Set-Sharing is not always redundant for Pair-Sharing

Francisco Bueno[1] and Maria Garcia de la Banda[2]

[1] Dept. of Computer Science, UPM, Spain
[2] School of Computer Science and S.E., Monash University, Melbourne

Abstract. Sharing among program variables is vital information when analyzing logic programs. This information is often expressed either as sets or as pairs of program variables that (may) share. That is, either as set-sharing or as pair-sharing. It has been recently argued that (a) set-sharing is interesting not as an observable property in itself, but as an encoding for accurate pair-sharing, and that (b) such an encoding is in fact redundant and can be significantly simplified without loss of pair-sharing accuracy. We show that this is not the case when set-sharing is combined with other kinds of information, such as the popular freeness.

1 Introduction

Program analysis is the process of inferring at compile–time inferring information about run–time properties of programs. In logic programs one of the most studied run-time properties is *sharing* among program variables. Two program variables share in a given run-time store if the terms to which they are bound have at least one run-time variable in common. A set of program variables share if they all have at least one run-time variable in common. The former kind of sharing is called *pair-sharing* while the latter is called *set-sharing*. Any of the two may be target observables of an analysis.

The importance (and hence popularity) of sharing comes from two sources. First, sharing information is in itself vital for several applications such as exploitation of independent AND-parallelism [18, 5], occurs check reduction [26, 27], and compile-time garbage collection [23]. And second, sharing can be used to accurately keep track of other interesting run-time properties such as *freeness* (a program variable is free in a run-time store if it is either unbound or bound to a run-time variable).

Sharing analysis has therefore raised an enormous amount of interest in our research community, with many different analysis domains being proposed in the literature (see e.g., [27, 17, 25, 3, 19]). Two of the best known sharing analysis domains are `ASub` defined by Søndergaard [27] and `Sharing` defined by Jacobs and Langen [17, 18]. The main difference between these two domains is the way in which they represent sharing information: while `ASub` keeps track of *pairs* of program variables that possibly share, `Sharing` keeps track of *sets* of program variables that possibly share certain variable occurrences.

These differences have subtle consequences. On the one hand, the pair sharing encoding in `ASub` allows it to keep track of linear program variables (a program

variable is *linear* in a run-time store if it is bound to a term which does not have multiple occurrences of the same run-time variable). Linearity information, in turn, allows `ASub` to improve the accuracy of the abstract sharing operations. On the other hand, the set sharing encoding in `Sharing` allows it to represent several other kinds of information (such as groundness and sharing dependencies) which also result in more accurate abstract operations. In fact, when combined with linearity, `Sharing` is strictly more accurate than `ASub`. In practice, this accuracy improvement has proved to be significant [7].

As a result, `Sharing` became the standard choice for sharing analysis, usually combined with other kinds of information such as freeness or structural information, even though its complexity can have significant impact on efficiency. However, the benefits of using set sharing for sharing analysis have been recently questioned (see [10, 1, 2]). As a paradigm of the case, we cite the title of a paper by Bagnara, Hill, and Zaffanella: "Set-Sharing is redundant for Pair-Sharing" [1, 2]. In this paper, the authors state the following

> **Assumption**: The goal of sharing analysis for logic programs is to detect which *pairs* of variables are definitely independent (namely they cannot be bound to terms having one or more variables in common).
>
> As far as we know this assumption is true. In the literature we can find no reference to the "independence of a *set* of variables". All the proposed applications of sharing analysis (compile-time optimizations, occur-check reduction and so on) are based on information about the independence of *pairs* of variables.

Based on the above assumption, the authors focus on defining a simpler version of `Sharing` which is however as precise as far as pair-sharing is concerned. This new simpler domain, referred to in the future as SS^{ρ}, is obtained by eliminating from `Sharing` information which is considered "redundant" w.r.t. the pair-sharing property. This elimination allows further simplification of the abstract operations in SS^{ρ} which can significantly improve its efficiency.

The popularity of the `Sharing` domain combined with the great accuracy and efficiency results obtained for SS^{ρ} (and the clarity with which the authors explained the intricacies of the `Sharing` domain), ensured the paper had a significant impact on the community, with many researchers now accepting that set-sharing is indeed redundant for pair-sharing (see, e.g., [20, 8, 22, 21]).

The aim of this paper is to prove that this is not always the case. In particular, we will show that: (1) There exist applications which use set-sharing analysis (combined with freeness) to infer properties other than sharing between pairs of variables; and (2) When combined with information capable of distinguishing among the different variable occurrences represented by `Sharing`, this domain can yield results not obtainable with SS^{ρ}, *including better pair-sharing*. Such a combination is found in at least two common situations: when `Sharing` is used as a carrier for other analyses (such as freeness), and when the analysis process is improved with extra information (such as in-lined knowledge of the semantics of some predicates, for example builtins). Possible approaches to combine SS^{ρ} with other kinds of information without losing accuracy are also suggested.

We believe our insights will contribute to the better understanding of an abstract domain which, while being one of the most popular and more intensively studied abstract domains ever defined, remains somewhat misunderstood.

2 Preliminaries

Let us start by introducing our notation as well as the basics of the `Sharing` domain [17, 18]. In doing this we will mainly follow the extremely clear summary presented in [1]. Given a set S, $\wp(S)$ denotes the powerset of S, and $\wp_f(S)$ denotes the set of all the finite subsets of S. $\mathcal{V}$ denotes a denumerable set of variables. $Var \in \wp_f(\mathcal{V})$ denotes a finite set of variables, called the *variables of interest* (e.g., the variables of a program). The set of variables in a syntactic object o is denoted $vars(o)$. $\mathcal{T}_\mathcal{V}$ is the set of first order terms over $\mathcal{V}$. A substitution θ is a mapping $\theta : \mathcal{V} \to \mathcal{T}_\mathcal{V}$, whose application to variable x is denoted by $x\theta$. Substitutions are denoted by the set of their bindings: $\theta = \{x \mapsto x\theta \mid x\theta \neq x\}$. We define the image of a substitution θ as the set $img(\theta) \stackrel{\text{def}}{=} \bigcup\{vars(x\theta) \mid x \in Var\}$.

The `Sharing` domain is formally defined as follows. Let $SH \stackrel{\text{def}}{=} \wp(SG)$, where $SG \stackrel{\text{def}}{=} \{S \subseteq Var \mid S \neq \emptyset\}$. Each element $S \in SG$ is called a *sharing set*. We will write sharing sets as strings with the variables that belong to it, e.g., sharing set $\{x, y, z\}$ will be denoted xyz. A sharing set of size 2 is called a *sharing pair*.

The function $occ(\theta, v)$ obtains a sharing set that represents the occurrence of variable v through the variables of interest as per the substitution θ.

$$occ(\theta, v) \stackrel{\text{def}}{=} \{x \in Var \mid v \in vars(x\theta)\}$$

The abstraction of a substitution θ is obtained by computing all relevant sharing sets:

$$\alpha(\theta) \stackrel{\text{def}}{=} \{occ(\theta, v) \mid v \in img(\theta)\}.$$

Abstract element $sh \in SH$ approximates substitution θ iff $\alpha(\theta) \subseteq sh$. Conversely, the concretization of $sh \in SH$ is the set of all substitutions approximated by sh. Projection over a set $V \subseteq Var$ is given by

$$proj(sh, V) \stackrel{\text{def}}{=} \{S \cap V \mid S \in sh[V]\}$$

where, for any syntactic object o and abstraction $sh \in SH$,

$$sh[o] \stackrel{\text{def}}{=} \{S \in sh \mid S \cap vars(o) \neq \emptyset\}.$$

The pairwise (or binary) union of two abstractions is defined as:

$$sh_1 \uplus sh_2 \stackrel{\text{def}}{=} \{S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2\}.$$

The closure under (or star) union of an abstract element sh is defined as the least set sh^* that satisfies:

$$sh^* = sh \cup \{S_1 \cup S_2 \mid S_1, S_2 \in sh^*\}.$$

Abstract unification for a substitution θ is given by extending to the set of bindings of θ the following abstract unification operation for a binding:

$$amgu(sh, x \mapsto t) = (sh \setminus (sh[x] \cup sh[t])) \cup (sh[x]^* \uplus sh[t]^*).$$

The set-sharing lattice is thus given by the set

$$SS \stackrel{\text{def}}{=} \{(sh, U) | sh \in SH, U \subseteq Var, \forall S \in sh : S \subseteq U\} \cup \{\bot, \top\}$$

which is a complete lattice ordered by $\leq_{SS}$ defined as follows. For elements $\{d, (sh_1, U_1), (sh_2, U_2)\} \subseteq SS$:

$\bot \leq_{SS} d$
$d \leq_{SS} \top$
$(sh_1, U_1) \leq_{SS} (sh_2, U_2)$ iff $U_1 = U_2$ and $sh_1 \subseteq sh_2$.

The lifting of $\cup$, $proj$, and $amgu$ defined over SH to define the abstract operations $\sqcup$, $Proj$, and $Amgu$ over SS is straightforward.

Example 1. Let $Var = \{x, y, z\}$ be the set of variables of interest and consider the substitutions $\theta_1 = \{x \mapsto f(u, u, v), y \mapsto g(u, v, w, o), z \mapsto h(u)\}$ and $\theta_2 = \{x \mapsto u, y \mapsto u, z \mapsto 1\}$. Then, $sh_1 = \alpha(\theta_1) = \{xy, xyz, y\}$, where sharing set xyz represents the occurrence of variable u in x, y and z, sharing set xy represents the occurrence of variable v in x and y, and sharing set y represents the occurrence of variables w and o in y. Similarly, we have that $sh_2 = \alpha(\theta_2) = \{xy\}$ where sharing set xy represents the occurrence of variable u in x and y. Let $U = Var$. We then have that $(sh_2, U) \leq_{SS} (sh_1, U)$ and thus $(sh_1, U) \sqcup (sh_2, U) = (sh_1, U)$. Finally, let $V = \{x, y\}$, $Proj((sh_1, U), V) = (\{xy, y\}, V)$. Note that the sharing set xy in the projected abstraction represents not only the occurrence of variable u but also that of v. $\diamond$

3 Eliminating redundancy from `Sharing`

One of the main insights in [10, 1] regarding the `Sharing` domain is the detection of sets which are redundant (and can thus be safely eliminated or not produced) as far as pair-sharing is concerned. Given an element sh of SH, sharing set $S \in sh$ is *redundant* w.r.t. pair sharing if and only if all its sharing pairs can be extracted from other subsets of S which also appear in sh. Formally, let $pairs(S) \stackrel{\text{def}}{=} \{xy \mid x, y \in S, x \neq y\}$. Then, S is redundant iff

$$pairs(S) = \bigcup \{pairs(T) \mid T \in sh, T \subset S\}$$

Example 2. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$. It is easy to see that set $xyz \in sh$ is redundant w.r.t. pair sharing. $\diamond$

Based on this insight, a closure operator, $\rho : SH \to SH$, is defined in [1] to add to each $sh \in SH$ the set of elements which are redundant for sh. Formally:

$$\rho(sh) \stackrel{\text{def}}{=} \{S \in SG \mid \forall x \in S : S \in sh[x]^*\}.$$

This function is then used to define a new domain SS^ρ which is the quotient of SS w.r.t. the new equivalence relation induced by ρ: elements d_1 and d_2 are equivalent iff $\rho(d_1) = \rho(d_2)$. The authors prove that (a) the addition of redundant elements does not cause any precision loss as far as pair-sharing is concerned, i.e., that SS^ρ is as good as SS at representing pair-sharing, and that (b) ρ is a congruence w.r.t. the abstract operations $Amgu$, $\sqcup$ and $Proj$. Thus, they conclude that SS^ρ is as good as SS also for propagating pair-sharing through the analysis process.

The above insight is used by [1] to perform two major changes to the `Sharing` domain. Firstly, redundant elements can be eliminated (although experimental results suggest that this is not always advantageous). And secondly, addition of redundant elements can be avoided by replacing the star union with the binary union operation without loss of accuracy. This is a very important change since it can have significant impact on efficiency by simplifying one of the most expensive abstract operations in `Sharing`.

The results obtained in [1] are indeed interesting and can be very useful in some contexts. However, there are situations in which the lack of redundant sets can lead to loss of accuracy w.r.t. pair sharing, and even incorrect results if the full expressive power of `Sharing` is assumed to be still present in SS^ρ.

Example 3. Consider the abstractions $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$, and note that $\rho(sh_1) = sh_2$, i.e., the sharing set xyz is redundant for sh_2.

Consider the Prolog builtin $x == y$ which succeeds if program variables x and y are bound at run-time to identical terms. A sophisticated implementation of the `Sharing` domain (such as that of [4]) could take advantage of this information and eliminate every single sharing set in which the program variables x and y appear but not together (since all variables which occur in x must also occur in y, and vice versa). Thus, correct and precise abstractions of a situation in which the builtin was successfully executed in stores represented by sh_1 and sh_2, will become $sh'_1 = \{z, xy\}$ and $sh'_2 = \{z, xy, xyz\}$, respectively. However, it is easy to see that $pairs(sh'_1) \neq pairs(sh'_2)$, since z is definitely independent of both x and y in sh'_1 while it might still share with them in sh'_2. $\diamond$

The above example shows that `Sharing` can make use of the information provided by other sources in order to improve the pair-sharing accuracy of its elements, while the same action might lead to incorrect results for elements of SS^ρ if redundant sharing sets had actually been eliminated from those elements. As we will see in the following sections, this can happen when using information coming not only from builtins, but also from other domains (such as freeness) which are usually combined with set-sharing. Furthermore, useful information other than sharing can be inferred from combinations of `Sharing` and other sources which are not possible with SS^ρ.

4 When redundant sets are no longer redundant

The problem illustrated in the previous example is rooted in the always surprising complexity of the information encoded by elements of SH. As indicated by [1, 2], elements of SH can encode definite groundness (e.g., x is ground), groundness dependencies (e.g., if x becomes ground then y is ground), and sharing dependencies.[3] However, as we will see in this section, these are only by-products of the

[3] The fact that it also encodes independence (e.g., x does not share with y) was probably obviated because this is also encoded by pair-sharing.

main property represented by elements of SH: the different variable occurrences shared by each set of program variables.

The groundness of variable x, and the sharing independence between variables x and y (i.e., the fact that x and y are known not to share) can be expressed by an element $sh \in SH$ as follows:

$$\begin{array}{c} ground(x) \text{ iff } \forall S \in sh : \ x \notin S \\ indep(x,y) \text{ iff } \forall S \in sh : xy \not\subseteq S \end{array}$$

where $ground(x)$ represents the fact that variable x is ground in all substitutions abstracted by sh, and $indep(x,y)$ represents the fact that variables x and y do not share in any substitution abstracted by $sh \in SH$.

Groundness dependencies in $sh \in SH$ can be easily obtained from the above statements in the following way. Let us assume that x is known to be ground. We can then modify sh by enforcing $\forall S \in sh : \ x \notin S$ to hold, i.e., by eliminating every $S \in sh$ such that $x \in S$. If we can then prove that the same statement holds for some other variable y, we would then know that the implication $ground(x) \rightarrow ground(y)$ holds for sh. This simply illustrates the well known result that `Sharing` subsumes the groundness dependency domain Def. The same method can be used for obtaining other dependencies for elements sh of SH. The following were used in [5] for simplifying parallelization tests:

1. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow ground(y)$ if
   $\forall S \in sh : \text{ if } y \in S \text{ then } \{x_1, \ldots, x_n\} \cap S \neq \emptyset$
2. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow indep(y,z)$ if
   $\forall S \in sh : \text{ if } \{y,z\} \subseteq S \text{ then } \{x_1, \ldots, x_n\} \cap S \neq \emptyset$
3. $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow ground(z)$ if
   $\forall S \in sh : \text{ if } z \in S \text{ then } \exists j \in [1,n], \ \{x_j, y_j\} \subseteq S$
4. $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow indep(w,z)$ if
   $\forall S \in sh : \text{ if } \{w,z\} \subseteq S \text{ then } \exists j \in [1,n], \ \{x_j, y_j\} \subseteq S$

Let us now characterize in a similar way the (non-symmetrical) property $covers(x,y)$ expressed by an element $sh \in SH$ as follows:

$$covers(x,y) \text{ iff } \forall S \in sh : \text{ if } y \in S \text{ then } x \in S$$

where $covers(x,y)$ indicates that variable y shares all its variables with variable x and, therefore, every sharing set in which y appears must also contain x. We can now derive other sharing dependencies for any $sh \in SH$, such as:

5. $covers(x_1,y_1) \wedge \ldots \wedge covers(x_n,y_n) \rightarrow ground(z)$ if
   $\forall S \in sh : \text{ if } z \in S \text{ then } \exists j \in [1,n], \ y_j \in S, \ x_j \notin S$
6. $covers(x_1,y_1) \wedge \ldots \wedge covers(x_n,y_n) \rightarrow indep(w,z)$ if
   $\forall S \in sh : \text{ if } \{w,z\} \subseteq S \text{ then } \exists j \in [1,n], \ y_j \in S, \ x_j \notin S$
7. $covers(x_1,y_1) \wedge \ldots \wedge covers(x_n,y_n) \rightarrow covers(w,z)$ if
   $\forall S \in sh : \text{ if } z \in S, w \notin S \text{ then } \exists j \in [1,n], \ y_j \in S, \ x_j \notin S$

It is important to note that while the expressions with only $ground(x)$ and $indep(x, y)$ elements can also hold for any element of SS^ρ, this is not true for the expressions with coverage information.

Example 4. Consider again the abstractions introduced by Example 3, $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ which are defined over $Var = \{x, y, z\}$. Let us assume that both abstractions belong to `Sharing`. While implication $covers(x, y) \wedge covers(y, x) \rightarrow indep(x, z)$ holds for sh_1, it does not hold for sh_2. If we now consider the SS^ρ domain, both abstractions would be represented by the element sh_1. Therefore, the implication should not hold for sh_1 in SS^ρ. $\diamond$

In order to understand why, consider the differences between the expressions $ground(x)$ iff $\forall S \in sh : x \notin S$, and $indep(x, y)$ iff $\forall S \in sh : xy \not\subseteq S$, and the expression $covers(x, y)$ iff $\forall S \in sh$: if $y \in S$ then $x \in S$. While in the first two the sharing sets which violate the right hand side of the expressions would always include the redundant set (if any), those which violate the last expression would not. Thus, to assume coverage might result in the subset of a redundant set being eliminated without the redundant set itself being eliminated. In this way sharing sets which are considered redundant at some point, might become non redundant once coverage information is added and, therefore, their elimination (or non generation) can lead to incorrect information. For example, consider the substitution $sh = \{xyz, xy, xz, yz\}$. While the problematic sets for $ground(x)$ and $indep(x, y)$ in sh are xyz, xy, xz and xyz, xy, respectively, the only one for $covers(x, y)$ is yz. But once yz is removed from sh, xyz is no longer redundant: it is the only sharing set able (when x covers y) to represent the possible sharing between x and y.

As a result, sharing sets initially redundant for pair-sharing can prove useful whenever combined with other sources of information (coming from builtins, other analysis domains, etc.) capable of distinguishing between the variable occurrences represented by the redundant sharing sets and the variable occurrences represented by their subsets, so that, once the extra information is added, a sharing set previously identified as redundant will no longer be so.

5 Combining `Sharing` with freeness

In this section we will use the popular combination of `Sharing` with freeness information to illustrate two points. First, that very common sources of information (such as freeness) can distinguish between variable occurrences, an ability which can be exploited in ways that can make a redundant set no longer redundant. Thus, it can be advantageous not to eliminate them. And second, that the goal of sharing analysis for logic programs is not only to detect which pairs of variables are definitely independent, but also to detect (or propagate) many other kinds of information.

In order to illustrate these points we will use the notion of *active* sharing sets [6]. A sharing set $S \in sh$ is said to be *active* for store $c \in \gamma(sh)$ iff $S \in \alpha(c)$. All

sharing sets $\{S_1, \cdots, S_n\} \subseteq sh$ are said to be active *at the same time* if there exists a store $c \in \gamma(sh)$ such that $\forall 1 \leq i \leq n, S_i \in \alpha(c)$. If only the information in `Sharing` is taken into account, then all sharing sets in any $sh \in SH$ can be active at the same time.

Example 5. Consider the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. All sets in sh can be active at the same time since there exists a store, say $\theta = \{x = f(u, v), y = f(v, w), z = f(w)\}$, such that $\alpha(\theta) = sh$. In particular, u is the variable represented by sharing set x, v is represented by xy, and w is represented by yz. $\diamond$

However, this is not always the case when considering information outside the scope of `Sharing`. In some cases, two or more sharing sets cannot be active at the same time since, thanks to some extra information, we can determine that these sharing sets must represent the same variable(s) occurrence.

Example 6. Consider again the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$, and let us now assume y and z are known to be free variables. As pointed out in [6], since each sharing set in an abstraction represents a different occurrence of one or more variables, no two sharing sets containing the same free variable can be active at the same time (the same variable cannot be a different occurrence). In our example, xy and yz cannot be active at the same time since there is no concrete store with both y and z free, such that both share a variable not shared with anyone else (sharing set yz) and y also shares a different variable with x (sharing set xy). $\diamond$

Knowing which sharing sets in abstraction sh can be active at the same time according to Ω is useful because we can use thois notion to divide sh into $\{sh_1, \cdots, sh_n\}$ such that $sh = sh_1 \cup \ldots \cup sh_n$, $\forall i, 1 \leq i \leq n$ all sets in sh_i can be active at the same time, and $\neg\exists j, 1 \leq j \leq n : j \neq i, sh_j \subseteq sh_i$.

Example 7. Consider again the abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. If y and z are known to be free variables, sh can be divided into two different sets, $\{x, xy\}$ and $\{x, yz\}$, whose sharing sets can all be active at the same time. The former represents the concrete stores in which x definitely shares a variable with y (which is actually known to be y itself), and x might also have some variable which is not shared with anyone else. The latter represents the stores in which the free variables y and z are aliased and x might have some variables which are not shared with anyone else. $\diamond$

Note that the different sh_i together with Ω describe disjoints sets of concrete stores. Furthermore, even though $(\bigcup_i \gamma(sh_i)) \cap \gamma(\Omega)$ is still equivalent to $\gamma(sh) \cap \gamma(\Omega)$ (which justifies the correctness of dividing sh into the different sh_i in the presence of Ω), it is often the case that $\bigcup_i \gamma(sh_i) \subset \gamma(sh)$, as it happens in the above example. As a result, it is generally easier to understand the concretization of sh and Ω by means of the concretization of each sh_i and Ω. Let us use this to

show how the direct-product domain [11] of `Sharing` and freeness can be used to improve pair-sharing.

Example 8. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over program variables x, y and z. If we knew that x, y, and z are free we could divide sh into the sets $sh_1 = \{xy\}, sh_2 = \{xz\}, sh_3 = \{yz\}$ and $sh_4 = \{xyz\}$. Now, sh_1 represents stores in which z is known to be ground, which is not true according to our freeness information. Thus, its sharing sets (xy) can be eliminated from sh. The same reasoning applies to sh_2 and sh_3. Thus, sh can be simplified to $\{xyz\}$ indicating that all variables definitely share (which of course also implies their definite pair-sharing dependencies). Note that if the set xyz did not belong to the abstraction, the concretization of sh in the context of freeness would be empty (indicating a failure in the program). $\diamond$

The above example shows how the direct-product domain of SS^ρ and freeness might be incorrect if the full power of set-sharing is assumed to be still present in SS^ρ. This occurs whenever a redundant set is known to contain a free variable, since it would then appear in an sh_i without one or more of its subsets. Thus, the set would no longer be redundant for sh_i. A simple solution would be to behave as if redundant sets containing free variables were present in the SS^ρ abstractions even if they do not appear explicitly in them. It would be easy to think that such solution does not lose accuracy w.r.t. pair sharing. This is, however, not true.

Example 9. Consider the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. If we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$, respectively representing the concrete stores in which x shares with y and z, which do not share among them, and those in which x does not share with anyone and y shares with z. Note that these two situations are mutually exclusive. This allow us to prove (among others) that:

$$indep(y, z) \text{ iff } \neg indep(x, y) \quad \text{and} \quad indep(y, z) \text{ iff } \neg indep(x, z).$$

This is crucial pair-sharing information (e.g., for automatic AND-parallelization, as we will see in the next section). If the redundant set xyz could have been eliminated from sh, the above expression might not hold, since the variables might then be aliased to the same free variable, thus capturing also the case in which all of them are definitely dependent of each other. $\diamond$

Let us now show how combining `Sharing` and freeness information, as done for example in `Sharing+Freeness` [25], yields interesting kinds of information other than the sharing itself, information which is the goal of such analyses for several applications.

Example 10. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. As mentioned above, if we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$. The concrete stores represented by these sets can in fact be described much more accurately than we did in the previous example: While sh_1 represents stores in which x is bound to a term with two (and only two) non-aliased free variables

(y and z), sh_2 represents those stores in which x is ground, and y and z are free aliased variables. As a result, we can be sure sh only represents stores in which x is bound to a non-variable term. $\diamond$

Definite information about non-variable bindings is used, for example, to determine whether dynamic scheduled goals waiting for a program variable to become non-variable can be woken up, as performed by [15]. However, such information cannot be obtained if redundant sets containing free variables are eliminated.

Example 11. Consider the set-sharing abstractions $sh = \{xy, xz, yz\}$ above and $sh' = sh \cup \{xyz\}$ where y and z are known to be free, we could divide sh' into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$ and $sh_3 = \{xyz\}$. The first two are as above, while the third represents stores in which all x, y and z share the same variables (with x possibly being a free variable). Thus, sh' does not only represent stores in which x is bound to a non-variable term. $\diamond$

Definite knowledge about non-variable bindings is not the only kind of useful information that can be inferred from combining `Sharing` and freeness. The combination can also be used to detect new bindings added by some body literal.

Example 12. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that sh is the abstract call for body literal $p(x, y, z)$ (i.e., the abstraction at the program point right before executing the literal) and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x, y, z)$ (i.e., the abstraction at the program point right after executing the literal) with y and z still known to be free. The addition of sharing set xyz means that a new binding aliasing y and z might have been introduced by $p(x, y, z)$. However, if the abstract answer is found to be identical to the call sh, we can be sure that none of the three program variables has been further instantiated (since they are still known to be free) nor any new aliasing introduced among them. $\diamond$

The above kind of information is used, for example, for detecting non-strict independence [6] as we will see in the next section. As shown in the above example, this information cannot be inferred if redundant sets might have been eliminated (or not produced).

6 When independence among sets is relevant

This section uses the well-known application of automatic parallelization within the independent AND-parallelism model [9] to illustrate how some applications (a) require independence among sets (as opposed to pairs) of variables, and (b) can benefit from combining `Sharing` with freeness information in ways which would not be possible with SS^{ρ}. The relevance of this application comes from the fact that it is not only one of the best known applications of sharing information, but also the one for which the `Sharing` domain was developed.

In the independent AND-parallelism model goals g_1 and g_2 in the sequence g_1, g_2 can be run in parallel in constraint store c if g_2 is independent of g_1 for

store c. In this context, independence refers to the conditions that the run-time behavior of these goals must satisfy in order to guarantee the correctness and efficiency of their parallelization w.r.t. their sequential execution. This can be expressed as follows: goal g_2 is independent of goal g_1 for store c iff the execution of g_2 in c has the same number of computation steps, cost, and answers as that of g_2 in any store c' obtained from executing g_1 in c.

Note that the general independence condition introduced above is thus neither symmetric nor established between pairs of variables, as assumed by [1, 2]. However, this general notion of independence is indeed rarely used. Instead, sufficient (and thus simpler) conditions are generally used to ensure independence. These conditions can be divided into two main groups: a priori and a posteriori. A priori conditions can always be checked prior to the execution of the goals involved, while a posteriori conditions can be based on the actual behaviour of the goals to be run in parallel.

A priori conditions are more popular even though they can be less accurate. The reasons are twofold. First, they can only be based on the characteristics of the store c and the variables belonging to the goals to be run in parallel. Thus, they are relatively simple. And second, they can be used as run-time tests without actually running the goals themselves. This is useful whenever the conditions cannot be proved correct at compile-time. Note that a priori conditions must be symmetric: goals g_1 and g_2 are independent for c iff g_1 is independent of g_2 for c *and* g_2 is independent of g_1 for c.

The most general a priori condition, called *projection independence*, was defined in [14] as follows: goals g_1 and g_2 are independent for c if for any variable $x \in vars(g_1) \cap vars(g_2)$, x is uniquely defined by c (i.e., ground), and the constraint obtained by conjoining the projection of c over $vars(g_1)$ and the projection of c over $vars(g_2)$ entails (i.e., logically implies) the constraint obtained by projecting c over $vars(g_1) \cup vars(g_2)$.

Example 13. Consider the literals $p(x), q(y), r(z)$ and constraint $c \equiv \{x = y + z\}$. The projection of c over the sets of variables containing either one or two variables from $\{x, y, z\}$ is the empty constraint *true*. Thus, we can ensure that every pair of literals, say $p(x)$ and $q(y)$, can run in parallel. However, no literal can run in parallel with the goal formed by the conjunction of the other two literals, e.g., $p(x)$ cannot run in parallel with goal $q(y), r(z)$, since the projection of c over $\{x, y, z\}$ is c itself, which is indeed not entailed by *true*. $\diamond$

Therefore, as mentioned in both [24] and [13], in general projection independence does indeed rely on the independence of a pair of *sets* of variables. However, for the Herbrand case projection independence is equivalent to the better known a priori condition called *strict independence*, which was introduced in [9, 12] and formally defined and proved correct in [16]. It states that goals g_1 and g_2 are strictly independent for substitution θ iff $vars(g_1)$ do not share with $vars(g_2)$ for θ, i.e., iff $vars(g_1\theta) \cap vars(g_2\theta) = \emptyset$. It is easy to prove that this is equivalent to requiring that for every pair of variables $xy, x \in vars(g_1), y \in vars(g_2)$, x and y do not share.

Therefore, only for a priori conditions and the Herbrand domain, is parallelization based on the independence of pairs of variables. And even in this case, the `Sharing` domain is more powerful than SS^{ρ} when combined with other kinds of information.

Example 14. Consider again the abstractions $sh = \{xy, xz, yz, xyz\}$ and $sh' = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. Example 9 illustrated how the formula

$$indep(y, z) \text{ iff } \neg indep(x, y) \quad \text{and} \quad indep(y, z) \text{ iff } \neg indep(x, z)$$

hold for sh' but not for sh when y and z are known to be free.

Consider the automatic parallelization of sequential goal `p(y),q(z),r(x)` for the usual case of the a priori condition strict independence and the Herbrand domain. In the absence of any information regarding the state of the store occurring right before the sequential goal is executed, the compiler could rewrite the sequential goal into the following parallel goal (leftmost column):

```
( indep(y,z) ->                 ( indep(y,z) ->                 ( indep(y,z) ->
  ( indep(x,y) ->
    ( indep(x,z) ->
      p(y)&q(z)&r(x)
    ; p(y)&(q(z),r(x))
    )
  ; (p(y)&q(z)),r(x)              (p(y)&q(z)),r(x)                (p(y)&q(z)),r(x)
  )
; indep(x,z) ->                 ;                               ; indep(x,z) ->
  p(y),(q(z)&r(x))                p(y),(q(z)&r(x))                p(y),(q(z)&r(x))
; p(y),q(z),r(x)                                                 ; p(y),q(z),r(x)
)                               )                               )
```

where the operator `&` represents parallel execution of two goals, and the run-time test `indep(x,y)` succeeds if the two variables do not share at run-time. The middle and right columns represent the simplifications that can be performed to the parallel goal in the context of sh' and sh, respectively. This is because while test `indep(x,y)` is known to fail if `indep(y,z)` succeeds for both sh and sh', test `indep(x,z)` is known to succeed if `indep(y,z)` fails for sh' but not for sh. Thus, `indep(x,z)` still needs to be tested at run-time with the resulting loss of efficiency. $\diamond$

The assumption is also incorrect when considering a posteriori conditions, even those associated to the Herbrand domain. In particular, strict independence has been generalised to several different [16] a posteriori notions of *non-strict independence*. These notions allow goals that share variables to run in parallel as long as the bindings established for those shared variables satisfy certain conditions. For example, one of the simpler notions only allows g_1 to instantiate a shared variable and does not allow any aliasing (of different shared variables) to be created during the execution of g_1 that might affect goals to the right. Thus, for this notion, the conditions are established between the *bindings* introduced by the two goals over their respective set of variables, and cannot be expressed using only sharing between pairs of variables.

There has been at least one attempt [6] at inferring non-strict independence at compile-time using the abstract domain `Sharing+Freeness`. The inference is based on two conditions. The first ensures that (C1) no shared variables are further instantiated by g_1. This is done by requiring that (a) all shared variables share through variables known to be free in the abstract call of g_1 (all sharing sets in the abstract call containing shared variables also contain a free variable), and (b) all these variables must remain free in the abstract answer of g_1 (all such sharing sets still contain a free variable after the analysis of g_1). This first condition can be detected in the SS^ρ domain since the existence of a free variable in every sharing pair ensures the existence of a free variable in the "redundant" sharing set. Thus, the absence of such sharing set is not a problem.

This is not however the case for the second condition, which ensures that no aliasing is introduced among shared variables by requiring C1 and, additionally, that (C2) there is no introduction in the abstract answer of any sharing set resulting from the union of several sets such that none contain the same free variable, and at least two contain variables belonging to both goals.

Example 15. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that sh is the abstract call for body $p(x, y, z), q(x, y, z)$ and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x, y, z)$ with y and z still known to be free. All sharing sets in sh containing variables from both literals contain a free variable which remains free in sh'. Thus, C1 is satisfied. However, there exists a set xyz in sh' which can be obtained by unioning at least two sets xy and xz in sh which contain variables from both literals and have no variable in common known to be free in sh. The appearance of such a set represents the possible aliasing of y and z by $p(x, y, z)$. This appearance violates C2 and thus the goals cannot run in parallel. Note that if the abstract answer was found to be identical to sh (i.e., if the redundant set xyz was absent), we would have been able to ensure that none of the three program variables had been further instantiated nor any new aliasing introduced among them. Therefore, we could have ensured that g_2 is independent of g_1 for the stores represented by sh and the associated freeness information, thus allowing their parallel execution. $\diamond$

The above example illustrates the fact that an equivalent inference cannot be performed in the SS^ρ domain augmented with freeness *unless care is taken when considering redundant sharing sets which include program variables known to be free.* This is because the inference strongly depends on distinguishing between the different bindings introduced during execution of the goals to be run in parallel, and as a result, on distinguishing between the different shared variables represented by the abstractions in the domain. Thus, elimination of redundant sets can render the method incorrect. One possible solution is to always assume that redundant sets containing free variables are present when combining SS^ρ with freeness information. However, as shown in Example 9, this might be imprecise. Another, more accurate solution, is to only eliminate redundant sets which do not contain variables known to be free.

7 Conclusion

We have shown that the power of set-sharing does not come from representing sets of variables that share, but from representing different variable occurrences. As a result, eliminating from `Sharing` information which is considered "redundant" w.r.t. the pair-sharing property as performed in SS^ρ can have unexpected consequences. In particular, when `Sharing` is combined with some other kinds of information capable of distinguishing among variable occurrences in a way that can make a redundant set no longer redundant, it can yield results not obtainable with SS^ρ, *including better pair-sharing*. Furthermore, there exist applications which use `Sharing` analysis (combined with freeness) to infer properties other than sharing between pairs of variables and which cannot be inferred if SS^ρ is used instead. We have proposed some possible solutions to this problem.

Acknowledgments

This work was funded in part by projects ASAP (EU IST FET Programme Project Number IST-2001-38059) and CUBICO (MCYT TIC 2002-0055). It was also supported by ARC IREX grant X00106666.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In *Static Analysis Symposium*, pages 53–67. Springer-Verlag, 1997.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.
3. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness — all at once. In *International Workshop on Static Analysis*, 1993.
4. F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), February 1994.
5. F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, 1999.
6. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.
7. M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.
8. Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
9. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

10. A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation for comparing static analyses. In *GULP-PRODE'94 Joint Conference on Declarative Programming*, pages 372–397, 1994.
11. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.
12. D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89. Springer Verlag, 1987.
13. M. Garcia de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91. Springer-Verlag, September 1996.
14. M. Garcia de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):296–339, 2000.
15. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*. MIT Press, December 1995.
16. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
17. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.
18. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.
19. A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming*. MIT Press, June 1995.
20. Andy King, Jan-Georg Smaus, and Patricia M. Hill. Quotienting share for dependency analysis. In *European Symposium on Programming*, pages 59–73, 1999.
21. V. Lagoon and P. J. Stuckey. Precise pair-sharing analysis of logic programs. In *ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 99–108, 2002.
22. Giorgio Levi and Fausto Spoto. Non pair-sharing and freeness analysis through linear refinement. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 52–61, 2000.
23. A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Seventh International Conference on Logic Programming*, pages 747–762. MIT Press, June 1990.
24. K. Muthukumar, F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
25. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
26. D. A. Plaisted. The occur-check problem in prolog. In *International Symposium on Logic Programming*, pages 272–281. IEEE, 1984.
27. H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 213*, pages 327–338. Springer-Verlag, 1986.