test whether X occurs in t. If this is the case, then unification fails. This test is commonly referred to as *occur-check* and is commonly omitted in Prolog implementations for the sake of efficiency. The missing occur-check corrupts the logical soundness of Prolog. Occur-check reduction aims at detecting unifications where the occur-check can be omitted safely. [11] describes an occur-check reduction algorithm based on sharing information.

Because of the importance of available sharing information, the development of efficient and accurate sharing analyses [12, 18, 23, 25] for logic programs has attracted a lot of research. Most of these analyses are based on the technique of abstract interpretation [8, 9]. The actual analyzers can be split into a generic abstract interpreter and an abstract domain which provides a lattice of abstract substitutions, an abstract unification procedure, and functions for abstract procedure entry and exit.

The well-known sharing domain **JL** of Jacobs and Langen [16] is considered one of the most precise sharing domains. It describes concrete substitutions by sets of sets of variables. The domain **JL** does not only express sharing, but is also able to express ground dependencies between variables. Unfortunately, **JL** is inherently inefficient. Its abstract substitutions can be exponentially large. Abstract unification is done by means of a closure operator whose computation has exponential worst-case complexity. Implementations of **JL** (section 6, [17, 24]) indicate that it is not feasible in practice, since a lot of (even small) programs can not be analyzed within reasonable time. **JL** has been enlarged with additional information, for instance freeness [22] and linearity [5, 19, 25]. Linearity helps to infer more accurate sharing information. Additionally, available linearity information keeps the sets of sets of variables small by avoiding the expensive closure operator whenever it is possible. Consequently, the abstract domain **JL+LIN** is often dramatically faster than **JL**[5]. However, these enriched sharing domains are also not able to analyze a lot of programs (section 7, [24]). Apparently, the additional information is either not present in the concrete computation, or cannot be inferred by the abstract interpreter.

Our contributions are threefold:

- We systematically derive a new sharing domain from **JL**. The new domain $\downarrow$**JL** is obtained by only considering downward closed powersets of variables which can be efficiently represented by their maximal elements. Abstract unification in $\downarrow$**JL** is defined on such representations and its soundness is proven. It is shown that $\downarrow$**JL** represents sharing and definite groundness, but no ground dependencies. Therefore, we add the groundness domain **POS** [20, 1, 27] to $\downarrow$**JL** yielding the new sharing domain $\downarrow$**JL+POS**.
- We implemented and **JL** and $\downarrow$**JL+POS** with the help of the Prolog analyzer generator GENA [14]. The generated analyzers were run on a large set of benchmark programs. The new domain $\downarrow$**JL+POS** is much more efficient than **JL**. Even more important, it can analyze even largest real-world programs in reasonable time. Surprisingly, the new sharing domain seems to have almost the same precision than **JL** in practice. There is only one program in our benchmark suite where **JL** computes better sharing information.
- We also implemented the abstract domains **JL+LIN** and $\downarrow$**JL+POS+LIN** in order to study the impact of linearity on the precision and the efficiency of the analysis. Whereas **JL+LIN** cannot analyze many (even some small) programs, $\downarrow$**JL+POS+LIN** can analyze *all* programs and is substantially faster than **JL+LIN** in most cases. Furthermore, $\downarrow$**JL+POS+LIN** seems to have the same precision than **JL+LIN** in practice.

The overall structure of our paper is as follows. In section 2 we recall fundamental notions and basic definitions of the theory of abstract interpretation of logic programs. The groundness domains **POS** and **DEF** are briefly described. Section 3 gives a detailed description of the sharing domain **JL** of Jacobs/Langen and recalls some of its important properties. In section 4 we first derive abstract domain $\downarrow$**JL** from **JL** by only allowing downward closed powersets as abstract substitutions. We show how abstract unification can be directly expressed on representations of downward closed sets. Furthermore, the expressive power of $\downarrow$**JL** is investigated. By adding groundness domain **POS** to $\downarrow$**JL** we obtain our new sharing domain $\downarrow$**JL+POS**. Section 5 theoretically compares $\downarrow$**JL+POS** with **JL**. Section 6 presents our implementations of **JL** and $\downarrow$**JL+POS**. For a large set of benchmark programs, we present the analysis time and precision of our sharing analyzers based on **JL** and $\downarrow$**JL+POS**. Section 7 discusses the impact of linearity and presents the abstract domains **JL+LIN** and $\downarrow$**JL+POS+LIN**. Finally, we summarize the results from our practical experiments and conclude.

2 Preliminaries

Let $\mathbf{V}$ be a nonempty, finite set of *variables*. Terms are defined as usual. A *substitution* ϑ is an almost identity mapping from variables to terms. A substitution ϑ is *idempotent* iff $\vartheta\vartheta = \vartheta$ holds. We denote the set of idempotent substitutions by **Subst**. A substitution ϕ is an *instance* of ϑ, $\phi \trianglelefteq \vartheta$, if there exists a substitution ψ with $\phi = \vartheta\psi$. A variable X is *ground* with respect to substitution $\vartheta \in$ **Subst** if $vars(X\vartheta) = \emptyset$. Two variables X and Y *share* with respect to substitution ϑ if $vars(X\vartheta) \cap vars(Y\vartheta) \neq \emptyset$.

Definition 1 (Definite Groundness)
$ground : \mathcal{P}(\mathbf{Subst}) \to \mathcal{P}(\mathbf{V})$
$ground(\Theta) = \bigcap_{\vartheta \in \Theta} \{X \in \mathbf{V} \mid vars(X\vartheta) = \emptyset\}$

Definition 2 (Possible Sharing)
$share : \mathcal{P}(\mathbf{Subst}) \to \{\mathcal{R} \subseteq \mathbf{V} \times \mathbf{V} \mid \mathcal{R} \text{ sym. and irrefl.}\}$
$share(\vartheta) = \bigcup_{\vartheta \in \Theta} \{(X, Y) \mid vars(X\vartheta) \cap vars(Y\vartheta) \neq \emptyset\}$

Note that a variable X which is ground with respect to a substitution ϑ cannot share with any other variable. Therefore, any reasonable sharing analysis should infer groundness as well.

Abstract interpretation [8, 9] formalizes program analysis as approximate computation. Instead of executing a program with data, it is executed with *descriptions* of the data. In our case, this means that logic programs are executed with *abstract substitutions* instead of concrete substitutions. Concrete and abstract substitutions are related by a *description relation*. To every operation on the concrete substitutions, there must exist a corresponding abstract operation mimicking the concrete operation on the abstract substitutions. A *generic abstract interpreter* [4] is an abstract interpreter which is parametrized over an *abstract domain*. An abstract domain consists of a complete lattice $(Asub, \leq, \bot, \top, \sqcup)$ of abstract substitutions and the abstract versions of the concrete operations. The description relation between concrete and abstract substitutions is often given by an *abstraction function* $\alpha : \mathcal{P}(\mathbf{Subst}) \to Asub$ or by a *concretization function* $\gamma : Asub \to \mathcal{P}(\mathbf{Subst})$. In the case of logic programs, the abstract domain must further provide an abstract unification procedure $aunify : Asub \times \mathbf{V} \times Term \to Asub$, and functions for abstract procedure entry and exit.

An *assignment* is a mapping $\sigma : \mathbf{V} \to \{0,1\}$ from variables to truth values. A *Boolean function* $f : (\mathbf{V} \to \{0,1\}) \to \{0,1\}$ maps assignments to truth values. An assignment σ is a *model* of the Boolean function f if $f(\sigma) = 1$ holds. A Boolean function f is *positive* if $f(\mathbf{1}) = 1$ where $\mathbf{1}$ is the assignment that maps each variable to truth value 1. A Boolean function is *definite* if the set of all its models is closed under intersection. Boolean functions can be ordered by the implication order. The resulting poset is a complete lattice. The ground variables of substitution ϑ can be represented by an assignment $[\vartheta]$.

$$[\vartheta](X) = \begin{cases} 1 & , X \text{ is ground with respect to } \vartheta \\ 0 & , \text{otherwise} \end{cases}$$

Definition 3 *A Boolean function f describes a substitution $\vartheta \in \mathbf{Subst}$ if $f([\phi]) = 1$ for all $\phi \trianglelefteq \vartheta$.*

The groundness domains **POS** and **DEF** [1] consist of the complete lattice of positive Boolean functions and definite Boolean functions, respectively, with the above description relation.

3 The Sharing Domain of Jacobs/Langen

The sharing domain **JL** of Jacobs and Langen [16] represents substitutions by sets of sets of variables, such as $\{\emptyset, \{X\}, \{X,Y\}, \{U,Y,Z\}\}$. The key notion in the definition of **JL** is that of *occurrence*.

Definition 4
$occ : \mathbf{Subst} \times \mathbf{V} \to \mathcal{P}(\mathbf{V})$
$occ(\vartheta, X) = \{Y \in \mathbf{V} \mid X \in vars(Y\vartheta)\}$

Definition 5 *The abstract domain* **JL***:*
$lattice : Asub_{\mathrm{JL}} = \{S \in \mathcal{P}(\mathcal{P}(\mathbf{V})) \mid S \neq \emptyset \Rightarrow \emptyset \in S\}$
$order : \sqsubseteq_{\mathrm{JL}} = \subseteq$
$lub : \sqcup_{\mathrm{JL}} = \cup$
$abs : \alpha_{\mathrm{JL}}(\Theta) = \bigsqcup_{\vartheta \in \Theta} \{occ(\vartheta, X) \mid X \in \mathbf{V}\}$

Example 1 *Let $\mathbf{V} = \{U,V,W,X,Y,Z\}$ be the variables of interest and $\vartheta = \{U/a, X/g(Y,V), Z/f(W,Y)\}$.*

$occ(\vartheta, U) = \emptyset$ $\quad$ $occ(\vartheta, V) = \{V, X\}$
$occ(\vartheta, W) = \{W, Z\}$ $\quad$ $occ(\vartheta, X) = \emptyset$
$occ(\vartheta, Y) = \{X, Y, Z\}$ $\quad$ $occ(\vartheta, Z) = \emptyset$

Thus, $\alpha_{\mathrm{JL}}(\vartheta) = \{\emptyset, \{V,X\}, \{W,Z\}, \{X,Y,Z\}\}$. □

The elements of an abstract substitution $S \in Asub_{\mathrm{JL}}$ are called *sharing groups* and are themselves sets of variables. It is well known that an abstract substitution $S \in Asub_{\mathrm{JL}}$ expresses the following information.

Sharing Variables X and Y share in any substitution ϑ described by S if there exists a sharing group $A \in S$ with $X, Y \in A$. More formally,

$$share(\gamma_{\mathrm{JL}}(S)) = \{(X,Y) \mid X \neq Y, \exists A \in S : X, Y \in A\}$$

Groundness Variable X is definitively ground in any substitution described by S if X does not occur in any sharing group of S, i.e., $X \notin vars(S)$. More formally,

$$ground(\gamma_{\mathrm{JL}}(S)) = \mathbf{V} - vars(S)$$

Ground Dependencies In [7] it is shown that S expresses ground dependencies which coincide with the definite Boolean function $def_{\mathrm{JL}}(S)$ defined as

$$def_{\mathrm{JL}}(S) = \wedge \{\wedge W_1 \to \wedge W_2 \mid \forall M \in S. \\ (W_2 \cap M \neq \emptyset) \Rightarrow (W_1 \cap M \neq \emptyset)\}$$

Example 2 *Let $\mathbf{V} = \{U,W,X,Y,Z\}$ be the variables of interest. Consider the abstract substitution $S = \{\emptyset, \{X\}, \{X,Y\}, \{Y,Z\}, \{U,X,Z\}\}$. The following information is expressed by S:*

sharing pairs	$\{(X,Y), (Y,Z), (U,X), (X,Z)\}$
ground variables	$\{W\}$
dependencies	$W \wedge (X \to U) \wedge (Z \to U)$ $\wedge\ (U \wedge Y \to Z) \wedge (X \wedge Y \to Z)$ $\wedge\ (X \wedge Z \to Y)$

□

Abstract Unification

We need some additional definitions in order to define the abstract unification function $aunify_{\mathrm{JL}} : Asub \times \mathbf{V} \times Term \to Asub$.

$$S_1 \bowtie S_2 = \{A \cup B \mid A \in S_1,\ B \in S_2\}$$
$$S_{|M} = \{A \in S \mid A \cap M \neq \emptyset\}$$

Definition 6 *A set $S \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$ is* closed under union *if $A \in S$ and $B \in S$ implies $A \cup B \in S$. The least superset of S that is closed under union is denoted by S^*.*

Abstract unification $aunify_{\mathrm{JL}}(S, X, t)$ is defined by following three cases.

Case 1: $X \notin vars(S)$
X is ground. After the successful unification all variables occurring in t must be ground, too. Therefore, we remove all sharing groups from S that contain at least one variable from t.
$\Rightarrow aunify_{\mathrm{JL}}(S, X, t) = S - \{A \in S \mid A \cap vars(t) \neq \emptyset\}$.

Case 2: $vars(t) \cap vars(S) = \emptyset$
All variables in t are ground. After the successful unification X must be ground, too. Therefore, we remove from S all sharing groups that contain X.
$\Rightarrow aunify_{\mathrm{JL}}(S, X, t) = S - \{A \in S \mid X \in A\}$

Case 3: Neither $X \notin vars(S)$, nor $vars(t) \cap vars(S) = \emptyset$
$\Rightarrow aunify_{\mathrm{JL}}(S, X, t) = (S - S_{|\{X\} \cup vars(t)}) \cup (S_{|\{X\}})^* \bowtie (S_{|vars(t)})^*$.

The first two cases deal with the propagation of groundness. Note that a ground variable cannot share with any other variable. If variable X becomes ground through unification, all sharing groups containing X can be safely removed from S. Case 3 is the most complicated one. The sharing groups which contain a variable occurring in the unification equation $X = t$ are replaced by the pairwise union of the closure of all groups sharing with the left-hand side and the closure of all groups sharing with right-hand side of the equation.

Example 3 *Consider* $S = \{\emptyset, \{U\}, \{X\}, \{Y\}, \{X, Y\}, \{Y, Z\}, \{U, Z\}\}$ *and the unification equation* $X = f(Y, Z)$. *Let us compute* $aunify_{\mathrm{JL}}(S, X, f(Y, Z))$.

$$\begin{aligned} S_{|\{X\}} &= \{\{X\}, \{X, Y\}\} \\ (S_{|\{X\}})^* &= \{\{X\}, \{X, Y\}\} \\ S_{|\{Y,Z\}} &= \{\{Y\}, \{X, Y\}, \{Y, Z\}, \{U, Z\}\} \\ (S_{|\{Y,Z\}})^* &= \{\{Y\}, \{X, Y\}, \{Y, Z\}\{U, Z\}, \{X, Y, Z\}, \\ &\quad \{U, Y, Z\}, \{U, X, Y, Z\}\} \end{aligned}$$

Thus, $(S_{|\{X\}})^* \bowtie (S_{|vars(t)})^* = \{\{X, Y\}, \{X, Y, Z\}, \{U, X, Z\}, \{U, X, Y, Z\}\}$ *and the result of the unification is* $aunify_{\mathrm{JL}}(S, X, f(Y, Z)) = \{\emptyset, \{U\}, \{X, Y\}, \{X, Y, Z\}, \{U, X, Z\}, \{U, X, Y, Z\}\}$. □

To summarize, **JL** accurately models sharing and ground dependencies. However, it is too inefficient in practice (see section 6). The abstract unification procedure is very inefficient. The computation of the closure S^* has exponential worst-case complexity. Application of the closure operator and the subsequent pairwise union rapidly lead to combinatorial explosion.

4 The Abstract Domains ↓JL and ↓JL+POS

One of the nice features of abstract interpretation is that one can always replace a description d by a greater description d', e.g., $d \sqsubseteq d'$. By doing so, the overall result of the abstract operations may be less precise, but it will always be correct. The correctness of the final result follows from the monotonicity of the description relation and the monotonicity of the abstract operations. This idea is utilized, for instance, in the technique of *widening* [8, 10] which speeds up fixpoint iterations or even forces possibly infinite iterations to terminate. Another application of the replacement idea would be to replace a description d whose computer representation is to big by a greater description with a more efficient representation.

Let us apply this idea in order to cope with the inherent inefficiency of abstract domain **JL**. Thus, we replace an $S \in Asub_{\mathrm{JL}}$ by an S' with $S \subseteq S'$. Unfortunately, S' contains more sharing groups than S. If we represent power sets of variables explicitly by listing their elements, then the representation of S' is larger than the representation of S. Thus, replacing S by S' does not only result in a loss of precision, but also makes the abstract interpretation even more inefficient. Nevertheless, we can apply our idea if we ensure that S' is always chosen in such a way that it allows for a compact representation. Good candidates are those sets which are *downward closed*.

Definition 7 *A set $S \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$ is* downward closed *iff the following holds: $M \in S$ implies $N \in S$ for each $N \subseteq M$.*

Definition 8 *The* downward closure $\downarrow S$ *of a powerset $S \in P(\mathcal{P}(\mathbf{V}))$ is defined by $\downarrow S = \{N \subseteq \mathbf{V} \mid \exists M \in S$ with $N \subseteq M\}$.*

A set $S \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$ is *represented* by a set $E \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$ iff $S = \downarrow E$. It is well-known that downward closed sets are represented by its maximal elements as the following lemma states.

Lemma 1 *If $S \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$ is downward closed, then S can be represented by its maximal elements: $S = \downarrow\{M \in S \mid M$ is maximal in $S\}$.*

Example 4 *Consider the downward closed set $S = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{W\}, \{X, Y\}, \{X, Z\}, \{Y, Z\}, \{X, W\}, \{X, Y, Z\}\}$. Set S is represented by its maximal elements $\{\{X, Y, Z\}, \{X, W\}\}$. Note that $S = \downarrow\{\{X, Y, Z\}, \{X, W\}\}$.* □

As explained above, an abstract substitution $S \in Asub_{\mathrm{JL}}$ expresses sharing information, definite groundness, and ground dependencies. What is lost when S is replaced by its downward closure $\downarrow S$?

Lemma 2 *1.* $share_{\mathrm{JL}}(\downarrow S) = share_{\mathrm{JL}}(S)$

2. $ground_{\mathrm{JL}}(\downarrow S) = ground_{\mathrm{JL}}(S)$

3. $def_{\mathrm{JL}}(\downarrow S) = \bigwedge ground_{\mathrm{JL}}(S)$

By taking the downward closure, only the ground dependencies are lost. The sharing and definite groundness information are not changed. Note, that the union of downward closed sets is downward closed, too. A new sharing domain $\downarrow$**JL** can thus be obtained from **JL** by restricting the set of abstract substitutions to downward closed powersets of variables. The abstract unification in $\downarrow$**JL** is the downward closure $\downarrow aunify_{\mathrm{JL}}(S, X, t))$ of the result of the abstract unification in **JL**.

Definition 9 *The abstract domain* $\downarrow$**JL***:*

$$
\begin{array}{ll}
Asub_{\downarrow\mathrm{JL}} & = \{X \in \mathcal{P}(\mathcal{P}(\mathbf{V})) \mid S \text{ is downward closed}\} \\
\sqsubseteq_{\downarrow\mathrm{JL}} & = \subseteq \\
\sqcup_{\downarrow\mathrm{JL}} & = \cup \\
\alpha_{\downarrow\mathrm{JL}}(\Theta) & = \downarrow \bigcup_{\vartheta \in \Theta} \{occ(\vartheta, X) \mid X \in \mathbf{V}\} \\
aunify_{\downarrow\mathrm{JL}}(S, X, t) & = \downarrow aunify_{\mathrm{JL}}(S, X, t)
\end{array}
$$

The soundness of the abstract unification in $\downarrow$**JL** follows from the soundness of the abstract unification in **JL** and from $aunify_{\mathrm{JL}}(S, X, t) \subseteq aunify_{\downarrow\mathrm{JL}}(S, X, t)$ for all downward closed $S \in \mathcal{P}(\mathcal{P}(\mathbf{V}))$, $X \in \mathbf{V}$, and terms t.

So far, nothing has been gained since abstract unification in $\downarrow$**JL** is defined in terms of abstract interpretation in **JL**. Since a downward closed powerset S is represented by a powerset E with $S = \downarrow E$, we will next define abstract unification directly on the representation E instead of $\downarrow E$. To this end, we make the following simple observations:

1. The powerset $\downarrow E$ contains the same variables as its representation E, i.e., $vars(E) = vars(\downarrow E)$.
2. Removing all sets from $\downarrow E$ which have a common element with set G is done on the representation by removing G from each set in E, i.e., $\downarrow\{A \in \downarrow E \mid A \cap G = \emptyset\} = \downarrow\{A - G \mid A \in E\}$.
3. The representation of a powerset $(\downarrow E)^*$ which is closed under union is simply the union of all sets in E, i.e., $\downarrow(\downarrow E)^* = \downarrow(\bigcup E)$. Note that the representation of $(\downarrow E)^*$ is a singleton.
4. $\downarrow((\downarrow E_1) \bowtie (\downarrow E_2)) = \downarrow(E_1 \cup E_2)$.

It should now be relatively easy to define the abstract unification $aunify_{\downarrow\mathrm{JL}}(\downarrow E, X, t)$ on the representation E directly. We consider the following three cases.

Case 1: $X \notin vars(E)$
Since $vars(\downarrow E) = vars(E)$, variable X is ground and, therefore all variables in t have to be ground, too. We remove all sharing groups from $\downarrow E$ which contain a variable from $vars(t)$. On the representation, this is achieved by deleting all variables in $vars(t)$ from E.
$\Rightarrow$ $aunify_{\downarrow\mathrm{JL}}(\downarrow E, X, t) = \downarrow\{A - vars(t) \mid A \in E\}$

Case 2: $vars(t) \cap vars(E) = \emptyset$
By the same argument as above, we conclude that X has to be ground. Therefore X is removed from the representation E.
$\Rightarrow$ $aunify_{\downarrow\mathrm{JL}}(\downarrow E, X, t) = \downarrow\{A - \{X\} \mid A \in E\}$

Case 3: Neither $X \notin vars(E)$ nor $vars(t) \cap vars(E) = \emptyset$
$\Rightarrow$ $aunify_{\downarrow\mathrm{JL}}(\downarrow E, X, t) = \downarrow((E - E_{|\{X\} \cup vars(t)}) \cup \bigcup E_{|\{X\} \cup vars(t)})$

The third case which actually deals with sharing has been greatly simplified. Instead of computing closures under union and pairwise unions of powersets, all sharing groups of the representation which contain a variable in $vars(X = t)$ are replaced by their union.

Theorem 1 $aunify_{\downarrow\mathrm{JL}}(\downarrow E, X, t) = \downarrow aunify_{\mathrm{JL}}(\downarrow E, X, t)$.

In contrast to **JL** abstract domain $\downarrow$**JL** does not model ground dependencies between variables. Hence, $\downarrow$**JL** may detect less ground variables than **JL**. Since no ground variable can share with any other variable, $\downarrow$**JL** may detect much more sharing pairs than **JL**. In order to compensate this loss of precision, we add the groundness domain **POS** [20, 1, 27] to $\downarrow$**JL**. Although groundness analysis with **POS** has to solve a theoretically intractable problem, implementations of **POS** are amazingly fast in practice. The reduced product [9] of $\downarrow$**JL** and **POS** is denoted $\downarrow$**JL+POS**. Abstract domain **POS** will never infer less ground variables than **DEF**. As a consequence, $\downarrow$**JL+POS** is more precise with respect to groundness than **JL**.

Abstract unification in $\downarrow$**JL+POS** is similar to abstract unification in $\downarrow$**JL**. The main difference is that ground variables are not inferred from the sharing component, but from the **POS**-component. Principally, abstract unification is independently done both in the sharing- and **POS**-component of the abstract substitution, and then the resulting pair is reduced. Reduction amounts to assuring that variables that are ground according to the **POS**-component do not occur in any sharing group in the sharing component of the abstract substitution.

5 Comparison of JL and $\downarrow$JL+POS

As shown above, $\downarrow$**JL** does not express ground dependencies between variables. To remedy this, we have added the groundness domain **POS** to $\downarrow$**JL** yielding the abstract domain $\downarrow$**JL+POS**. Since there are programs for which **POS** computes strictly better groundness information than **DEF**, which is a part of **JL**, there are

also programs where $\downarrow$**JL+POS** computes strictly better sharing information than **JL**.

Example 5 *Consider the following logic program:*

```
main :- p(X, Y), X = Y.
p(a, _).
p(_, b).
```

POS *infers that both X and Y are ground after the unification $X = Y$. Thus,* $\downarrow$**JL+POS** *infers that X and Y are independent at the exit point of* main/0*. To the contrast,* **DEF** *is not able to infer the groundness of X and Y. Hence, an analysis with* **JL** *yields that X and Y possibly share at the exit point of* main/0*.* □

Besides definite groundness and ground dependencies, **JL** does express a further property which is useful for computing better sharing information. This property (*strong coupling between sharing pairs*) is illustrated by the following example.

Example 6 *Let $\mathbf{V} = \{X, Y, Z\}$ be the variables of interest. Consider the following abstract substitutions in* **JL***.*

$$\begin{aligned} S_1 &= \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{X, Y, Z\}\} \\ S_2 &= \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{X, Y\}, \{Y, Z\}\} \end{aligned}$$

Both S_1 and S_2 express the same sharing information ($\{(X, Y), (X, Z), (Y, Z)\}$) and the same ground dependencies ($1 \in$ **DEF***). Note that S_2 is downward closed and thus $S_2 \in \downarrow$**JL**. Now consider the unification $X = a$. Grounding X in S_1 yields $\{\emptyset, \{Y\}, \{Z\}\}$. Thus, there is no sharing between Y and Z. Although variable X does not occur in the sharing pair (Y, Z), the grounding of X has removed the sharing pair (Y, Z). The sharing pairs (X, Y), (X, Z), and (Y, Z) are* strongly coupled *in S_1. Grounding one of $\{X, Y, Z\}$ will remove all sharing between these variables. However, grounding X in S_2 yields $\{\emptyset, \{Y\}, \{Z\}, \{Y, Z\}\}$. Thus, there is still a possible sharing between Y and Z.* □

This example shows that there are programs where **JL** computes strictly better sharing information than $\downarrow$**JL+POS**. Summarizing, **JL** and $\downarrow$**JL+POS** are uncomparable. In practice, however, **JL** is at least as accurate as $\downarrow$**JL**.

6 Implementation and Experimental Evaluation

In order to practically compare efficiency and precision of **JL** and $\downarrow$**JL+POS** we implemented analyzers based on these domains and run them on a large set of Prolog programs. The implementation work has been greatly simplified by the use of the Prolog analyzer generator GENA [14] which is implemented itself in SML. The fixpoint algorithm we used for the benchmarks is the generic and general algorithm **WRT** from [13]. This algorithm computes a part of the precise abstract input-output semantics $[\![P]\!] : Pred \times Asub \to Asub$.

Sets of variables are implemented by lists of integers where each integer is used as a bit vector. Since most clauses have less than 31 variables, this representation is very compact and efficient. Powersets of variables are implemented as balanced binary trees over sets of variables. Downward closed powersets are canonically represented as the list of their maximal elements. At last, Boolean functions are implemented by binary decision diagrams [3, 2, 27], the state-of-the-art technique for representing and manipulating Boolean functions in computers.

Figure 1 shows the results of our experiments. The measurements were done on a Sun 20 with 64MB main memory. We used SML of New Jersey version 109. The analysis times include system and garbage collection time and are the average of five runs. Our analyzer computes call modes for the predicates in the program. A call mode for a predicate is a statement about its arguments. The precision of a call mode measured as the number of sharing pairs between arguments. The precision in figure 1 is the sum of the precisions of the call modes for all predicates in the program. Recall that the less the number of sharing pairs, the higher is the precision.

Some of the programs in our benchmark suite are large real-word applications: *aqua-c* is the complete source code of Peter Van Roy's Aquarius Prolog compiler (about 16000 lines of code), *b2* is a large mathematical application (about 2000 lines of code), *chat* is D.H.D. Warrens chat-80 system (about 5000 lines of code), and *read* and *readq* are Prolog readers. The other programs were either taken from the benchmark suite of Aquarius Prolog or from the benchmark suites of the GAIA [4] and PLAIA [23] systems.

The following observations can be made from the numbers of figure 1:

1. Programs *action*, *aqua-c*, *chat*, *chat-parser*, *reducer* and *sdda* could not be analyzed by **JL** within one hour. Even worse, none of them could be ever analyzed completely. For instance, the analysis of *chat-parser* had to been interrupted after seven (!) days. The inability of **JL** to analyze these programs is not due to the actual size of the programs (*reducer* and *sdda* are smal programs each having about 300 lines of code), but to the large number of variables in some of their clauses.

2. *All* programs can be analyzed with $\downarrow$**JL+POS**.

Programm	efficiency			precision	
	↓JL+POS	**JL**	*ratio*	**↓JL+POS**	**JL**
action.pl	11.76	∞	∞	41	?
ann.pl	0.51	14.32	28.3	42	42
aqua-c.pl	212.07	∞	∞	3879	?
b2.pl	1.61	48.83	30.3	48	48
boyer.pl	0.21	1.33	6.5	20	20
browse.pl	0.10	5.35	51.5	6	6
chat.pl	21.95	∞	∞	1152	?
chat-parser.pl	7.85	∞	∞	439	?
chess.pl	0.35	33.83	95.6	19	18
circuit.pl	0.05	0.05	0.9	0	0
cs.pl	0.22	1.92	8.9	2	2
flatten.pl	0.31	15.56	49.6	32	32
gabriel.pl	0.14	0.26	1.8	11	11
life.pl	0.08	0.11	1.4	1	1
nand.pl	0.80	2.99	3.7	0	0
peep.pl	0.19	0.14	0.7	0	0
press.pl	0.85	13.65	16.0	50	50
read.pl	0.65	4.62	7.2	30	30
readq.pl	2.28	103.89	45.6	100	100
reducer.pl	0.51	∞	∞	104	?
scc.pl	0.35	0.31	0.9	0	0
sdda.pl	0.42	∞	∞	53	?
sendmore.pl	0.21	0.29	1.4	1	1
serialise.pl	0.11	2.72	25.2	10	10
triangle.pl	0.16	0.10	0.6	0	0
wgc.pl	0.04	0.03	0.7	1	1

Table 1: Experimental Evaluation of **↓JL+POS** and **JL**

Most analysis times are very small or at least moderate. Even the huge *aqua-c* could be analyzed within reasonable time. Programs *reducer* and *sdda* which **JL** was unable to analyze are analyzed by **↓JL+POS** within half a second. On most programs, **↓JL+POS** is dramatically faster than **JL**.

3. For most programs, **↓JL+POS** infers the same number of sharing pairs than **JL**. Indeed, there is only one program (*chess*) where **↓JL+POS** is less precise and infers one additional sharing pair. This loss of precision is due to inability of **↓JL+POS** to express strong coupling between sharing pairs. Thus, **↓JL+POS** seems to have almost the same precision than **JL** in practice.

7 The Impact of Linearity

A variable X is *linear* in a substitution ϑ if no variable occurs more than once in $X\vartheta$. Definite linearity information is useful for computing accurate sharing information [25, 19, 5].

Example 7 *Consider the unification $X = f(Y, Z)$ and assume that X, Y, and Z are all nonground and independent before the unification. If X is bound to a linear term, then Y and Z are still independent after the unification. However, if X is bound to a nonlinear term, variables Y and Z may possibly share after the unification, e.g., consider the case where X is bound to the nonlinear term $f(A, A)$.* □

Linearity does not only help to compute better sharing information. It heavily affects the efficiency of the sharing analysis. In **JL**, linearity information helps to avoid the expensive closure under union operation [25]. Not computing the closure under union keeps the sets of sets of variables small which additionally speeds up subsequent computations. We also implemented the abstract domains **JL+LIN** and **↓JL+POS+LIN**based on the description in [25]. The results of our experiments are shown in figure 7.

1. The abstract domains with linearity **JL+LIN** and **↓JL+POS+LIN** often compute substantially better sharing information than **JL** and **↓JL+POS**, respectively.

2. For some programs, **JL+LIN** is dramatically faster than **JL**. However, there are some programs (*action*, *aqua-c*, *chat*, *chat-parser*, *reducer*, *sdda*) which could not be analyzed with **JL+LIN**. We had to interrupt the analyzer after one hour.

3. **↓JL+POS+LIN** can analyze *all* program in reasonable time. Most programs are analyzed very fast. For all our programs, **↓JL+POS+LIN** has the same precision as **JL+LIN**.

8 Related Work and Conclusion

We have presented two new domains for sharing analysis of logic programs. The new domains **↓JL+POS** and **↓JL+POS+LIN** have been derived systematically from the sharing domains **JL** and **JL+LIN**. As a consequence, the soundness of the abstract unification procedure was easily established. The new domains were experimentally evaluated on a large set of Prolog programs. In practice, they seem to have almost the same precision than the domains **JL** and **JL+LIN**, respectively. More important, the new domains are substantially more efficient.

In contrast to previous work on sharing analysis [23, 23, 19, 25, 5, 21], we evaluated our analyzers on a large set of Prolog programs, including large real-word programs which are hard to analyze. Our sharing analyzers

Programm	efficiency			precision	
	↓**JL+POS +LIN**	**JL+LIN**	*ratio*	↓**JL+POS +LIN**	**JL+LIN**
action.pl	32.12	∞	∞	37	?
ann.pl	0.76	4.72	6.2	24	24
aqua-c.pl	690.51	∞	∞	3327	?
b2.pl	2.49	3.54	1.4	23	23
boyer.pl	0.43	1.76	4.1	20	20
browse.pl	0.14	2.04	15.0	2	2
chat.pl	36.77	∞	∞	887	?
chat-parser.pl	12.88	∞	∞	267	?
chess.pl	0.71	19.74	27.9	7	7
circuit.pl	0.06	0.05	0.9	0	0
cs.pl	0.24	0.16	0.6	0	0
flatten.pl	0.63	14.62	23.4	22	22
gabriel.pl	0.22	0.23	1.1	1	1
life.pl	0.09	0.06	0.7	0	0
nand.pl	1.05	0.62	0.6	0	0
peep.pl	0.24	0.16	0.7	0	0
press.pl	1.26	4.02	3.2	42	42
read.pl	0.69	1.08	1.6	5	5
readq.pl	2.81	52.82	18.8	44	44
reducer.pl	0.87	∞	∞	100	?
scc.pl	0.44	0.27	0.6	0	0
sdda.pl	0.91	∞	∞	53	?
sendmore.pl	0.21	0.16	0.8	0	0
serialise.pl	0.26	4.40	16.9	9	9
triangle.pl	0.18	0.10	0.6	0	0
wgc.pl	0.04	0.03	0.7	0	0

Table 2: Experimental Evaluation of ↓**JL+POS+LIN** and **JL+LIN**

are substantially faster than the analyzers described in [5, 21]. This may partly be due to the fact that we use a highly optimized SML implementation with efficient data structures, whereas the analyzers from [5, 21] are based on the PLAIA system [23] which is implemented in Prolog.

Our experiments confirm that linearity can significantly improve the quality of the sharing analysis. This was already pointed out in [5, 21]. Thus, future sharing analyzers should include a linearity component.

Our new abstract domains have been carefully designed in order to remedy the efficiency problems of **JL** and **JL+LIN**. To our knowledge, ↓**JL+POS+LIN** is the fastest sharing analyzer reported so far. It is precise and fast enough to be integrated into a production quality Prolog compiler.

Acknowledgement We would like to thank Helmut Seidl for many fruitful discussions on the expressiveness of the sharing domains **JL**.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean Functions for Dependency Analysis: Algebraic Properties and Efficient Representation. In *International Static Analysis Symposium SAS'94*, pages 266–280. Springer-Verlag LNCS 864, 1994.

[2] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[3] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulations. *IEEE Transactions on Computers*, C-35(8), August 1986.

[4] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *TOPLAS*, 16(1):35–101, 1994.

[5] M. Codish, A. Mulkers, M. Bruynooghe, M. Garcìa de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):28–44, January 1995.

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

[7] A. Cortesi, G. Filé, and W. Winsborough. Comparison of Abstract Interpretations. In *ICALP'92*, pages 521–532. Springer Verlag, LNCS 623, 1992.

[8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2):103–179, 1992.

[10] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, PLILP'92*, pages 269–295, Leuven, Belgium, 1992. Springer Verlag, LNCS 631.

[11] L. Crnogorac, A.D. Kelly, and H. Søndergaard. A Comparison of Three Occur-Check Analysers. In *Third International Static Analysis Symposium (SAS'96)*, Aachen, Germany, 1996. Springer Verlag, LNCS.

[12] Saumya K. Debray. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5:207–230, 1988.

[13] C. Fecht and H. Seidl. An Even Faster Solver for General Systems of Equations. In *Static Analysis Symposium (SAS'96)*, Aachen, 1996. Springer Verlag, LNCS.

[14] Christian Fecht. GENA – a Tool for Generating Prolog Analyzers from Specifications. In Alan Mycroft, editor, *Second International Symposium on Static Analysis (SAS'95)*, pages 418–419. Springer Verlag, LNCS 983, 1995.

[15] M. Hermenegildo. *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, 1986.

[16] D. Jacobs and A. Langen. Static analysis of logic programs for independent and-parallelism. *Journal of Logic Programming*, 13:291–314, 1992.

[17] G. Janssens and W. Simoens. On the Implementation of Abstract Interpretation Systems for (Constraint) Logic Programs. In Peter Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, pages 172–187, Edinburgh, U.K., 1994. Springer Verlag, LNCS 786.

[18] N.D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of PROLOG. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood, 1987.

[19] A. King. A Synergistic Analysis for Sharing and Groundness which Traces Linearity. In Donald Sannella, editor, *ESOP'94, 5th European Symposium on Programming*, pages 363–378, Edinburg, U.K, 1994. Springer Verlag, LNCS 788.

[20] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2:181–196, 1993.

[21] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *ICLP95*, pages 781–795, 1995.

[22] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables through Abstract Interpretation. In *ICLP91*, pages 49–63, 1991.

[23] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *The Journal of Logic Programming*, 13:315–347, 1992.

[24] Magnus Nordin. IGOR*: A tool for developing abstract domains for Prolog Analyzers*. PhD thesis, Computing Science Department, Uppsala University, 1995.

[25] R. Sundararajan and S. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *Proceedings twelfth FST/TCS conference*, pages 203–216. Springer Verlag, LNCS 652, 1992.

[26] Andrew Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sidney, 1991.

[27] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain Prop. *The Journal of Logic Programming*, 23(3):237–278, 1995.

[28] Peter van Roy. *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, University of Berkeley, 1991.

Appendix

Proof of Lemma 2:

1. From $S \subseteq {\downarrow}S$, immediately follows $share_{\mathrm{JL}}(S) \subseteq share_{\mathrm{JL}}({\downarrow}S)$. Consider now a sharing pair $(X,Y) \in share_{\mathrm{JL}}({\downarrow}S)$. Hence, there is an $M \in {\downarrow}S$ with $X,Y \in M$ and an $M' \in S$ with $M \subseteq M'$. Thus, $(X,Y) \in share_{\mathrm{JL}}(S)$, too.

2. From $ground_{\mathrm{JL}}(S) = \mathbf{V} - vars(S)$ and $S \subseteq {\downarrow}S$ follows $ground_{\mathrm{JL}}({\downarrow}S) \subseteq S$. Consider now an arbitrary $X \in ground_{\mathrm{JL}}(S)$. Then $X \notin M$ for each $M \in S$. Hence, $X \notin M'$ for each $M' \subseteq M$ and $M \in S$. From this follows $X \in ground_{\mathrm{JL}}({\downarrow}S)$.

3. $def_{\mathrm{JL}}({\downarrow}S) = \bigwedge \{ \bigwedge W_1 \rightarrow \bigwedge W_2 \mid \forall M \in {\downarrow}S.(W_2 \cap M \neq \emptyset) \Rightarrow (W_1 \cap M \neq \emptyset)\}$. Since $((\mathbf{V} - vars(S)) \cap M \neq \emptyset) \Rightarrow (\emptyset \cap M \neq \emptyset)$ for all $M \in S$, the conjunction $\bigwedge ground_{\mathrm{JL}}(S)$ is a conjunct of $def_{\mathrm{JL}}({\downarrow}S)$. Consider now two other sets W_1 und W_2 which fulfill the above condition. Suppose that there is a variable X with $X \in W_2$ and $X \in vars({\downarrow}S)$. Hence, $\{X\} \in {\downarrow}S$ und $W_2 \cap \{X\} \neq \emptyset$. It must follow that $W_1 \cap \{X\} \neq \emptyset$. Thus, $X \in W_1$, too. We have shown that $W_2 \cap vars(S) \subseteq W_1 \cap vars(S)$. Because of the following two propositional rules $x \wedge y \wedge (y \wedge x \rightarrow w) = x \wedge y \wedge (x \rightarrow w$ and $x \wedge y \wedge (x \rightarrow y \wedge w) = x \wedge y \wedge (x \rightarrow w)$ we can assume

that W_1 and W_2 does not contain variables from $ground_{\mathrm{JL}}(S) = \mathbf{V} - vars(S)$. In this case $W_2 \subseteq W_1$ holds. Thus, the conjunct $\bigwedge W_1 \rightarrow \bigwedge W_2$ is equivalent to $true$. □