Set-Sharing is Redundant for Pair-Sharing

Roberto Bagnara[1,*], Patricia M. Hill[1,*], and Enea Zaffanella[2]

[1] School of Computer Studies, University of Leeds, Leeds, LS2 9JT, U.K.
[2] Servizio IX Automazione, Università degli Studi di Modena, and Dipartimento di Informatica, Università di Pisa, Italy.

Abstract. Although the usual goal of sharing analysis is to detect which pairs of variables share, the standard choice for sharing analysis is a domain that characterizes set-sharing. In this paper, we question, apparently for the first time, whether this domain is over-complex for pair-sharing analysis. We show that the answer is *yes*. By defining an equivalence relation over the set-sharing domain we obtain a simpler domain, reducing the complexity of the abstract unification procedure. We present preliminary experimental results, showing that, in practice, our domain compares favorably with the set-sharing one over a wide range of benchmark programs.

1 Introduction

In logic programming, a knowledge of sharing between variables is important for optimizations such as the exploitation of parallelism.

Today, talking about sharing analysis for logic programs is almost the same as talking about the *set-sharing* domain Sharing of Jacobs and Langen [11,12]. The adequacy of this domain is not normally questioned. Researchers appear to be more concerned as to which *add-ons* are best: linearity, freeness, depth-k abstract substitutions and so on [3–5,13,14,16] rather than whether it is the optimal domain for the sharing information under investigation.

What is the reason for this "standard" choice? Well, the *set-sharing* domain is quite accurate: when integrated with linearity information it is strictly more precise than its old challenger, the *pair-sharing* domain ASub of Søndergaard [17]. Indeed, Sharing encodes a lot of information. As a consequence, it is quite difficult to understand: taking an abstract element and writing down its concretization (namely, the concrete substitutions that are approximated by it) is a hard task. So the question arises: is this complexity actually needed for an accurate sharing analysis?

Before answering this question we must agree on what the purpose of sharing analysis is. This paper relies on 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).*

* The work of R. Bagnara and P. M. Hill has been supported by EPSRC under grant GR/L19515.

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.

We thus focus our attention on the pair-sharing property and assume that set-sharing is just a way to compute pair-sharing with a higher degree of accuracy: there may well be other ways. In this paper we question, apparently for the first time, whether the Sharing domain is really the best one for detecting which pairs of variables can share. The answer turns out to be negative: there exists a domain that is simpler than Sharing and, at the same time, is as precise as Sharing, as far as *pair-sharing* is concerned. This domain is the subject of this paper.

The paper is organized as follows. In the next section, we introduce the notation and recall the definition of the abstract domain Sharing. In Section 3, we show that Sharing is unnecessarily complex for capturing pair-sharing information. A new equivalence relation between its elements is defined which is shown to exactly factor out the unwanted information. Section 4 explains the practical consequences of these results and shows that the complexity of abstract unification using our domain is polynomial compared to the exponential complexity for Sharing. Section 5 gives the experimental results and Section 6 concludes the paper. The proofs of the presented results can be found in [2].

2 Preliminaries

In this section we introduce some mathematical notation that will be used throughout the paper, as well as recalling the *set-sharing* domain.

2.1 Notation

For a set S, $\# S$ is the cardinality of S, $\wp(S)$ is the powerset of S, whereas $\wp_{\mathrm{f}}(S)$ is the set of all the *finite* subsets of S. The symbol *Vars* denotes a denumerable set of variables, whereas $\mathcal{T}_{Vars}$ denotes the set of first-order terms over *Vars*. The set of variables occurring in a syntactic object o is denoted by $vars(o)$. A substitution σ is a total function $\sigma \colon Vars \rightarrow \mathcal{T}_{Vars}$ that is the identity almost everywhere; in other words, the *domain* of σ, $\mathrm{dom}(\sigma) \stackrel{\mathrm{def}}{=} \{\, x \in Vars \mid \sigma(x) \neq x \,\}$, is finite. Substitutions are denoted by the set of their *bindings*, thus σ is identified with $\{\, x \mapsto \sigma(x) \mid x \in \mathrm{dom}(\sigma) \,\}$. A substitution σ is *idempotent* if $vars\bigl(\sigma(x)\bigr) \cap \mathrm{dom}(\sigma) = \varnothing$ for each $x \in \mathrm{dom}(\sigma)$. The set of all the idempotent substitutions is denoted by *Subst*.

2.2 The Sharing Domain

The Sharing domain is due to Jacobs and Langen [11].

Definition 1. (The *set-sharing* lattice.) Let[1]

$$SG \stackrel{\mathrm{def}}{=} \{\, S \in \wp_{\mathrm{f}}(\mathit{Vars}) \bigm| S \neq \varnothing \,\}$$

and let $SH \stackrel{\mathrm{def}}{=} \wp(SG)$. The *set-sharing lattice* is given by the set

$$SS \stackrel{\mathrm{def}}{=} \{\, (sh, U) \bigm| sh \in SH, U \in \wp_{\mathrm{f}}(\mathit{Vars}), \forall S \in sh : S \subseteq U \,\} \cup \{\bot, \top\}$$

ordered by $\preceq_{ss}$ defined as follows, for each $d, (sh_1, U_1), (sh_2, U_2) \in SS$:

$$\begin{aligned}
\bot &\preceq_{ss} d, \\
d &\preceq_{ss} \top, \\
(sh_1, U_1) \preceq_{ss} (sh_2, U_2) \quad &\iff \quad (U_1 = U_2) \wedge (sh_1 \subseteq sh_2).
\end{aligned}$$

It is straightforward to see that every subset of SS has a least upper bound with respect to $\preceq_{ss}$. Hence SS is a complete lattice.

Before introducing the abstract operations over SH we need some ancillary definitions.

Definition 2. (Auxiliary functions.) The *closure under union* function $(\cdot)^{\star} \colon SH \to SH$ (also called *star-union*) is given, for each $sh \in SH$, by

$$sh^{\star} \stackrel{\mathrm{def}}{=} \{\, S \in SG \bigm| \exists n \geq 1 \,.\, \exists T_1, \ldots, T_n \in sh \,.\, S = T_1 \cup \cdots \cup T_n \,\}.$$

For each $sh \in SH$ and each $T \in \wp_{\mathrm{f}}(\mathit{Vars})$, the operation of extracting the *relevant component of s with respect to T* is encoded by the function $\mathrm{rel} \colon \wp_{\mathrm{f}}(\mathit{Vars}) \times SH \to SH$ defined as

$$\mathrm{rel}(T, sh) \stackrel{\mathrm{def}}{=} \{\, S \in sh \mid S \cap T \neq \varnothing \,\}.$$

For each $sh_1, sh_2 \in SH$, the *binary union* function $\mathrm{bin} \colon SH \times SH \to SH$ is given by

$$\mathrm{bin}(sh_1, sh_2) \stackrel{\mathrm{def}}{=} \{\, S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2 \,\}.$$

[1] The literature on Sharing is almost unanimous in defining sharing-sets so that they *always* contain the empty set. We deviate from this *de facto* standard: in our approach sharing-sets *never* contain the empty set. We do this because

1. there is no real need of having $\varnothing$ and $\subseteq$ as the bottom element and the ordering of the domain, respectively (the original motivation for including the empty set in each sharing-set);
2. the definitions turn out to be easier;
3. we describe the implementation (where the empty set never appears in sharing-sets) more faithfully.

The function proj: $SH \times \wp_f(Vars) \to SH$ *projects* an element of SH onto a set of variables of interest: if $sh \in SH$ and $V \in \wp_f(Vars)$, then

$$\mathrm{proj}(sh, V) \stackrel{\mathrm{def}}{=} \{\, S \cap V \mid S \in sh, S \cap V \neq \varnothing \,\}.$$

The auxiliary function amgu captures the effects of a binding $x \mapsto t$ on an SH element. Let x be a variable and t a term in which x does not occur. Let also $sh \in SH$ and

$$A \stackrel{\mathrm{def}}{=} \mathrm{rel}\bigl(\{x\}, sh\bigr),$$
$$B \stackrel{\mathrm{def}}{=} \mathrm{rel}\bigl(vars(t), sh\bigr).$$

Then

$$\mathrm{amgu}(sh, x \mapsto t) \stackrel{\mathrm{def}}{=} \bigl(sh \setminus (A \cup B)\bigr) \cup \mathrm{bin}(A^\star, B^\star).$$

It is shown in [15] that amgu is both commutative and idempotent. Thus we can define the extension amgu: $SH \times Subst \to SH$ by

$$\mathrm{amgu}(sh, \varnothing) \stackrel{\mathrm{def}}{=} sh,$$
$$\mathrm{amgu}\bigl(sh, \{x \mapsto t\} \cup \sigma\bigr) \stackrel{\mathrm{def}}{=} \mathrm{amgu}\bigl(\mathrm{amgu}(sh, x \mapsto t), \sigma \setminus \{x \mapsto t\}\bigr).$$

The Sharing domain is given by the complete lattice SS together with the following abstract operations needed for the analysis. Trivial operations, such as consistent renaming of variables, are omitted.

Definition 3. (Abstract operations over SS.) The lub operation over SS is given by the function $\sqcup$: $SS \times SS \to SS$ defined as follows, for each $d, (sh_1, U_1), (sh_2, U_2) \in SS$:

$$\bot \sqcup d \stackrel{\mathrm{def}}{=} d \sqcup \bot \stackrel{\mathrm{def}}{=} d,$$
$$\top \sqcup d \stackrel{\mathrm{def}}{=} d \sqcup \top \stackrel{\mathrm{def}}{=} \top,$$
$$(sh_1, U_1) \sqcup (sh_2, U_2) \stackrel{\mathrm{def}}{=} \begin{cases} (sh_1 \cup sh_2, U_1), & \text{if } U_1 = U_2; \\ \top, & \text{otherwise.} \end{cases}$$

The projection function Proj: $SS \times \wp_f(Vars) \to SS$ is given, for each set of variables of interest $V \in \wp_f(Vars)$ and each description $(sh, U) \in SS$, by

$$\mathrm{Proj}(\bot, V) \stackrel{\mathrm{def}}{=} \bot,$$
$$\mathrm{Proj}(\top, V) \stackrel{\mathrm{def}}{=} \top,$$
$$\mathrm{Proj}\bigl((sh, U), V\bigr) \stackrel{\mathrm{def}}{=} \bigl(\mathrm{proj}(sh, V), U \cap V\bigr).$$

The operation Amgu: $SS \times \mathit{Subst} \to SS$ extends the SS description it takes as an argument, to the set of variables occurring in the substitution it is given as the second argument. Then it applies amgu:

$$\begin{aligned}&\mathrm{Amgu}\bigl((sh, U), \sigma\bigr)\\ &\qquad\stackrel{\mathrm{def}}{=} \Bigl(\mathrm{amgu}\Bigl(sh \cup \{\, \{x\} \mid x \in \mathit{vars}(\sigma) \setminus U \,\}, \sigma\Bigr), U \cup \mathit{vars}(\sigma)\Bigr).\end{aligned}$$

For the distinguished elements $\bot$ and $\top$ of SS we have[2]

$$\begin{aligned}\mathrm{Amgu}(\bot, \sigma) &\stackrel{\mathrm{def}}{=} \bot,\\ \mathrm{Amgu}(\top, \sigma) &\stackrel{\mathrm{def}}{=} \top.\end{aligned} \tag{1}$$

3 Sharing is Redundant for Pair-Sharing

3.1 The Pair-Sharing Property

Let us define the pair-sharing property through a domain that captures it exactly. This domain is similar to Søndergaard's ASub (but without the groundness and linearity information) [17].

Definition 4. (The *pair-sharing* domain.) Let S be a set. Then

$$\mathrm{pairs}(S) \stackrel{\mathrm{def}}{=} \bigl\{\, P \in \wp(S) \bigm| \# P = 2 \,\bigr\}.$$

The *pair-sharing domain* is given by the complete lattice

$$PS \stackrel{\mathrm{def}}{=} \Bigl\{\, (ps, U) \Bigm| U \in \wp_{\mathrm{f}}(\mathit{Vars}), ps \in \wp\bigl(\mathrm{pairs}(U)\bigr) \,\Bigr\} \cup \{\bot, \top\}$$

ordered by $\preceq_{PS}$, which is defined, for each $d, (ps_1, U_1), (ps_2, U_2) \in PS$, by

$$\begin{aligned}\bot &\preceq_{PS} d,\\ d &\preceq_{PS} \top,\\ (ps_1, U_1) \preceq_{PS} (ps_2, U_2) &\quad\iff\quad (U_1 = U_2) \land (ps_1 \subseteq ps_2).\end{aligned}$$

Clearly, PS is a strict abstraction of SS through the abstraction function $\alpha_{PS} \colon SS \to PS$ given, for each $(sh, U) \in SS$, by

$$\begin{aligned}\alpha_{PS}(\bot) &\stackrel{\mathrm{def}}{=} \bot,\\ \alpha_{PS}(\top) &\stackrel{\mathrm{def}}{=} \top,\\ \alpha_{PS}\bigl((sh, U)\bigr) &\stackrel{\mathrm{def}}{=} \bigl(\mathrm{Down}(sh) \cap \mathrm{pairs}(\mathit{Vars}), U\bigr),\end{aligned}$$

[2] Notice that the only reason we have $\top \in SS$ is in order to turn SS into a lattice rather than a CPO. As the description $\top$ is never used in the analysis, Equation 1 is only provided for completeness.

where

$$\mathrm{Down}(sh) \stackrel{\mathrm{def}}{=} \{\, S \in \wp(\mathit{Vars}) \mid \exists T \in sh \,.\, S \subseteq T \,\}.$$

An element of the pair-sharing domain is, roughly speaking, the "end-user image" of the result of the analysis. That is, the only interest of the end-user of our analysis (e.g., the optimizer module of the compiler) is knowing which *pairs* of variables possibly share. The PS domain will be used to measure the accuracy of the other domains in computing pair-sharing.

3.2 What is in Sharing

We now look at the information content of the elements of the Sharing domain. We refer the reader to, e.g., [6] for a formal definition of the concretization function $\gamma\colon SS \to \mathit{Subst} \times \wp_{\mathrm{f}}(\mathit{Vars})$.

As it has been observed by several authors, the SS lattice encodes several properties, besides pair-sharing. We present them here by means of examples that show their usefulness. In what follows, the set of variables of interest is fixed as $U \stackrel{\mathrm{def}}{=} \{x, y, z\}$ and will be omitted from elements of SS. Moreover, the elements of SH will be written in a simplified notation, omitting the inner braces. For example, $\bigl(\{\{x\}, \{x, y\}, \{x, z\}\}, \{x, y, z\}\bigr)$ will be written simply as $\{x, xy, xz\}$.

Groundness. Consider $sh_1 \stackrel{\mathrm{def}}{=} \{xy\}$ and $sh_2 \stackrel{\mathrm{def}}{=} \{xy, z\}$. They encode the same pair-sharing information, namely $\alpha_{PS}(sh_1) = \alpha_{PS}(sh_2) = \{xy\}$. In sh_1 we know that the variable z is ground. This knowledge is useful for pair-sharing detection:

$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_1, x \mapsto z)\bigr) = \alpha_{PS}(\varnothing) = \varnothing,$$
$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_2, x \mapsto z)\bigr) = \alpha_{PS}\bigl(\{xyz\}\bigr) = \{xy, xz, yz\}.$$

Ground dependencies. Let $sh_1 \stackrel{\mathrm{def}}{=} \{xy, xyz\}$ and $sh_2 \stackrel{\mathrm{def}}{=} \{xy, xz, yz\}$. Again, they encode the same pair-sharing information. They also encode the same groundness information (no variable is ground). However, in sh_1 the groundness of y depends on the groundness of x. Let us ground x and see what happens:

$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_1, x \mapsto a)\bigr) = \alpha_{PS}(\varnothing) = \varnothing,$$
$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_2, x \mapsto a)\bigr) = \alpha_{PS}\bigl(\{yz\}\bigr) = \{yz\}.$$

Sharing dependencies. This example is taken from [6]. Let

$$sh_1 \stackrel{\mathrm{def}}{=} \{x, y, z, xyz\},$$
$$sh_2 \stackrel{\mathrm{def}}{=} \{x, y, z, xy, xz, yz\}.$$

They encode the same pair-sharing, groundness, and ground dependency information. Again, let us ground x and look at the results:

$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_1, x \mapsto a)\bigr) = \alpha_{PS}\bigl(\{y, z\}\bigr) = \varnothing,$$
$$\alpha_{PS}\bigl(\mathrm{amgu}(sh_2, x \mapsto a)\bigr) = \alpha_{PS}\bigl(\{y, z, yz\}\bigr) = \{yz\}.$$

In sh_1 the sharing between y and z depends on the (non-) groundness of x, while in sh_2 this is not the case.

Given these three examples, one gets the impression that different elements in SH do encode different information with respect to the pair-sharing property. However, this is not always the case. Consider

$$sh_1 \stackrel{\mathrm{def}}{=} \{x, y, z, xy, xz, yz\},$$
$$sh_2 \stackrel{\mathrm{def}}{=} \{x, y, z, xy, xz, yz, xyz\}.$$

These two different elements do encode the same pair-sharing, groundness, ground dependency, and sharing dependency information. Since the set of variables of interest is $U = \{x, y, z\}$, we can observe that

$$\gamma\bigl((sh_2, U)\bigr) = (Subst, U).$$

What does $\gamma\bigl((sh_1, U)\bigr)$ look like? The only relevant information in sh_1 is that the sharing group xyz is not allowed: sh_1 represents all the idempotent substitutions σ such that

$$vars\bigl(\sigma(x)\bigr) \cap vars\bigl(\sigma(y)\bigr) \cap vars\bigl(\sigma(z)\bigr) = \varnothing.$$

That is, the variables x, y, and z cannot share the *same* variable (but they still can share pairwise). As observed before, this difference is irrelevant from the end-user point of view. Therefore, we want to show that sh_1 and sh_2 are completely equivalent with respect to the pair-sharing property. This is the same as saying that the sharing group xyz is "useless" in sh_2 and can be dropped.

Definition 5. (Redundancy.) Let $sh \in SH$ and $S \in SG$. S is *redundant for sh* if and only if $\# S > 2$ and

$$\mathrm{pairs}(S) = \bigcup \bigl\{\, \mathrm{pairs}(T) \bigm| T \in sh, T \subset S \,\bigr\}.$$

Read it this way: S is redundant for sh if and only if all its sharing pairs can be extracted from the elements of sh that are contained in S. As the name suggests, redundant sharing groups can be dropped. For the moment, as we are walking on theoretical ground, we *add* them so to obtain a sort of *normal form*. We thus define an upper closure operator over SH that induces an equivalence relation over the elements of SH.

Definition 6. (A closure operator on SH.) The function $\rho\colon SH \to SH$ is given, for each $sh \in SH$, by

$$\rho(sh) \stackrel{\text{def}}{=} sh \cup \{\, S \in SG \mid S \text{ is redundant for } sh \,\}.$$

A set S can be added to a sharing set sh without changing the pair-sharing information only if, *for each variable x in S*, every pair in S such as xy, is already in an element in sh. This implies that S must be the union of some of the sets in sh that contain x. This observation leads to the following alternative definition for ρ.

Theorem 7. *If $sh \in SH$ then*

$$\rho(sh) = \Bigl\{\, S \in SG \bigm| \forall x \in S : S \in \operatorname{rel}\bigl(\{x\}, sh\bigr)^\star \,\Bigr\}.$$

Proof. Let us define, for each $sh \in SH$,

$$\dot\rho(sh) \stackrel{\text{def}}{=} \Bigl\{\, S \in SG \bigm| \forall x \in S : S \in \operatorname{rel}\bigl(\{x\}, sh\bigr)^\star \,\Bigr\}.$$

Let $sh \in SH$: we want to show that $\rho(sh) = \dot\rho(sh)$. First suppose $S \in \rho(sh)$. If $S \in sh$, then $S \in \dot\rho(sh)$. Suppose $S \notin sh$. Then as S is redundant for sh, we have $S = \{x, x_1, \ldots, x_n\}$ with $n \geq 2$, and, for each x_i there exists a T_i such that $T_i \in sh$, $T_i \subset S$, and $\{x, x_i\} \subseteq T_i$. Thus $S = T_1 \cup \cdots \cup T_n$. As $T_1, \ldots, T_n \in \operatorname{rel}\bigl(\{x\}, sh\bigr)$, we have $S \in \operatorname{rel}\bigl(\{x\}, sh\bigr)^\star$. Since the choice of $x \in S$ was arbitrary, $S \in \dot\rho(sh)$.

Secondly, suppose $S \in \dot\rho(sh)$. If $S \in sh$, then $S \in \rho(sh)$. Suppose that $S \notin sh$. Then we need to show that S is redundant for sh. That is, we need to show that $\# S > 2$ and

$$\operatorname{pairs}(S) = \bigcup \{\, \operatorname{pairs}(T) \bigm| T \in sh, T \subset S \,\}. \tag{2}$$

By definition of $\dot\rho(sh)$, for each $x \in S$,

$$S = \bigcup \Bigl\{\, T \in sh \bigm| T \subseteq S, x \in T \Bigr\}. \tag{3}$$

Since $S \notin sh$, the case $T = S$ can be ruled out in (3) obtaining

$$S = \bigcup \Bigl\{\, T \in sh \bigm| T \subset S, x \in T \Bigr\} \tag{4}$$

and thus $\# S > 2$. Also, as (4) holds for all $x \in S$,

$$S = \bigcup \{\, T \in sh \bigm| T \subset S \,\}. \tag{5}$$

Thus,

$$\operatorname{pairs}(S) \supseteq \bigcup \{ \operatorname{pairs}(T) \mid T \in sh, T \subset S \}. \tag{6}$$

Suppose $\{x, y\} \in \operatorname{pairs}(S)$ for some $x, y \in \mathit{Vars}$. Then, by (4), there is a $T \in sh$ such that $T \subset S$ and $x, y \in T$ and hence $\{x, y\} \in \operatorname{pairs}(T)$. Hence

$$\operatorname{pairs}(S) \subseteq \bigcup \{ \operatorname{pairs}(T) \mid T \in sh, T \subset S \} \tag{7}$$

Combining (6) and (7) gives (2) as required. □

While the original definition refers directly to the pair-sharing concept, the alternative definition provided by Theorem 7 is very elegant and concise, and turns out to be useful for proving several results.

Abusing notation, we can easily define the overloading $\rho \colon SS \to SS$ such that

$$\begin{aligned} \rho(\bot) &\stackrel{\mathrm{def}}{=} \bot, \\ \rho(\top) &\stackrel{\mathrm{def}}{=} \top, \\ \rho\bigl((sh, U)\bigr) &\stackrel{\mathrm{def}}{=} \bigl(\rho(sh), U\bigr). \end{aligned}$$

We have thus implicitly defined a new domain that we will denote by SS^ρ. The domain SS^ρ is the quotient of SS with respect to the equivalence relation induced by ρ: d_1 and d_2 are equivalent if and only if $\rho(d_1) = \rho(d_2)$. Clearly, SS^ρ is a proper abstraction of SS.

It is straightforward to prove the following.

Proposition 8. *For each $d \in SS$ we have $\alpha_{PS}\bigl(\rho(d)\bigr) = \alpha_{PS}(d)$.*

Thus the addition of redundant elements does not cause any precision loss, as far as pair-sharing is concerned. In other words, SS^ρ is as good as SS for *representing* pair-sharing. Now we show that ρ is a congruence with respect to the operations Amgu, $\sqcup$, and Proj.

Theorem 9. *Let $d_1, d_2 \in SS$. If $\rho(d_1) = \rho(d_2)$ then, for each $\sigma \in \mathit{Subst}$, each $d' \in SS$, and each $V \in \wp_{\mathrm{f}}(\mathit{Vars})$,*

1. $\rho\bigl(\operatorname{Amgu}(d_1, \sigma)\bigr) = \rho\bigl(\operatorname{Amgu}(d_2, \sigma)\bigr)$;
2. $\rho(d' \sqcup d_1) = \rho(d' \sqcup d_2)$; *and*
3. $\rho\bigl(\operatorname{Proj}(d_1, V)\bigr) = \rho\bigl(\operatorname{Proj}(d_2, V)\bigr)$.

As a corollary of the two results above we have that SS^ρ is as good as SS for *propagating* pair-sharing through the analysis process. Not only that. We show that any proper abstraction of SS^ρ is less precise than SS^ρ on computing pair-sharing.

Theorem 10. *For each $d_1, d_2 \in SS$, $\rho(d_1) \neq \rho(d_2)$ implies*

$$\exists \sigma \in \mathit{Subst} \;.\; \alpha_{PS}\bigl(\mathrm{Amgu}(d_1, \sigma)\bigr) \neq \alpha_{PS}\bigl(\mathrm{Amgu}(d_2, \sigma)\bigr).$$

To summarize, the equivalence relation induced by ρ identifies two elements if and only if their behavior in the analysis process is indistinguishable with respect to the pair-sharing property. As a final remark, the technique we use to "extract" from SS the component that is relevant in order to compute pair-sharing is very similar to the one introduced by Cortesi, Filé, and Winsborough in [7], even though the formal definitions are slightly different.

4 *Star-union* is not needed

The theory developed in the previous section has at least one practical consequence: in the definition of the abstract unification for domain SS^{ρ}, the star-union operator can be *safely* replaced by the binary-union operator.

Theorem 11. *For each $sh \in SH$ we have $sh^{\star} = \rho\bigl(\mathrm{bin}(sh, sh)\bigr)$.*

In our opinion, this is a very important result of this research. In the worst-case, the complexity of the star-union operator is exponential in the number of sharing groups of the input, while for the binary-union operator the complexity is quadratic.

Notice that the complexity improvement provided by Theorem 11 comes at a price. In order to test for fixpoint on SS^{ρ}, we cannot perform a simple identity check, because two syntactically different elements can be mapped onto the same element by ρ: a suitable equivalence test is needed. In the worst case, the complexity of this test is bounded by the square of the number of sharing groups, but it is our opinion that it can be implemented quite efficiently (that is, more efficiently than in our current prototype implementation). This brings us to the next section.

5 Experimental Evaluation

The ideas presented in this paper have been experimentally validated in the context of the development of the China analyzer [1]. China is a data-flow analyzer for CLP($\mathcal{H}_{\mathcal{N}}$) languages (i.e., Prolog, CLP($\mathcal{R}$), `clp(FD)` and so forth), $\mathcal{H}_{\mathcal{N}}$ being an extended Herbrand system where the values of a numeric domain $\mathcal{N}$ can occur as leaves of the terms. China, which is written in C++ and Prolog, performs bottom-up analysis deriving information on both call- and success-patterns by means of program transformations and optimized fixpoint computation techniques.

We have analyzed a number of programs using composite domains of the kind Pattern($\mathcal{D}$), where $\mathcal{D}$ is one of our analysis domains and Pattern($\cdot$) [1] is a generic structural domain similar to `Pat`($\Re$) [8,9]. The construction

Pattern($\cdot$) upgrades a domain $\mathcal{D}$ (which must support a certain set of basic operations) with structural information. The resulting domain, where structural information is retained to some extent, is usually much more precise then $\mathcal{D}$ alone. Of course, there is a price to be paid: in the analysis based on Pattern($\mathcal{D}$), the elements of $\mathcal{D}$ that are to be manipulated are often bigger (i.e., they consider more variables) than those that arise in analyses that are simply based on $\mathcal{D}$. The domains $\mathcal{D}$ that we have tried are: straight sharing *à la* Jacobs and Langen (SS), the domain SS^ρ where star-union has been replaced by binary-union (SS^ρ), and the same domain where all the elements are always maximally reduced, that is, they do not contain any redundant sharing-group ($SS^\rho + \mathrm{red}$), plus all the possible combinations of the three domains above with domains for *linearity* and *freeness*. These combinations have been performed following [4].

The experimental results are reported in Tables 1 and 2. Table 1 refers to our three sharing domains either taken alone or in combination with the linearity domain. The result of adding the freeness domain to such combinations is depicted in Table 2. The tables give the analysis time of each program.[3] The computation times have been taken on a Pentium90 machine with 24 MB of RAM running Linux 2.0.29, and the timings are in seconds of user time as provided by the `getrusage` system call. Many of the tested programs have become more or less standard for the evaluation of data-flow analyzers. Notice that for these tests we have switched off all the other domains currently supported by China[4], as well as all the mechanisms, such as widenings, that are used to throttle the complexity of the analysis.

First of all, the results indicate that, from a practical point of view, analyses based on Sharing *without* linearity or freeness do not make any sense: while the overhead for keeping track of these additional properties is quite small, the number of star-unions that can be avoided thanks to the extra information obtained allow for consistent, and sometimes huge, speedups (not to count the increased precision). Exceptions to this rule, such as in the case of the programs `Disj`, `N_and_C`, and `Zebra` are quite rare, and the slowdown involved is always of modest entity. This fact has already been remarked (see, e.g., [13]).

Experimentation also shows that the SS^ρ domain is indeed a good idea. By replacing star-union with binary-union we have, in the worst case, an almost negligible slowdown. In the best case, instead, we obtain significant speedups. It is interesting to observe that these speedups occur when they are most needed, that is for the analysis of programs where SS, with or without linearity, behaves badly. In other words, SS^ρ has a more *stable* behavior: this is no surprise, since we have replaced an algorithm with exponential

---

[3] The current test-suite of China comprises more than 160 programs. Here we give the results only for those programs whose analysis time using Pattern(SS) (without linearity) are above a certain threshold. Notice also that Tables 1 and 2 are sorted on the analysis time with Pattern(SS).

[4] Numerical bounds and relations, groundness, and polymorphic types.

	Without freeness					
	Without linearity			With linearity		
Program	SS	SS^ρ	$SS^\rho + \mathrm{red}$	SS	SS^ρ	$SS^\rho + \mathrm{red}$
`Life`	0.89	0.6	0.67	0.55	0.55	0.61
`Kalah`	1.28	1.08	1.09	1.15	1.16	1.19
`NRev`	1.41	0.67	0.54	0.05	0.05	0.04
`Queens`	2.22	1.21	0.87	0.06	0.06	0.07
`Meta_Qsort`	2.97	1.35	1.08	1.47	0.95	1.27
`Neural`	3.37	1.99	1.21	0.99	0.99	1.01
`Mastermind`	3.39	2.38	2.64	2.19	2.18	2.26
`Browse`	3.78	2.14	2.01	1.15	0.9	1.06
`Disj`	6.34	5.54	5.57	7.44	7.47	7.47
`DNF`	7.56	6.85	6.8	11.14	11.2	11.31
`Boyer`	8.93	5.07	2.65	7.8	5.31	2.3
`SCC`	10.16	7.39	7.56	9.9	8.98	9.49
`Gabriel`	14.11	7.38	4.33	11.57	6.74	3.58
`CS`	16.64	14.03	14.33	29.13	27.74	28.31
`N_and_C`	19.63	10.9	10.3	2.02	1.98	2.45
`Palindrome`	22.61	12.59	4.58	0.27	0.26	0.33
`Zebra`	28.29	26.56	28.14	29.58	29.63	31.5
`Treeorder`	177.01	105.29	35.46	115.97	78.28	19.84
`Peep`	258.64	145.01	48.41	117.09	76.77	27.17
`Parser_DCG`	496.3	316.87	79.75	21.33	18.88	24.57
`Read`	882.83	555.85	104.06	101.46	63.37	32.71
`R_on_P`	∞	∞	∞	2.67	2.69	6.65

Table 1. Experimental results obtained with the CHINA analyzer.

complexity with a quadratic one. This stability is highly desirable for practical data-flow analyzers.

The last indication we can draw from the experimental results is that eliminating the redundant elements from sharing sets requires care. Even though on the toughest programs systematic reduction can give rise to a threefold increase in the analysis speed (`Treeorder`), it can also result in a threefold slowdown (`R_on_P`). The unfavorable case happens when reduction is repeatedly attempted on sharing sets that have few or no redundant elements. We have conducted some experimentation on the use of heuristics in order to trigger the reduction process. Even though the preliminary results we have obtained with this technique are encouraging, we do not present them

	With freeness					
	Without linearity			With linearity		
Program	SS	SS^ρ	$SS^\rho + \text{red}$	SS	SS^ρ	$SS^\rho + \text{red}$
`Life`	0.67	0.6	0.7	0.58	0.57	0.64
`Kalah`	1.17	1.17	1.2	1.15	1.14	1.17
`NRev`	0.07	0.06	0.08	0.04	0.04	0.05
`Queens`	0.32	0.2	0.26	0.06	0.06	0.07
`Meta_Qsort`	1.3	0.72	0.99	1.31	0.88	1.2
`Neural`	1.1	1.01	1.05	0.95	0.96	0.97
`Mastermind`	3.2	2.92	3.18	2.46	2.47	2.58
`Browse`	2.07	1.44	1.66	1.16	0.94	1.11
`Disj`	6.31	6.4	6.41	7.04	7.05	7.15
`DNF`	8.14	8.07	8.02	10.89	10.72	10.85
`Boyer`	8.46	5.4	2.31	6.85	4.58	2.18
`SCC`	7.9	7.51	8.03	8.66	8.63	9.09
`Gabriel`	12.87	6.79	3.6	10.4	5.87	3.32
`CS`	14.02	14.2	14.52	26.77	26.89	27.52
`N_and_C`	22.28	13.03	10.77	1.93	1.91	2.4
`Palindrome`	2.98	1.57	1.37	0.24	0.24	0.3
`Zebra`	29.71	29.94	32.18	27.75	28.06	29.99
`Treeorder`	128.5	80.51	19.83	103.58	67.64	18.87
`Peep`	140.02	85.15	31.65	104.19	66.08	26.19
`Parser_DCG`	14.07	12.37	16.62	19.06	17.33	23.56
`Read`	118.43	68.83	35.22	90.68	55.17	30.89
`R_on_P`	∞	∞	∞	2.33	2.3	6.32

Table 2. Experimental results obtained with the China analyzer (cont'd).

here mostly because we believe that more theoretical work is needed on the subject.

Of course, analyses based on either SS^ρ or $SS^\rho + \text{red}$ require less (sometimes much less) memory than those based on SS. Moreover, our prototype implementation of both SS^ρ and SS^ρ+red is the most obvious one: we believe that there is much room for improvement.

6 Conclusion

We have questioned, apparently for the first time, whether the set-sharing domain Sharing is the most adequate for tracking pair-sharing between program variables. The answer turned out to be negative. We have presented a new domain SS^ρ that is, at the same time, a strict abstraction of SS and as precise as SS on pair-sharing. We have also shown that no abstract domain weaker than SS^ρ can enjoy this last property. This theoretical work has led us to an important practical result: the exponential star-union operation in the abstract unification procedure can be safely replaced by the binary-union operation, which has quadratic complexity. We have presented preliminary experimental results, showing that, in practice, our new domain compares favorably with SS over a wide range of benchmark programs.

Even though space limitations do not allow us to be more precise, it must be stressed that our theoretical results, obtained in this paper for SS, can also be obtained for the combination SS plus *Lin* plus *Free* as described in [4], where *Lin* and *Free* are the usual domains for linearity and freeness.[5] We emphasize that this claim holds for the analysis (domain *and* operators) defined in [4]. It is known that this analysis, though very accurate, is not optimal. A more powerful abstract unification operator has been defined in [10], which exploits some non-trivial interactions between the sharing and the freeness components. When this refined operator is employed, it is no longer true that SS^ρ plus *Lin* plus *Free* is as accurate as SS plus *Lin* plus *Free*. However, our experimentation has revealed that the abstract operator formalized in [10] is characterized by an extremely unfavorable cost/precision ratio.

References

1. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy, March 1997. Printed as Report TD-1/97.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Sharing revisited. Technical Report 97.19, School of Computer Studies, University of Leeds, 1997.
3. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness — All at once. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis, Proceedings of the Third International Workshop*, volume 724 of *Lecture Notes in Computer Science*, pages 153–164, Padova, Italy, 1993. Springer-Verlag, Berlin.
4. M Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In F. S. de Boer and M. Gabbrielli, editors, *Verification and Analysis of Logic Languages, Proceedings of the W2 Post-Conference Workshop, International Conference on Logic Programming*, pages 213–230, Santa Margherita Ligure, Italy, 1994.
5. M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs-and correctness? In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, MIT

[5] Note that when considering freeness information we have to consider accuracy with respect to the freeness property also.

Press Series in Logic Programming, pages 116–131, Budapest, Hungary, 1993. The MIT Press.

6. A. Cortesi and G. Filé. Comparison and design of abstract domains for sharing analysis. In D. Saccà, editor, *Proceedings of the "Eighth Italian Conference on Logic Programming (GULP'93)"*, pages 251–265, Gizzeria, Italy, 1993. Mediterranean Press.
7. A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation for comparing static analyses. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the "1994 Joint Conference on Declarative Programming (GULP-PRODE '94)"*, pages 372–397, Peñíscola, Spain, September 1994. An extended version will appear in *Theoretical Computer Science.*
8. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and software support for abstract domain design: Generic structural domain and open product. Technical Report CS-93-13, Brown University, Providence, RI, 1993.
9. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, 1994.
10. G. Filé. Share $\times$ Free: Simple and correct. Technical Report 15, Dipartimento di Matematica, Università di Padova, December 1994.
11. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165. The MIT Press, Cambridge, Mass., 1989.
12. D. Jacobs and A. Langen. Static analysis of logic programs for independent AND parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.
13. A. King. A synergistic analysis for sharing and groundness which traces linearity. In D. Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 363–378, Edinburgh, UK, 1994. Springer-Verlag, Berlin.
14. A. King and P. Soper. Depth-k sharing and freeness. In P. Van Hentenryck, editor, *Logic Programming: Proceedings of the Eleventh International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 553–568, Santa Margherita Ligure, Italy, 1994. The MIT Press.
15. A. Langen. *Static Analysis for Independent And-Parallelism in Logic Programs.* PhD thesis, University of Southern California, 1990.
16. K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 49–63, Paris, France, 1991. The MIT Press.
17. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of the 1986 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, Berlin, 1986.