Set-Sharing is Redundant for Pair-Sharing ⋆

Roberto Bagnara

Department of Mathematics, University of Parma, I-43100 Parma, Italy,
`bagnara@cs.unipr.it`

Patricia M. Hill

School of Computer Studies, University of Leeds, Leeds, LS2 9JT, U.K.,
`hill@scs.leeds.ac.uk`

Enea Zaffanella

Department of Mathematics, University of Parma, I-43100 Parma, Italy,
`zaffanella@cs.unipr.it`

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 experimental results showing that, in practice, our domain compares favorably with the set-sharing one over a wide range of benchmark and real programs.

Key words: Logic Programming; Data-flow Analysis; Abstract Interpretation; Sharing Analysis.

⋆ This work is a revised and extended version of [3].

[1] Most of the work of R. Bagnara has been conducted while the author was at the School of Computer Studies, University of Leeds, Leeds, LS2 9JT, U.K. His work has been supported by EPSRC under grant GR/L19515.

1 Introduction

1.1 Basic Notions and Motivations

In the execution of a logic program, two variables are *aliased*, at some program point, if they are bound to terms that share a common variable. In logic programming, a knowledge of the possible aliasing between variables has some important applications.

Information about variable aliasing is essential for the efficient exploitation of AND-parallelism [10,38,43,52]. Informally, two atoms in a goal are executed in parallel if, by a mixture of compile-time and run-time checks, it can be guaranteed that they do not share any variable. This implies the absence of *binding conflicts* at run-time, that is, it will never happen that the processes associated to the two atoms try to bind the same variable. Another significant application is known as *occur-check reduction* [28,53,55]. It is well-known that many implemented logic programming languages (in particular, almost all Prolog systems) omit the *occur-check* from the unification procedure. Occur-check reduction amounts to identifying the unifications where such omission is safe, and, for this purpose, information on the possible aliasing of program variables is crucial. Aliasing information can also be used indirectly in the computation of other interesting program properties. For instance, the precision with which freeness information can be computed depends on the precision with which aliasing can be tracked [5,6,12,31,46,47,51].

Notice that, often, it is not a knowledge about possible aliasing that is required but its converse, called "definite independence". Two variables are *independent* if they are bound to terms that have no variables in common. Thus, when an analysis concludes that two variables are not possibly aliased we can deduce that they are *definitely independent*. It is also worth noticing that another property of interest in logic programming is the dual concept of *definite aliasing* [56,59]. Definite aliasing, however, is beyond the scope of this paper.

Before continuing, a brief note on terminology: a variable is *free* if it is unbound, it is *ground* if it is bound to a term containing no variables, it is *linear* if it is free or ground or bound to a term that does not contain multiple occurrences of a variable, it is *compound* if it is bound to a compound term. In logic programming the expression "sharing information" often refers to a mixture of groundness, aliasing, and linearity information, since groundness and linearity are properties that allow a more precise characterization of the sharing of program variables. Thus, what is called "a domain for sharing" usually captures groundness, aliasing, and quite often also linearity. A "sharing analysis" is an analysis based on a sharing domain. Notice that this idiom is nothing more than a historical accident: as we will briefly mention in the sequel, *freeness*, *compoundness*, and other kinds of structural information could also be included in the collective term “sharing information”.

1.2 Historical Remarks

Sharing analysis has a long history and several domains have been presented. We will just mention the more influential ones. Chang [9] proposed to capture variable sharing by classifying each clause variable as either ground or belonging to a “coupling class” of mutually dependent variables. Such a partitioning describes all the substitutions that share no variables across coupling classes and that satisfy the specified groundness conditions. The domain of Chang suffered from a lack of expressivity: both variable aliasing and ground dependencies could be represented with very limited accuracy. As far as ground dependencies are concerned, the domain of Chang was improved by Citrin [11].

Jones and Søndergaard described an abstract domain constituted by sets of pairs of clause variables that might be aliased [44]. An approach that is essentially equivalent was introduced by Debray [29]. Here each clause variable is mapped to the set of variables with which it might share. These domains, compared to those defined by Chang and Citrin, capture the independence of variables with much greater accuracy. The same is not true for groundness dependencies.

The domains that have influenced most of the recent research on sharing analysis are the following:

- the domain ASub of Søndergaard [55], which combines elementary information on groundness, *pair-sharing*, and linearity. The abstract operators were formalized rigorously in [13].
- the domain Sharing of Jacobs and Langen [42,43,47], which is based on the concept of *sharing set* (in contrast with the concept of *sharing pair* adopted by ASub).

While ASub takes advantage of linearity information, Sharing is more accurate in capturing groundness dependencies. See Section 8 for a brief review of the research work that has been devoted to the comparison and combination of ASub with Sharing and to the combination with other domains.

Today, talking about sharing analysis for logic programs is almost the same as talking about the *set-sharing* domain Sharing. 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 [5,6,12,45,46,51], 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 classic challenger, the *pair-sharing* domain ASub.[2] 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 not easy. 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).*

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 high degree of accuracy. 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, as precise as Sharing, as far as *pair-sharing* is concerned[3]. This domain is the subject of this paper.

---

[2] We note in passing that Langen's PhD thesis [47] contains the definition of an extended version of Sharing, called ESharing, which integrates linearity. This fact seems to have escaped the attention of most researchers in the field. See Section 8 for more on this subject.

[3] It is well-known, and we will show it later, that being as precise as Sharing on

The paper is organized as follows. In the next section, we introduce the notation and recall the definition of the abstract domain $\mathsf{Sharing}$. Section 3 recalls the pair-sharing property, while Section 4 presents an intuitive explanation of the information content of $\mathsf{Sharing}$. In Section 5, we show that $\mathsf{Sharing}$ 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 6 explains the practical consequences of these results and shows that the complexity of abstract unification using our domain is polynomial (in the number of sharing groups) compared to the exponential complexity for $\mathsf{Sharing}$. Section 7 gives the experimental results, Section 8 describes some work that is more or less related to ours, and Section 9 concludes the paper. The proofs of the presented results can be found in Appendix A, while some details on the programs used in the experimental evaluation are given in Appendix B.

2 Preliminaries

In this section we introduce some mathematical notation that will be used in the paper, as well as recalling the *set-sharing* domain of Jacobs and Langen [42,43,47].

2.1 Basic Concepts and 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.

A *preorder* $\preceq$ over a set P is a binary relation that is reflexive and transitive. If $\preceq$ is also antisymmetric, then it is called a *partial order*. A partial order $\preceq$ is a *linear order* or *total order* if any two elements are comparable, that is, for each $x, y \in P$, either $x \preceq y$ or $y \preceq x$. A set P equipped with a partial order $\preceq$ is said to be *partially ordered* and sometimes written $\langle P, \preceq \rangle$. Partially ordered sets are also called *posets*. An *increasing chain* over the poset $\langle P, \preceq \rangle$ is a subset X of P such that $\preceq$ is a linear order on X.

Given a poset $\langle P, \preceq \rangle$ and $S \subseteq P$, $y \in P$ is an *upper bound* for S if and only if $x \preceq y$ for each $x \in S$. An upper bound y for S is the *least upper bound* (or lub) of S if and only if, for every upper bound y' for S, $y \preceq y'$. The lub, when it exists, is unique. In this case we write $y = \operatorname{lub} S$. The terms *lower bound* and *greatest lower bound* (or glb) are defined dually. A *complete partial order*,

pair-sharing implies being as precise as $\mathsf{Sharing}$ on groundness.

or simply *cpo*, is a poset such that every increasing chain has a least upper bound.

A poset $\langle L, \preceq \rangle$ such that, for each $x, y \in L$, both $\mathrm{lub}\{x, y\}$ and $\mathrm{glb}\{x, y\}$ exist, is called a *lattice*. In this case, lub and glb are also called, respectively, the *join* and the *meet* operations of the lattice. A *complete lattice* is a lattice $\langle L, \preceq \rangle$ such that every subset of L has both a least upper bound and a greatest lower bound. The *top* element of a complete lattice L, denoted by $\top$, is such that $\top \in L$ and $\forall x \in L : x \preceq \top$. The *bottom* element of L, denoted by $\bot$, is defined dually.

An algebra $\langle L, \wedge, \vee \rangle$ is also called a *lattice* if $\wedge$ and $\vee$ are two binary operations over L that are commutative, associative, idempotent, and satisfy the following *absorption laws*, for each $x, y \in L$: $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$. The two definitions of lattices are equivalent. This can be seen by setting up the isomorphism given by: $x \preceq y \overset{\mathrm{def}}{\iff} x \wedge y = x \overset{\mathrm{def}}{\iff} x \vee y = y$, $\mathrm{glb}\{x, y\} \overset{\mathrm{def}}{=} x \wedge y$, and $\mathrm{lub}\{x, y\} \overset{\mathrm{def}}{=} x \vee y$.

Let $\langle P, \preceq \rangle$ be a poset. A function $f \colon P \to P$ is called *monotonic* if, for each $x, y \in P$, $x \preceq y$ implies $f(x) \preceq f(y)$, whereas it is called *idempotent* if, for each $x \in P$, $f(x) = f\bigl(f(x)\bigr)$.

Given two posets $\langle P^\flat, \preceq^\flat \rangle$ and $\langle P^\sharp, \preceq^\sharp \rangle$, a *Galois connection* is a pair of *monotonic* functions $\alpha \colon P^\flat \to P^\sharp$ and $\gamma \colon P^\sharp \to P^\flat$ such that

$$
\begin{aligned}
&\forall x^\flat \in P^\flat : x^\flat \preceq^\flat \gamma\bigl(\alpha(x^\flat)\bigr), \\
&\forall x^\flat \in P^\flat : \alpha\bigl(\gamma(x^\sharp)\bigr) \preceq^\flat x^\sharp.
\end{aligned}
$$

In this case α and γ are often called, respectively, the *abstraction function* and the *concretization function* of the Galois connection.

A monotone and idempotent self-map $\rho \colon P \to P$ over a poset $\langle P, \preceq \rangle$ is called a *closure operator* (or *upper closure operator*) if it is also *extensive*, namely $\forall x \in P : x \preceq \rho(x)$. If C is a complete lattice, then each upper closure operator ρ over C is uniquely determined by the set of its fixpoints, that is, by its image $\rho(C) \overset{\mathrm{def}}{=} \bigl\{\, \rho(x) \bigm| x \in C \,\bigr\}$. The set of all upper closure operators over a complete lattice C, denoted by $\mathrm{uco}(C)$, form a complete lattice ordered as follows: if $\rho_1, \rho_2 \in \mathrm{uco}(P)$, $\rho_1 \sqsubseteq \rho_2$ if and only if $\rho_2(C) \subseteq \rho_1(C)$. Upper closure operators are often denoted by the sets of their fixpoints. The reader is referred to [37] for an extensive treatment of closure operators.

The symbol *Vars* denotes a denumerable set of variables and $\mathcal{T}_{\mathit{Vars}}$ the set of first-order terms over *Vars*. The set of variables occurring in a syntactic object o is denoted by $\mathit{vars}(o)$. A substitution σ is a total function $\sigma \colon \mathit{Vars} \to \mathcal{T}_{\mathit{Vars}}$

that is the identity almost everywhere; in other words, the *domain* of σ,

$$\mathrm{dom}(\sigma) \stackrel{\mathrm{def}}{=} \bigl\{\, x \in \mathit{Vars} \bigm| \sigma(x) \neq x \,\bigr\},$$

is finite. Substitutions are denoted by the set of their *bindings*, thus σ is identified with $\bigl\{\, x \mapsto \sigma(x) \bigm| x \in \mathrm{dom}(\sigma) \,\bigr\}$. A substitution σ is *idempotent* if $\mathit{vars}\bigl(\sigma(x)\bigr) \cap \mathrm{dom}(\sigma) = \emptyset$ for each $x \in \mathrm{dom}(\sigma)$. The set of all the idempotent substitutions is denoted by *Subst*.

2.2 *The* Sharing *Domain*

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 the definitions turn out to be easier and, moreover, they describe the implementation (where the empty set never appears in sharing sets) more faithfully.

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

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

and let

$$SH \stackrel{\mathrm{def}}{=} \wp(SG).$$

The set-sharing lattice *is given by the set*

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

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[4]. We refer the reader to, for instance, [17] for a formal definition of the concretization function $\gamma \colon SS \to \mathit{Subst} \times \wp_{\mathrm{f}}(\mathit{Vars})$.

[4] Notice that the only reason we have $\top \in SS$ is in order to turn SS into a lattice rather than a cpo. Observe also that the description $\top$ is never used in the analysis.

Before introducing the abstract operations over SS we define all the required functions over SH.

Definition 2 (Functions over SH.) *The* closure under union *function (or* star-union*, as it is also called),* $(\cdot)^\star\colon SH \to SH$*, is given, for each* $sh \in SH$*, by*

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

Observe that $(\cdot)^\star$ *is an upper closure operator over* $\langle SH, \subseteq \rangle$*.*

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

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

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

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

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

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

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

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

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 [40,41] *that* amgu *is both commutative and idempotent. Thus we can define the extension* $\mathrm{amgu}\colon SH \times Subst \to SH$ *by*

$$\begin{aligned} \mathrm{amgu}(sh, \emptyset) &\stackrel{\mathrm{def}}{=} sh, \\ \mathrm{amgu}\bigl(sh, \{x \mapsto t\} \cup \sigma\bigr) &\stackrel{\mathrm{def}}{=} \mathrm{amgu}\bigl(\mathrm{amgu}(sh, x \mapsto t), \sigma \setminus \{x \mapsto t\}\bigr). \end{aligned}$$

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

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

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

The projection function $\mathrm{Proj} \colon SS \times \wp_{\mathrm{f}}(\mathit{Vars}) \to SS$ *is given, for each set of variables of interest $V \in \wp_{\mathrm{f}}(\mathit{Vars})$ and each description $(sh, U) \in SS$, by*

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

The operation $\mathrm{Amgu} \colon SS \times \mathit{Subst} \to SS$ *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*:*

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

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

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

3 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) [55].

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*

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

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

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

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.

4 What Is in Sharing

We now look at the information content of the elements of the Sharing domain. First consider the pair-sharing information.

Pair-sharing. 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

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

where

$$\mathrm{Down}(sh) \stackrel{\mathrm{def}}{=} \bigl\{\, S \in \wp(Vars) \bigm| \exists T \in sh \mathrel{.} S \subseteq T \,\bigr\}.$$

As it has been observed by several authors, the SS lattice encodes several properties besides pair-sharing. We next give examples that show the relevance of these properties with respect to computing the pair-sharing information. In what follows, the set of variables of interest is fixed as $U \stackrel{\text{def}}{=} \{x, y, z\}$ 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,

$$\Bigl(\bigl\{\{x\}, \{x, y\}, \{x, z\}\bigr\}, \{x, y, z\}\Bigr)$$

will be written simply as

$$\{x, xy, xz\}.$$

Groundness. Consider $sh_1 \stackrel{\text{def}}{=} \{xy\}$ and $sh_2 \stackrel{\text{def}}{=} \{xy, z\}$. They encode the same pair-sharing information, namely $\alpha_{PS}(sh_1) = \alpha_{PS}(sh_2) = \{xy\}$. Since z does not occur in any sharing group of sh_1, we know that the variable z is ground. In contrast, in concrete substitutions abstracted by sh_2, z is not necessarily ground. This knowledge is useful for pair-sharing detection:

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

Incidentally, this example constitutes a proof of the fact that any domain as precise as $\mathsf{Sharing}$ on pair-sharing (like the domain that is the subject of this paper) is also as precise as $\mathsf{Sharing}$ on groundness. The proof is by contraposition: lose only one ground variable and precision on pair-sharing is compromised.

Ground dependencies. Let $sh_1 \stackrel{\text{def}}{=} \{xy, xyz\}$ and $sh_2 \stackrel{\text{def}}{=} \{xy, xz, yz\}$. They still encode the same pair-sharing information. They also encode the same groundness information (no variable is ground). However, in contrast to sh_2, x occurs in all sharing groups in sh_1 that contain y. Thus, for sh_1, the groundness of y depends solely on the groundness of x. Let us ground x and see what happens:

$$\begin{aligned}\alpha_{PS}\bigl(\mathrm{amgu}(sh_1, x \mapsto a)\bigr) &= \alpha_{PS}(\emptyset) = \emptyset,\\ \alpha_{PS}\bigl(\mathrm{amgu}(sh_2, x \mapsto a)\bigr) &= \alpha_{PS}\bigl(\{yz\}\bigr) = \{yz\}.\end{aligned}$$

Therefore, a knowledge of ground dependencies is important for pair-sharing detection.

Pair-sharing dependencies. This example is taken from [17]. Let

$$\begin{aligned}sh_1 &\stackrel{\text{def}}{=} \{x, y, z, xyz\},\\ sh_2 &\stackrel{\text{def}}{=} \{x, y, z, xy, xz, yz\}.\end{aligned}$$

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

$$\begin{aligned}\alpha_{PS}\Bigl(\mathrm{amgu}(sh_1, x \mapsto a)\Bigr) &= \alpha_{PS}\Bigl(\{y, z\}\Bigr) = \emptyset,\\ \alpha_{PS}\Bigl(\mathrm{amgu}(sh_2, x \mapsto a)\Bigr) &= \alpha_{PS}\Bigl(\{y, z, yz\}\Bigr) = \{yz\}.\end{aligned}$$

In sh_1, x occurs in all the sharing groups that contain the pair yz. Thus 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.

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

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

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 $sh_2 = \wp(U)$. This means that any sharing is possible, and thus that sh_2 describes all the idempotent substitutions over U. In contrast, the only relevant information in sh_1 is that the sharing group xyz is not allowed: sh_1 represents all the idempotent substitutions σ over U *such that*

$$\mathit{vars}\bigl(\sigma(x)\bigr) \cap \mathit{vars}\bigl(\sigma(y)\bigr) \cap \mathit{vars}\bigl(\sigma(z)\bigr) = \emptyset.$$

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. We will show that sh_1 and sh_2 are completely equivalent with respect to the pair-sharing property and that the sharing group xyz in sh_2 is redundant for pair-sharing.

5 Sharing Is Redundant for Pair-Sharing

In the previous example, we noted that xyz was redundant in sh_2. We now formalize this notion of redundancy.

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 as to obtain a sort of *normal form*. A notable advantage is that we can still use subset inclusion for the ordering. 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*

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

Theorem 7 *The function $\rho\colon SH \to SH$ is indeed an upper closure operator.*

In Definition 5, a sharing group S can be added to a sharing set sh without changing the pair-sharing information if and only if, *for each variable x in S*, every pair such as xy in S is in some sharing group in sh which is also a subset of S. This implies that, for each variable x in S, 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 8 *If $sh \in SH$ then*

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

While the original definition refers directly to the pair-sharing concept, the alternative definition provided by Theorem 8 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

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

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.

Theorem 9 *For each $d \in SS$ we have $\alpha_{PS}\bigl(\rho(d)\bigr) = \alpha_{PS}(d)$.*

Thus the addition of redundant sharing groups 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. We now show that ρ is a congruence with respect to the operations Amgu, $\sqcup$, and Proj.

Theorem 10 *Let $d_1, d_2 \in SS$. If $\rho(d_1) = \rho(d_2)$ then, for each $\sigma \in Subst$, each $d' \in SS$, and each $V \in \wp_f(Vars)$,*

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

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. We also show that any proper abstraction of SS^ρ is less precise than SS^ρ on computing pair-sharing.

Theorem 11 *For each $d_1, d_2 \in SS$, $\rho(d_1) \neq \rho(d_2)$ implies*

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

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.

6 Star-Union Is Not Needed

When moving from the theoretical to the practical ground, the first issue concerns the choice of an actual representation for the elements of SS^ρ, which are the equivalence classes induced by ρ over SS. One possibility is to fix once and for all the representative of each equivalence class. In particular, this would allow for an implementation of the equivalence check as an identity check.

One obvious candidate representative for the class is the image under ρ of any element of the class. By a standard result of the theory of closure operators, this element is the *maximum* element in its class with respect to the lattice ordering. Of course, as far as efficiency is concerned, this would be a really unfortunate choice as, in general, $\#\, \rho(sh)$ is an exponential function of $\#\, sh$.

A much more interesting alternative is made possible by the following result, which shows that all the equivalence classes based on ρ are also endowed with a *minimum* element with respect to the lattice ordering.

Theorem 12 *For all* $sh_1, sh_2 \in SH$,

$$\rho(sh_1) = \rho(sh_2) \implies \rho(sh_1 \cap sh_2) = \rho(sh_2).$$

Not surprisingly, the minimum element is the only element of the equivalence class containing no redundant sharing groups.

Theorem 13 *For all* $sh \in SH$,

$$sh \setminus \{\, S \in SG \mid S \textit{ is redundant for } sh \,\} = \bigcap \Bigl\{\, sh' \in SH \Bigm| \rho(sh') = \rho(sh) \Bigr\}.$$

Using the minimum elements (from now on also called *reduced* elements) as representatives for the equivalence classes would seem the best thing to do: memory occupation and the computational cost would be kept at a minimum. However, it must not be forgotten that reducing a sharing-set (i.e., removing all its redundant sharing groups) has a price. Moreover, the abstract operators on sharing-sets may generate non-reduced elements even from reduced ones.

A different solution to the problem of deciding on the classes' representatives is to allow the implementation to select it dynamically. In this setting, the implementation is left free to choose *any* element of an equivalence class as the representative of the class. Moreover, the implementation can make different choices at different times during the analysis. These choices can be guided by several heuristics, with the objective of finding a good trade-off between the cost of reductions and the benefits of working with smaller, and possibly minimal, elements. One of the consequences of allowing this kind of freedom is that the equivalence check can no longer be implemented as an identity check. This does not constitute a serious drawback: as we will see in Section 7, the complexity of the equivalence check is bounded by the square of the number of sharing groups.

Since the computational complexity of all the abstract operators depends on the cardinality of the sharing-sets involved, a general recipe for efficiency is avoiding, wherever possible, the generation of redundant sharing groups. For this purpose, another very interesting practical consequence of the theory developed in the previous section is that the star-union operator can be *safely* replaced by the binary-union operator.

Theorem 14 *For each* $sh \in SH$ *we have* $sh^\star = \rho\bigl(\mathrm{bin}(sh, sh)\bigr)$.

In words, $\mathrm{bin}(sh, sh)$ is granted to be in the same equivalence class of $sh^\star$ and it is quite likely to contain less redundant sharing groups. Moreover, 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.

This method for computing (a representative of) the star-union can be safely applied in the computation for abstract unification. We prove here the result for amgu when it is applied to a single binding. As Amgu is defined in terms of amgu, the revised definition for amgu can be used in the computation of Amgu.

Theorem 15 *Let x be a variable and t a term in which x does not occur. Let also $sh \in SH$ and*

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

Then

$$\rho\bigl(\operatorname{amgu}(sh, x \mapsto t)\bigr) = \rho\Bigl(\bigl(sh \setminus (A \cup B)\bigr) \cup \operatorname{bin}\bigl(\operatorname{bin}(A, A), \operatorname{bin}(B, B)\bigr)\Bigr).$$

7 Experimental Evaluation

The ideas presented in this paper have been experimentally validated in the context of the development of the China analyzer [2]. 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++, performs bottom-up analysis deriving information on both call-patterns and success-patterns by means of program transformations and optimized fixpoint computation techniques.

7.1 The Implementation

At the implementation level, each variable is associated to a non-negative integer. Thus variables inherit from the integers the usual total ordering relation. Finite sets of variables are represented by dynamically resizing bit-vectors. Sharing-sets, that is sets of sets of variables, are implemented by means of the `set` associative container provided by standard C++. The total ordering relation employed to this purpose, $< \subseteq \wp_\mathrm{f}(\mathbb{N}_0) \times \wp_\mathrm{f}(\mathbb{N}_0)$, is an extension of the $\subset$ partial ordering. In other words, for each $S_1, S_2 \in \wp_\mathrm{f}(\mathbb{N}_0)$, if $S_1 \subset S_2$ then $S_1 < S_2$. This ordering is exploited in several places in the implementation and proved to be a very effective device.

It is important to remark that the implementation of Sharing we use for comparison against SS^ρ is a rather refined one. For instance, care was taken in

Require: the sharing set $sh \stackrel{\text{def}}{=} \{S_1, \ldots, S_n\}$ such that $S_1 < \cdots < S_n$.
Ensure: on exit $sh_{\text{star}} = sh^{\star}$.

1: $sh_{\text{star}} := \emptyset$
2: $sh_{\text{done}} := \emptyset$
3: **for** $i := 1$ to n **do**
4:     **if** $S_i \neq \bigcup \{\, T \in sh_{\text{done}} \mid T \subset S_i \,\}$ **then**
5:         $sh_{\text{done}} := sh_{\text{done}} \cup \{S_i\}$
6:         $sh_{\text{star}} := sh_{\text{star}} \cup \{S_i\} \cup \{\, S_i \cup U \mid U \in sh_{\text{star}} \,\}$
7:     **end if**
8: **end for**

Fig. 1. An optimized algorithm for computing star-union.

the implementation of star-union. The algorithm we used is given in Figure 1. The optimization implemented by lines 2, 4, and 5 avoids the computation of redundant unions, and can give rise to efficiency gains of an order of magnitude and more. Suppose $sh = \{S_1, \ldots, S_n\}$, $i \in \{1, \ldots, n\}$, $J \subset \{1, \ldots, n\}$ with $i \notin J$, and $S_i = \bigcup_{j \in J} S_j$. Then $sh^{\star} = \bigl(sh \setminus \{S_i\}\bigr)^{\star}$. Observe how the total ordering used for representing sharing-sets simplifies the task of checking the applicability condition for this optimization (line 4 of the algorithm in Figure 1). In fact, in order to have $S_i = \bigcup_{j \in J} S_j$ and $i \notin J$ we must have $\forall j \in J : S_j \subset S_i$ and thus $\forall j \in J : S_j < S_i$.

Theorem 16 *On exit from the algorithm of* Figure 1, $sh_{\text{star}} = sh^{\star}$.

As far as the implementation of SS^{ρ} is concerned, the code for all the abstract operations can be reused, once star-union has been replaced by binary union. This does not mean that it is not possible to produce sharing-sets with less redundant sharing groups. For instance, consider the amgu operation applied to a sharing-set sh and a binding $x \mapsto t$. Then, if we let

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

Theorem 15 tells us that

$$\bigl(sh \setminus (A \cup B)\bigr) \cup \operatorname{bin}\bigl(\operatorname{bin}(A, A), \operatorname{bin}(B, B)\bigr) \tag{1}$$

is in the same equivalence class as $\operatorname{amgu}(sh, x \mapsto t)$. We will show that expression (1), even though, in general, produces less redundant sharing groups than $\mathrm{amgu}(sh, x \mapsto t)$, it is not optimal in this respect. Let us define

$$\begin{aligned}
\hat{A} &\stackrel{\mathrm{def}}{=} A \setminus B,\\
\hat{B} &\stackrel{\mathrm{def}}{=} B \setminus A,\\
C &\stackrel{\mathrm{def}}{=} A \cap B.
\end{aligned}$$

Thus $A = \hat{A} \cup C$ and $B = \hat{B} \cup C$. For $sh \in SH$, let us define, for each $n \in \mathbb{N}$,

$$sh^n \stackrel{\mathrm{def}}{=} \{\, S_1 \cup \cdots \cup S_n \mid S_1 \in sh, \ldots, S_n \in sh \,\},$$

so that $sh^1 = sh$, $sh^2 = \mathrm{bin}(sh, sh)$, $sh^3 = \mathrm{bin}(sh, sh^2) = \mathrm{bin}(sh^2, sh)$, and so forth. For $sh_1, sh_2 \in SH$, we have $(sh_1 \cup sh_2)^2 = sh_1^2 \cup sh_2^2 \cup \mathrm{bin}(sh_1, sh_2)$. So, the expansion of expression (1) looks like

$$\begin{aligned}
\cdots \cup \mathrm{bin}\Bigl(\mathrm{bin}(A, A), \mathrm{bin}(B, B)\Bigr) &= \cdots \cup \mathrm{bin}\Bigl(A^2, B^2\Bigr)\\
&= \cdots \cup \mathrm{bin}\Bigl((\hat{A} \cup C)^2, (\hat{B} \cup C)^2\Bigr)\\
&= \cdots \cup \mathrm{bin}(\cdots \cup C^2, \cdots \cup C^2)\\
&= \cdots \cup \cdots \cup C^4.
\end{aligned}$$

However, it is straightforward to show that, for each $n \geq 2$, $\rho(sh^n) = \rho(sh^2)$ and $sh^n \supseteq sh^2$. It is thus clear that expression (1), by computing C^4, may introduce (and, in fact, often introduces) redundant sharing groups. It can be proved that a better way to compute a sharing-set in the same equivalence class of $\mathrm{amgu}(sh, x \mapsto t)$ is given by the following expression:

$$\Bigl(sh \setminus (\hat{A} \cup \hat{B} \cup C)\Bigr) \cup \Bigl(\mathrm{bin}(\hat{A} \cup C, \hat{B}) \cup \mathrm{bin}(\hat{A}, C)\Bigr)^2 \cup C^2. \tag{2}$$

Theoretically speaking, expression (2) is undoubtedly better than expression (1). Nonetheless we were unable, despite our attempts, to obtain a competitive implementation of the abstract mgu operation based on expression (2): the implementation relying on expression (1) followed by the elimination of redundant sharing groups was more efficient. Of course, this does not prove anything, and the question whether even more efficient abstract operations can be obtained remains open.

The algorithm for removing redundant sharing groups is pretty easy. The only important observation is that redundant sharing groups can be removed in any order. In fact, suppose two sharing groups $S, T \in sh$ are redundant for sh and let $sh_1 \stackrel{\mathrm{def}}{=} sh \setminus \{S\}$ and $sh_2 \stackrel{\mathrm{def}}{=} sh \setminus \{T\}$. Then $\rho(sh_1) = \rho(sh_2) = \rho(sh)$ and, by Theorem 12, $\rho(sh_1 \cap sh_2) = \rho\Bigl(sh \setminus \{S, T\}\Bigr) = \rho(sh)$.

It turns out that reduction (i.e., the elimination of redundant sharing groups) and equivalence check fit together very well. Thus we present an algorithm for achieving both at the same time in Figure 2.

Require: the sharing sets $sh_{\mathrm{new}} \stackrel{\mathrm{def}}{=} \{S_1, \ldots, S_n\}$ and $sh_{\mathrm{old}} \stackrel{\mathrm{def}}{=} \{T_1, \ldots, T_m\}$ obtained at the current and previous iteration, respectively. The set sh_{old} is guaranteed not to contain redundant sharing groups. Moreover, we have $S_1 < \cdots < S_n$ and $T_1 < \cdots < T_m$.

Ensure: on exit sh is guaranteed not to contain redundant sharing groups and $\rho(sh) = \rho(sh_{\mathrm{new}})$. Moreover, the Boolean variable *changed* is set to **true** if and only if $sh \neq sh_{\mathrm{old}}$ and hence $\rho(sh_{\mathrm{new}}) \neq \rho(sh_{\mathrm{old}})$.

```
sh := sh_new
changed := false
i := 1
j := 1
while ¬changed and i ≤ n and j ≤ m do
    if S_i = T_j then
        i := i + 1
        j := j + 1
    else if redundant(S_i, sh) then
        sh := sh \ {S_i}
        i := i + 1
    else
        changed := true
    end if
end while
while i ≤ n do
    if redundant(S_i, sh) then
        sh := sh \ {S_i}
    else
        changed := true
    end if
    i := i + 1
end while
```

Fig. 2. An optimized algorithm for applying redundancy elimination and checking whether a fixpoint has been reached at the same time.

The function *redundant*, when given a sharing-set $sh \in SH$ and a sharing group $S_i \in sh$, returns the Boolean value **true** if and only if S_i is redundant in sh. Its implementation is straightforward and matches the condition given in Definition 5. Notice that the implementation of *redundant* benefits from the total ordering used to represent sharing-sets as ordered sequences of sharing groups.

If we assume (as we do in our implementation) that the result of the abstract evaluation of each clause is reduced, then the algorithm of Figure 2 allows to reuse part of the reduction work done at iteration k in order to simplify reduction at iteration $k+1$, and to perform the equivalence check (i.e., the test required to detect whether a local fixpoint has been reached) at the same time. Here, once again, the total ordering among sharing groups proves very useful. The algorithm proceeds as follows: the sharing groups in the sharing-sets obtained at iterations k and $k+1$ (sh_{old} and sh_{new}, respectively) are considered in ascending order (recall that $S_i \subset S_j$ implies $S_i < S_j$ and thus, by contraposition, $S_i \geq S_j$ implies $S_i \not\subset S_j$). Since a sharing group can be "made redundant" only by its proper subsets, and since sh_{old} is reduced, as long as no difference is observed between the sharing groups in sh_{old} and sh_{new}, we know that the sharing groups seen so far are not redundant. Notice that, as a consequence of the analysis process, we have $sh_{\mathrm{old}} \subseteq sh_{\mathrm{new}}$, since sh_{new} is always obtained as $sh_{\mathrm{old}} \cup sh'$ for some $sh' \in SH$ (set theoretic union is the lub on SH). When two different sharing groups are observed there are two possibilities: either the sharing group $S_i \in sh_{\mathrm{new}}$ is redundant, in which case it is eliminated and the algorithm proceeds with the first loop, or S_i is not redundant, in which case we know that the fixpoint has not been reached ($changed := \mathbf{true}$) and the algorithm continues with simple reduction at lines 16–23. The algorithm for reduction only can be obtained by just considering lines 1, 3, 16–18, and 21–23.

7.2 Experimental Results

We have compared the performance of our implementations of SS and SS^ρ on a number of Prolog and CLP program with the China analyzer. The comparison has been done on a very practical setting, so as to be as conclusive as possible. First of all, following several other authors (see, e.g., [45]), we observed in [3] that, from a practical point of view, sharing analysis without freeness or linearity does not make sense. Both these properties allow, in a significant proportion of cases, to dispense with costly operations (such as star-union or binary union) increasing the precision of sharing information at the same time, and this with very little overhead that is repayed by consistent, and sometimes huge, speedups. Moreover, freeness is a useful property in itself. We have thus compared the combination of SS with the usual domains for freeness and linearity with the same combination where SS has been replaced by SS^ρ. These combinations have been performed following [6]. The reader interested in the comparison between plain SS and SS^ρ is referred to [3], where all the possible combinations (with freeness only, with linearity only, with both, and with none of them) are taken into account. Of course, in absence of freeness and/or linearity the analysis based on SS has to perform more star-unions. Since star-union is much more computationally complex than binary union, the lack of freeness and/or linearity is more penalizing for SS than it is for SS^ρ.

The second ingredient of what we called "practical setting" concerns the choice of the programs used as benchmarks. Our comparison involved all the 260 programs in the current test-suite of CHINA. This includes *all* the programs we have found by dredging the Web, and not only the few benchmarks that, unfortunately, are still customary in the field. Some details on the programs, which count several real applications among them, are given in Appendix B. The only programs omitted in the following table are the smallest ones, i.e., analyzable in a few hundredths of a second, plus the Goedel, LogTalk, and QD-Janus compilers and run-time systems. For these latter ones, CHINA answers a plain "don't know": while Goedel and LogTalk contain goals of the form `assert(G)`, where the principal functor of `G` is unknown, QD-Janus makes use of `setarg/3`.[5]

Full timing information is reported in Table 1, where the fixpoint computation time, in seconds, is given for all the programs such that at least one timing was above 0.1 seconds. The number of clauses of the programs are also reported. Table entries containing '$> 10^3$' mean that the outer widening employed in CHINA fired. This widening imposes a time limit (1,000 seconds for the experiments presented here) on the analysis of a program. When this limit is reached CHINA forces a "don't know" result for those call-patterns and success-patterns whose analysis is still incomplete.

The experiments were performed on a PC equipped with an AMD K6-2 clocked at 300MHz and running Linux 2.2.5. The timings are in seconds of user time as provided by the `getrusage` system call. Notice that for these tests we have switched off all the other domains currently supported by CHINA.[6]

Table 1: SS vs SS^ρ: fixpoint computation timings in seconds.

		Goal-Independent		Goal-Dependent	
Program	# cl	SS	SS^ρ	SS	SS^ρ
`action.pl`	174	0.94	0.46	99.01	14.63
`aircraft.pl`	878	0.15	0.18	0.92	0.92
`ann.pl`	224	0.18	0.21	0.61	0.72
`aqua_c.pl`	4662	$> 10^3$	$> 10^3$	$> 10^3$	$> 10^3$

[5] As `setarg/3` has been dropped by SICStus since version 3.6 (and we hope has been/will be dropped by all major implementations) we decided that implementing its support in CHINA was not worthwhile.

[6] Namely, numerical bounds and relations, groundness, compoundness, and polymorphic types.

Table 1: (continued)

		Goal-Independent		Goal-Dependent	
Program	# cl	*SS*	SS^ρ	*SS*	SS^ρ
arch1.pl	215	0.26	0.23		
bmtp.pl	1795	222.83	19.83		
boyer.pl	144	0.04	0.03	0.51	0.29
bp0-6.pl	56	0.07	0.08	0.10	0.09
bryant.pl	99	0.10	0.11	0.29	0.30
bup-all.pl	194	0.16	0.11		
caslog.pl	2914	$> 10^3$	661.24	$> 10^3$	$> 10^3$
cg_parser.pl	382	0.13	0.14		
chasen-all.pl	114	0.03	0.04	1.54	0.23
chat80.pl	2754	248.05	11.54	$> 10^3$	358.64
chat_parser.pl	499	25.21	1.18	$> 10^3$	114.78
chess.pl	198	0.05	0.06	0.33	0.33
cobweb.pl	236	$> 10^3$	0.92		
crip.pl	92	0.06	0.07	0.11	0.10
cugini_ut.pl	326	0.13	0.13		
dpos_an.pl	450	0.13	0.16	0.93	1.03
eliza.pl	178	0.08	0.08	1.14	0.53
ftfsg.pl	265	0.11	0.12	0.09	0.10
ftfsg2.pl	358	0.13	0.15	0.27	0.27
ga.pl	171	0.04	0.05	0.12	0.14
ileanTAP.pl	91	$> 10^3$	33.53	$> 10^3$	$> 10^3$
ime_v2-2-1.pl	51	0.07	0.08	0.26	0.28
jugs.pl	30	0.02	0.03	0.25	0.24
lc.pl	58	0.05	0.06	0.18	0.21
ldl-all.pl	10063	$> 10^3$	$> 10^3$		

Table 1: (continued)

		Goal-Independent		Goal-Dependent	
Program	# cl	*SS*	SS^ρ	*SS*	SS^ρ
`leanTAP.pl`	153	0.08	0.08	62.08	2.37
`lg_sys.pl`	4701	$> 10^3$	640.67		
`linTAP.pl`	98	$> 10^3$	374.87	$> 10^3$	$> 10^3$
`ljt.pl`	101	3.69	0.72	0.14	0.16
`llprover.pl`	511	0.18	0.19	1.56	2.21
`log_interp.pl`	354	0.57	0.53	4.49	2.67
`lojban.pl`	305	663.28	11.30	$> 10^3$	179.44
`metutor.pl`	894	0.22	0.23		
`mixtus-all.pl`	1897	$> 10^3$	22.66		
`nand.pl`	205	$> 10^3$	$> 10^3$	0.41	0.42
`nbody.pl`	97	0.06	0.07	0.30	0.30
`oldchina.pl`	1625	$> 10^3$	5.47	$> 10^3$	$> 10^3$
`parser_dcg.pl`	172	0.11	0.12	0.60	0.64
`peephole1.pl`	292	0.09	0.09	1.10	0.94
`pets_an.pl`	940	$> 10^3$	23.31	$> 10^3$	276.67
`peval.pl`	321	0.36	0.40	299.18	6.94
`plaiclp.pl`	1246	0.67	0.72	0.02	0.03
`pmatch.pl`	483	$> 10^3$	228.17	$> 10^3$	$> 10^3$
`press.pl`	123	0.09	0.12	0.71	0.75
`puzzle.pl`	21	4.40	0.12	63.90	2.56
`quot_an.pl`	536	0.35	0.39	4.30	3.76
`read.pl`	159	0.14	0.14	0.82	0.76
`reducer.pl`	106	0.06	0.08	40.59	4.45
`reg.pl`	455	$> 10^3$	133.28	2.82	0.77
`rubik.pl`	277	0.05	0.05	0.28	0.29

Table 1: (continued)

		Goal-Independent		Goal-Dependent	
Program	# cl	*SS*	SS^ρ	*SS*	SS^ρ
`sax-all.pl`	1991	$> 10^3$	58.13		
`scc.pl`	52	$> 10^3$	210.20	0.15	0.15
`sdda.pl`	93	0.46	0.24	24.91	3.75
`semi.pl`	51	0.03	0.03	0.16	0.16
`sim.pl`	300	$> 10^3$	55.27	$> 10^3$	$> 10^3$
`sim_v5-2.pl`	318	0.17	0.18	0.53	0.52
`simple_an.pl`	136	0.11	0.13	1.07	0.75
`slice-all.pl`	1180	1.35	0.98		
`spsys.pl`	1004	8.22	2.63		
`trs.pl`	112	$> 10^3$	34.10	$> 10^3$	$> 10^3$
`unify.pl`	74	0.07	0.10	0.58	0.82
`warplan.pl`	53	0.04	0.04	0.70	0.72

The blank entries in the goal-dependent columns are for those program whose goal-dependent analysis is pointless. This usually happens because the program contains a procedure call to an unknown procedure (e.g., by means of `call/1`). The CHINA analyzer promptly recognizes these cases and reverts to a goal-independent analysis. This is one of the reasons why focusing only on goal-dependent analyses is, in our opinion, a mistake. The other reason being that the ability of analyzing libraries (like `cugini_ut.pl`) once and for all is desirable and, more generally, so is the separate analysis of different program modules, especially in very large projects. Focusing only on goal-independent analyses is the opposite mistake: goal-dependent analyses, when possible, are more precise than goal-independent ones. For these reasons, we insist in presenting experimental results for both.

Note that in Table 1 we report on the results obtained with Prolog programs only. This is because, while the CLP programs in our test-suite are (unfortunately) quite small, they are also characterized by a high percentage of numeric variables. As numeric variables cannot share with other variables (at least, not in the sense that is of interest in this work), our CLP benchmarks fell under the 0.1 seconds threshold.

As far as the choice of representatives for the SS^{ρ} equivalence classes is concerned, the results presented in Table 1 refer to the case where reduction is performed after each binary union operation, at the end of each clause evaluation, and during the equivalence check.

Experimentation shows clearly that the SS^{ρ} domain is indeed a good idea. The results indicate that by replacing SS with SS^{ρ} we have, in the worst case, a limited slowdown (40% at most, and this is only in cases when SS takes less that 2 seconds). In the best case, instead, we obtain significant speedups (up to three orders of magnitude), the average case being definitely in favor of SS^{ρ}. It is interesting to observe that the speedups occur when they are most needed, that is for the analysis of programs where SS behaves badly. In other words, SS^{ρ} has a much more *stable* behavior: this is no surprise, since, among other things, we have replaced an algorithm with exponential complexity with a quadratic one. This stability is highly desirable for practical data-flow analyzers. Of course, analyses based on SS^{ρ} always require less (often much less) memory than those based on SS.

The observed slowdowns happen when reduction is repeatedly attempted on sharing sets that have few or no redundant sharing groups. As pointed out in [3], the slowdowns can be almost eliminated by not applying reduction after each binary union. With this choice, SS^{ρ} is always more efficient than SS but the maximal speedups obtained are not as high as Table 1, even though they reach an order of magnitude. We have conducted some experimentation on the use of heuristics in order to trigger the reduction process. Simple heuristics, such as performing reduction after binary unions only if the size of the sharing-set involved exceeds a certain threshold, work very well. This is because large sharing-sets, besides being the culprit for bad performance, almost always contain many redundant sharing groups. However, after a careful experimentation, we decided to set aside this line of work on the basis that the observed slowdowns are of really minor importance, both in relative and absolute terms (the largest slowdown observed in the test-suite amounts to 0.65 seconds of CPU time).

We refer the interested reader to [3], where a number of programs are analyzed using composite domains of the kind Pattern($\mathcal{D}$), where $\mathcal{D}$ is one of our analysis domains and Pattern($\cdot$) [2] is a generic structural domain similar to $\mathtt{Pat}(\Re)$ [23,24]. 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 $\mathcal{D}$. Thus, it is not surprising that the results reported in [3] confirm the superiority of SS^{ρ}, even though they refer to a less refined implementation with respect to the one we use now.

The last lesson to be learned from Table 1 is that, even though SS^ρ is an important step towards practical and precise sharing analysis for (constraint) logic programs, it is not the last one. We refer the reader to the very recent work presented in [60], where we propose a family of widenings on SS^ρ that, at the cost of an almost negligible precision loss, allows to achieve the desired goal. As a final remark, it is perhaps interesting to observe that some of the theoretical consequences of the work described in this paper are conveniently exploited in [60].

8 Related Work

8.1 Comparisons and Integrations

In [20] Cortesi, Filé and Winsborough establish that ASub and Sharing are incomparable from the precision point of view. While ASub has its strength in keeping track of linearity, Sharing is more accurate as far as groundness and sharing information is concerned.

The relationship between ASub and Sharing has been studied later by Cortesi and Filé in [17]. They also introduced the domain $\mathtt{Sharing}^{\otimes}$ that is aimed at capturing all the information of ASub and Sharing. In reality, $\mathtt{Sharing}^{\otimes}$ is more precise than the reduced product [25] of ASub and Sharing, as $\mathtt{Sharing}^{\otimes}$ contains the domain Prop [19,49] (or *Pos*, as it is now less ambiguously called [1]) for the propagation of groundness information.

Codish et al. propose a more pragmatic way of integrating the information of ASub and Sharing [15]: performing the analysis with both the domains at the same time, and exchanging information between the two components at each step.

It is interesting to note that the remedy to the only weakness of Sharing with respect to ASub (lack of linearity) was proposed by one of the inventors of Sharing in his PhD thesis [47]. It is really unfortunate that this work, which has anticipated a substantial part of the subsequent research, has remained virtually unknown. Besides the integration of linearity information into Sharing, [47] shows how the aliasing information allows freeness to be computed with a good degree of accuracy. This integration of freeness is, however, not complete, since freeness is not used to improve the aliasing analysis.

The synergy attainable from a real integration between aliasing and freeness information has been pointed out, for the first time, by Muthukumar and Hermenegildo [51]. It must be noted that the relationship between freeness and aliasing is more complex than the relationship between linearity and aliasing. In fact, a knowledge of freeness allows to infer more accurate structural information. The later research works [5,6,12,46] are thus seeking a better integration of aliasing and freeness information. These works reached conclusions that are quite similar to one another, the main differences concerning the complexity and precision of the abstract operators employed.

Codish et al. [12] use systems of abstract equations and a non-deterministic operator that, while granting a high degree of precision, is computationally quite complex. Bruynooghe et al. [5] extend the approach of [12] to linearity, in addition to sharing and freeness. In [6], a different kind of abstract equation is used together with a deterministic operator that uses and computes information on sharing, freeness, and linearity. In addition the operator computes the property of compoundness, although that, however, is never used. (In the version published as technical report [7] compoundness is employed in order to improve freeness and, consequently, also aliasing and linearity.) 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 [6]. [7] We emphasize that this claim holds for the analysis (domain *and* operators) defined in [6]. As we will see in a moment, it is known that this analysis, though very accurate, is not optimal.

King and Soper propose, in [46], the integration of sharing and freeness with a depth-k component [48,54]. King [45] shows also how a more refined tracking of linearity (essentially, pushing linearity at the levels of sharing groups) allows for further precision improvements.

A remarkable piece of work, in terms of elegance and cleanliness, is constituted by [31]. Here Filé is the first to define formally the reduced product between Sharing and *Free* (the usual domain for freeness), identifying the elements of the Cartesian product that are redundant. The important merit of this work is due to the fact that it operates a clear distinction between the benefits of the integration between Sharing and *Free* from those obtainable by the integration of Sharing, *Free*, and some kind of structural information. (Notice that most abstract equation systems that have been proposed in the literature contain a significant amount of structural information, though in a more or less hidden way.)

The abstract unification operator defined in [31] is more powerful than the operator, restricted to sharing and freeness, of [6]. The more refined operator exploits some non-trivial interactions between the sharing and the freeness

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

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 [31] is characterized by an extremely unfavorable cost/precision ratio.

It is interesting to observe that all the works mentioned above use, for the representation of sharing information, the domain of Jacobs and Langen "as is". In the next subsection we review some (more or less) alternative approaches.

8.2 Alternative Domains and Representations

Bruynooghe et al. [8] propose a new domain for sharing and freeness analysis based on the concept of *pre-interpretation* [4]. The domain elements are sets of *domain relations*, where a domain relation is a set of assignments of values from the pre-interpretation to the tuple of variables of interest. Roughly speaking, since it is possible to map a domain relation onto a set of sharing groups (plus freeness), this domain seems to have the same expressive power of the disjunctive completion [26] of the domain of Jacobs and Langen. However, the semantic operators defined in [8] are responsible, in certain cases, for some precision loss, thus making the analyses based on the two domains incomparable. Moreover, it seems that the efficiency of the analysis described in [8] relies on considering, as variables of interest, the tuple of arguments in the head of the (normalized) clause: it is not clear what impact the integration of more accurate structural information could have.

In [30], Fecht proposes a domain derived from Sharing. This domain is obtained by only considering downward closed sharing sets, that is, if a sharing set sh contains the sharing group S, then it also contains the sharing group S' for $S' \subset S$. Downward closed sharing sets can be efficiently represented by means of the sets of their maximal elements. The resulting domain, called $\downarrow \mathbf{JL}$, is, essentially, $PS \otimes Ground$, where '$\otimes$' denotes the reduced product and *Ground* is the simplest domain for groundness [44,50]. In fact, in $\downarrow \mathbf{JL}$, ground dependencies and pair-sharing dependencies are lost. To compensate this loss of precision, Fecht combines $\downarrow \mathbf{JL}$ with *Pos* and then with *Lin*, where *Lin* is the usual, simple domain for linearity. [8] For the analyses based on these combined domains, Fecht reports huge speedups and only a negligible loss of precision, compared to Sharing with or without the addition of linearity.

Codish et al. describe, in [14], an algebraic approach to the sharing analysis of logic programs that is based on *set logic programs*. A set logic program is a logic program in which the terms are sets of variables, and standard unification

[8] It is interesting to note that the reduced product of $\downarrow \mathbf{JL}$ with *Pos* with *Lin* is *exactly* the `ASub`$^+$ domain of [17].

is replaced by a suitable unification for sets based on the notion of *ACI1-unification*. Namely, unification in the presence of an associative, commutative, and idempotent equality theory with a unit element (the empty set). The abstract domain described in [14] is also shown to be isomorphic to Sharing and thus, in view of the present work, it is redundant with respect to pair-sharing. A suggested advantage of the approach proposed in [14] is that the adoption of an *abstract compilation* technique [16,34,39] for the implementation of a sharing analyzer is made easier.

8.3 The Quotient of Abstract Interpretations

After an initial attempt in [20], Cortesi, Filé, and Winsborough defined, in [21], the notion of *quotient of an abstract interpretation with respect to a certain property*. This notion is intended to isolate, in a given abstract domain, those parts that are useful to compute the selected property. Our work can thus be considered as an application of [21] where we take Sharing as the starting domain and pair-sharing (PS) as the property under investigation. However, we chose not to fully adhere to the terminology and the methodology proposed in [21].

In our opinion, the terminology adopted in [21] can be the source of misunderstandings. When the authors talk about the quotient of a domain D "with respect to the property P", it seems that they mean "with respect to the equivalence relation $\equiv_{\alpha_P}$ induced by the property P". Such a relation is defined by

$$d_1 \equiv_{\alpha_P} d_2 \quad \stackrel{\text{def}}{\iff} \quad \forall i \geq 0 : \forall \mu : \alpha_P\bigl(\mu^i(d_1)\bigr) = \alpha_P\bigl(\mu^i(d_2)\bigr), \tag{3}$$

where μ is an arbitrary "derived operator", that is, any expression built from operators and elements of the domain D and involving only one variable. In contrast the, by now *standard*, notion of equivalence relation induced by an abstraction function (see [25]) is formalized as follows:

$$d_1 \equiv_{\alpha_P} d_2 \quad \stackrel{\text{def}}{\iff} \quad \alpha_P(d_1) = \alpha_P(d_2).$$

As far as the overall approach is concerned, let us briefly review the methodology we have followed [9]. For the purpose of the present discussion, let us adopt the closure operator approach to abstract interpretation and let us call SS the "concrete domain" and SS^ρ the "abstract domain". We have looked for an *upper closure operator* over SS that is

[9] All the conceptual devices we have resorted to are part of the classic inheritance of the semantics of programming languages. In this respect, we did not invent anything.

(1) "more concrete" than the upper closure operator associated to PS, and
(2) such that the induced equivalence relation over SS is a congruence relation with respect to all the semantic operators of the domain.

These properties give rise to a *completeness* result of the abstract semantics (SS^ρ) with respect to the concrete semantics (SS). In other words, every pair-sharing captured by SS is also captured by SS^ρ, with their respective operations. Moreover, we established that ρ is the weakest upper closure operator satisfying the above requisites. This property constitutes a *full abstraction* result of the abstract semantics with respect to the property under investigation: if two elements are different in the abstract semantics then there exists a context (i.e., a program, in our case a single substitution) that shows the difference between the two elements in terms of pair-sharing. Once put together, the completeness and full abstraction results mean that the domain SS^ρ we have found is exactly the quotient of SS with respect to PS (using the terminology of [21]).

Comparing the proposed methodology in [21] with our approach, we see that although the results would possibly have been more general, the proofs would have been more complex. However, generality is not a concern for us as we are not proposing, as [21], a *framework* but rather an interesting application to a specific domain and a specific language for which "our" alternative approach works well.

It is interesting to note that, in the view of recent results on abstract domain completeness [35], SS^ρ is the *least fully-complete extension* (*lfce*) of PS with respect to SS. From a *purely theoretical* point of view, the quotient of an abstract interpretation with respect to a property of interest and the least fully-complete extension of an upper closure operator are not equivalent. It is known [21] that the quotient may not exist, while the lfce is always defined (assuming the concrete semantics operators are continuous, as it is almost always the case). However, it is also known [36] that when the quotient exists it is exactly the same as the lfce. Moreover, it should be noted that the quotient will exist as long as we consider a semantics where *at least one* of the domain operators is additive and this is almost always the case (just consider the merge-over-all-paths operator, usually implemented as the lub of the domain). Therefore, for the case considered here, these two approaches to the completeness problem in abstract interpretation are equivalent.

8.4 The Pseudo-Complement of Abstract Interpretations

Another interesting operator over semantic domains is the *pseudo-complement*, which has been introduced recently by Cortesi et al. in [18]. In this case, given a domain D and a property P that is represented in D, the aim is to characterize the information that remains in D after the removal of P. More formally, the pseudo-complement of P in D is the weakest abstraction Q of the domain D such that the reduced product [25] of P and Q (re-) generates D. The pseudo-complement can be used in order to decompose a complex domain into its basic components. In fact, the pseudo-complement can obtain a factorization that avoids the duplication of information in the factors.

One of the most studied domains for the application of the pseudo-complement operator has been Sharing. In particular, in [18] the authors compute the complement of *Def* with respect to Sharing. The result, called $Sharing^{+}$, has been further studied and decomposed in [32], where an attempt has been made to remove the pair-sharing information. Here Filé and Ranzato introduce a new method for computing the pseudo-complement. This is much simpler than the original proposal of [18], as it allows the computation of the pseudo-complement by observing only the *meet-irreducible* elements of the reference domain. As an application, [32] shows a decomposition of Sharing into three components "each representing one of the *elementary* properties that coexist in the elements of Sharing",[10] that is, groundness dependencies, pair-sharing, and set-sharing (the latter intended to be without the previous two). The authors observe that $SS \sim PS = SS$ (where $SS \sim PS$ denotes the pseudo-complement of PS with respect to SS) and conclude that PS is too abstract to be extracted from SS by means of complementation. For a non-trivial decomposition of SS they propose a different (and unnatural) definition for pair-sharing called PS'. The problem outlined above is due to the "information preserving" property of pseudo-complementation, as any factorization obtained in this way is such that the reduced product of the factors give back the original domain. In particular, any factorization of SS encodes the redundant information identified in the present work. Such a problem disappears if one considers SS^{ρ} as the reference domain, thus eliminating any redundant information. In [61] it is proved that we have $SS^{\rho} \sim PS \neq SS^{\rho}$, and hence the natural definition of PS allows for a non-trivial decomposition of SS^{ρ}.[11]

9 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

[10] Op. cit., Section 1.

[11] For the expert: PS does represent *exactly* some meet-irreducible elements of SS^{ρ}.

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 experimental results showing that, in practice, our new domain compares favorably with SS over a wide range of benchmark and real programs.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[2] 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.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In Van Hentenryck [57], pages 53–67.

[4] D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting S-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 432–446, Madrid, Spain, 1994. Springer-Verlag, Berlin.

[5] 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. An extended version is available as Technical Report CW 179, Department of Computer Science, K.U. Leuven, September 1993.

[6] 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.

[7] M. Bruynooghe, M. Codish, and A. Mulkers. A composite domain for freeness, sharing, and compoundness analysis of logic programs. Technical Report CW 196, Department of Computer Science, K.U. Leuven, Belgium, July 1994.

[8] M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A freeness and sharing analysis of logic programs based on a pre-interpretation. In Cousot and Schmidt [27], pages 128–142.

[9] J.-H. Chang. *High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis.* PhD thesis, Computer Science Division (EECS), University of California at Berkeley, 1986. Printed as Report UCB/CSD 86/263.

[10] J.-H. Chang, A. M. Despain, and D. DeGroot. AND-parallelism of logic programs based on a static data dependency analysis. In *Digest of Papers of COMPCON Spring'85*, pages 218–225. IEEE Computer Society Press, 1985.

[11] W. V. Citrin. *Parallel Unification Scheduling in Prolog.* PhD thesis, Computer Science Division (EECS), University of California at Berkeley, 1988. Printed as Report UCB/CSD 88/415.

[12] 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 Press Series in Logic Programming, pages 116–131, Budapest, Hungary, 1993. The MIT Press. An extended version is available as Technical Report CW 161, Department of Computer Science, K.U. Leuven, December 1992.

[13] M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In Furukawa [33], pages 79–93.

[14] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In Van Hentenryck [57], pages 68–82.

[15] 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*, 17(1):28–44, January 1995.

[16] P. Codognet and G. Filé. Computations, abstractions and constraints. In *Proceedings of the Fourth IEEE International Conference on Computer Languages.* IEEE Computer Society Press, 1992.

[17] 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.

[18] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM Transactions on Programming Languages and Systems*, 19(1):7–47, 1997.

[19] A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 322–327, Amsterdam, The Netherlands, 1991. IEEE Computer Society Press.

[20] A. Cortesi, G. Filé, and W. Winsborough. Comparison of abstract interpretations. In M. Kuich, editor, *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, volume

623 of *Lecture Notes in Computer Science*, pages 521–532, Wien, Austria, 1992. Springer-Verlag, Berlin.

[21] 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 has been published as [22].

[22] A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation for comparing static analyses. *Theoretical Computer Science*, 202(1&2):163–192, 1998.

[23] 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.

[24] 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.

[25] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[26] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[27] R. Cousot and D. A. Schmidt, editors. *Static Analysis: Proceedings of the 3rd International Symposium*, volume 1145 of *Lecture Notes in Computer Science*, Aachen, Germany, 1996. Springer-Verlag, Berlin.

[28] L. Crnogorac, A. D. Kelly, and H. Søndergaard. A comparison of three occur check analysers. In Cousot and Schmidt [27], pages 159–173.

[29] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[30] C. Fecht. Efficient and precise sharing domains for logic programs. Technical Report A/04/96, Universität des Saarlandes, Fachbereich 14 Informatik, Saarbrücken, Germany, 1996.

[31] G. Filé. Share × Free: Simple and correct. Technical Report 15, Dipartimento di Matematica, Università di Padova, December 1994.

[32] G. Filé and F. Ranzato. Complementation of abstract domains made easy. In M. Maher, editor, *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press Series in Logic Programming, pages 348–362, Bonn, Germany, 1996. The MIT Press.

[33] K. Furukawa, editor. *Logic Programming: Proceedings of the Eighth International Conference on Logic Programming*, MIT Press Series in Logic Programming, Paris, France, 1991. The MIT Press.

[34] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–247, 1995.

[35] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: a domain perspective. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 231–245, Sydney, Australia, 1997. Springer-Verlag, Berlin.

[36] R. Giacobazzi, F. Ranzato, and F. Scozzari. Complete abstract interpretations made constructive. In J. Gruska and J. Zlatuska, editors, *Proceedings of 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 366–377. Springer-Verlag, Berlin, 1998.

[37] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.

[38] M. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 253–268, Jerusalem, Israel, 1990. The MIT Press.

[39] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

[40] P. M. Hill, R. Bagnara, and E. Zaffanella. The correctness of set-sharing. In G. Levi, editor, *Static Analysis: Proceedings of the 5th International Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 99–114, Pisa, Italy, 1998. Springer-Verlag, Berlin.

[41] P. M. Hill, R. Bagnara, and E. Zaffanella. The correctness of set-sharing. Technical Report 98.03, School of Computer Studies, University of Leeds, 1998.

[42] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press Series in Logic Programming, pages 154–165, Cleveland, Ohio, USA, 1989. The MIT Press.

[43] D. Jacobs and A. Langen. Static analysis of logic programs for independent AND parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.

[44] 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*, chapter 6, pages 123–142. Ellis Horwood Ltd, West Sussex, England, 1987.

[45] 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.

[46] 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.

[47] A. Langen. *Static Analysis for Independent And-Parallelism in Logic Programs.* PhD thesis, Computer Science Department, University of Southern California, 1990. Printed as Report TR 91-05.

[48] K. Marriott and H. Søndergaard. On describing success patterns of logic programs. Technical Report 12, The University of Melbourne, 1988.

[49] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In S. K. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press Series in Logic Programming, pages 531–547, Austin, Texas, USA, 1990. The MIT Press.

[50] C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[51] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In Furukawa [33], pages 49–63. An extended version appeared in [52].

[52] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2&3):315–347, 1992.

[53] D. A. Plaisted. The occur-check problem in Prolog. *New Generation Computing*, 2(4):309–322, 1984.

[54] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[55] 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.

[56] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, Basser Department of Computer Science, University of Sydney, Sydney, Australia, June 1991.

[57] P. Van Hentenryck, editor. *Static Analysis: Proceedings of the 4th International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, Paris, France, 1997. Springer-Verlag, Berlin.

[58] M. Ward. The closure operators of a lattice. *Ann. Math.*, 43(2):191–196, 1942.

[59] G. Weyer and W. Winsborough. Annotated structure shape graphs for abstract analysis of Prolog. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, Proceedings of the Eighth International Symposium*, volume 1140 of *Lecture Notes in Computer Science*, pages 92–106, Aachen, Germany, 1996. Springer-Verlag, Berlin.

[60] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening set-sharing. Quaderno 188, Dipartimento di Matematica, Università di Parma, 1999.

[61] E. Zaffanella, P. M. Hill, and R. Bagnara. Decomposing non-redundant sharing by complementation. Technical Report 99.07, School of Computer Studies, University of Leeds, 1999.

A Proofs

Lemma 17 *Suppose $sh \in SH$. Then S is redundant for $\rho(sh)$ if and only if S is redundant for sh.*

PROOF. Since $sh \subseteq \rho(sh)$, if S is redundant for sh, then S is redundant for $\rho(sh)$.

Suppose that S is redundant for $\rho(sh)$. Then

$$S = \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \in \rho(sh), T \subset S \Big\}.$$

Thus, by definition of ρ,

$$\begin{aligned} S = {} & \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \in sh, T \subset S \Big\} \\ & \cup \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \text{ is redundant for } sh, T \subset S \Big\}. \end{aligned}$$

However, if T is redundant for sh,

$$T = \bigcup \Big\{ \operatorname{Pairs}(U) \Bigm| U \in sh, U \subset T \Big\}$$

and hence, if $T \subset S$,

$$T \subseteq \bigcup \Big\{ \operatorname{Pairs}(U) \Bigm| U \in sh, U \subset S \Big\}.$$

It follows that,

$$\begin{aligned} & \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \text{ is redundant for } sh, T \subset S \Big\} \\ & \qquad \subseteq \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \in sh, T \subset S \Big\}. \end{aligned}$$

Therefore

$$S = \bigcup \Big\{ \operatorname{Pairs}(T) \Bigm| T \in sh, T \subset S \Big\}$$

and so S is redundant for sh. $\square$

Proof of Theorem 7. Monotonicity and extensivity of ρ are direct consequences of the definition. For idempotency, suppose that $sh \in SH$. We show that

$$\rho\big(\rho(sh)\big) = \rho(sh).$$

By definition,

$$\rho\Big(\rho(sh)\Big) = \rho(sh) \cup \Big\{\, S \in SG \;\Big|\; S \text{ is redundant for } \rho(sh) \,\Big\}.$$

Therefore, by Lemma 17,

$$\begin{aligned}\rho\Big(\rho(sh)\Big) &= \rho(sh) \cup \Big\{\, S \in SG \;\Big|\; S \text{ is redundant for } sh \,\Big\} \\ &= \rho(sh).\end{aligned}$$

□

Proof of Theorem 8. Let us define, for each $sh \in SH$,

$$\dot{\rho}(sh) \stackrel{\text{def}}{=} \Big\{\, S \in SG \;\Big|\; \forall x \in S : S \in \operatorname{rel}\Big(\{x\}, sh\Big)^\star \,\Big\}.$$

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 $T_1, \ldots, T_n \in \operatorname{rel}\Big(\{x\}, sh\Big)$, we have $S \in \operatorname{rel}\Big(\{x\}, sh\Big)^\star$. 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

$$\operatorname{pairs}(S) = \bigcup \Big\{\, \operatorname{pairs}(T) \;\Big|\; T \in sh, T \subset S \,\Big\}. \tag{A.1}$$

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

$$S = \bigcup \Big\{\, T \in sh \;\Big|\; T \subseteq S, x \in T \Big\}. \tag{A.2}$$

Since $S \notin sh$, the case $T = S$ can be ruled out in (A.2) obtaining

$$S = \bigcup \Big\{\, T \in sh \;\Big|\; T \subset S, x \in T \Big\}, \tag{A.3}$$

and thus $\# S > 2$. Also, as (A.3) holds for all $x \in S$,

$$S = \bigcup \Big\{\, T \in sh \;\Big|\; T \subset S \,\Big\}. \tag{A.4}$$

Thus,

$$\operatorname{pairs}(S) \supseteq \bigcup \Big\{\, \operatorname{pairs}(T) \;\Big|\; T \in sh, T \subset S \,\Big\}. \tag{A.5}$$

Suppose $\{x, y\} \in \mathrm{pairs}(S)$ for some $x, y \in \mathit{Vars}$. Then, by (A.3), there is a $T \in sh$ such that $T \subset S$ and $x, y \in T$ and hence $\{x, y\} \in \mathrm{pairs}(T)$. Hence

$$\mathrm{pairs}(S) \subseteq \bigcup \bigl\{ \mathrm{pairs}(T) \bigm| T \in sh, T \subset S \bigr\}. \tag{A.6}$$

Combining (A.5) and (A.6) gives (A.1) as required. $\square$

Since both ρ (by Theorem 7) and $(\cdot)^\star$ are upper closure operators it follows that

$$sh_1 \subseteq \rho(sh_2) \iff \rho(sh_1) \subseteq \rho(sh_2), \tag{A.7}$$
$$sh_1 \subseteq sh_2^\star \iff sh_1^\star \subseteq sh_2^\star. \tag{A.8}$$

Lemma 18 *For each $sh \in SH$ and each $V \in \wp_\mathrm{f}(Vars)$,*

$$\rho(sh) \setminus \mathrm{rel}\bigl(V, \rho(sh)\bigr) = \rho\bigl(sh \setminus \mathrm{rel}(V, sh)\bigr).$$

PROOF. By Theorem 8,

$$\begin{aligned} S \in \rho\bigl(sh \setminus \mathrm{rel}(V, sh)\bigr) &\iff \forall x \in S : S = \bigcup \left\{ T \subseteq S \;\middle|\; \begin{array}{l} T \in \mathrm{rel}\bigl(\{x\}, sh\bigr) \\ T \notin \mathrm{rel}\bigl(V, sh\bigr) \end{array} \right\} \\ &\iff S \in \rho(sh) \wedge S \cap V = \emptyset \\ &\iff S \in \rho(sh) \setminus \mathrm{rel}\bigl(V, \rho(sh)\bigr). \end{aligned}$$

$\square$

Lemma 19 *For each $sh_1, sh_2 \in SH$ and each $V \in \wp_\mathrm{f}(Vars)$,*

$$sh_1 \subseteq \rho(sh_2) \implies \mathrm{rel}(V, sh_1)^\star \subseteq \mathrm{rel}(V, sh_2)^\star.$$

PROOF. Suppose $S \in \mathrm{rel}(V, sh_1)$. Then, $S \in sh_1$ and $V \cap S \neq \emptyset$. By the hypothesis, $S \in \rho(sh_2)$. Suppose $x \in V \cap S$. Then, by Theorem 8, we have $S = T_1 \cup \cdots \cup T_k$ where, for each $i = 1, \ldots, k$, $x \in T_i$ and $T_i \in sh_2$. Hence, $T_i \in \mathrm{rel}(V, sh_2)$ for $i = 1, \ldots, k$. Thus $S \in \mathrm{rel}(V, sh_2)^\star$.

The result then follows from (A.8). $\square$

Lemma 20 *Suppose $sh_1, sh_2 \in SH$. Then, for each $\sigma \in Subst$,*

$$\rho(sh_1) = \rho(sh_2) \implies \rho\Bigl(\mathrm{amgu}\bigl(sh_1, \sigma\bigr)\Bigr) = \rho\Bigl(\mathrm{amgu}\bigl(sh_2, \sigma\bigr)\Bigr).$$

PROOF. The proof is by induction on the size of σ. The inductive step, when σ has more than one element, is straightforward. For the base cases, if $\sigma = \emptyset$, the statement is obvious from the definitions. It remains to show that

$$sh_1 \subseteq \rho(sh_2) \implies \mathrm{amgu}\bigl(sh_1, \{x \mapsto t\}\bigr) \subseteq \rho\Bigl(\mathrm{amgu}\bigl(sh_2, \{x \mapsto t\}\bigr)\Bigr).$$

The result then follows from (A.7).

Let $v_x \stackrel{\mathrm{def}}{=} \{x\}$, $v_t \stackrel{\mathrm{def}}{=} \mathit{vars}(t)$. Suppose $S \in \mathrm{amgu}\bigl(sh_1, \{x \mapsto t\}\bigr)$. Then, by definition of amgu,

$$S \in \bigl(sh_1 \setminus \mathrm{rel}(v_x \cup v_t, sh_1)\bigr) \cup \mathrm{bin}\bigl(\mathrm{rel}(v_x, sh_1)^\star, \mathrm{rel}(v_t, sh_1)^\star\bigr).$$

There are two cases:

(1) $S \in sh_1 \setminus \mathrm{rel}(v_x \cup v_t, sh_1)$. Then, by hypothesis, $S \in \rho(sh_2)$. Hence we have $S \in \rho(sh_2) \setminus \mathrm{rel}\bigl(v_x \cup v_t, \rho(sh_2)\bigr)$. Thus, by Lemma 18,

$$S \in \rho\bigl(sh_2 \setminus \mathrm{rel}(v_x \cup v_t, sh_2)\bigr).$$

(2) $S \in \mathrm{bin}\bigl(\mathrm{rel}(v_x, sh_1)^\star, \mathrm{rel}(v_t, sh_1)^\star\bigr)$. Then we must have $S = T \cup R$ where $T \in \mathrm{rel}(v_x, sh_1)^\star$ and $R \in \mathrm{rel}(v_t, sh_1)^\star$. By Lemma 19 we have that $T \in \mathrm{rel}(v_x, sh_2)^\star$ and $R \in \mathrm{rel}(v_t, sh_2)^\star$. Hence,

$$S \in \mathrm{bin}\bigl(\mathrm{rel}(v_x, sh_2)^\star, \mathrm{rel}(v_t, sh_2)^\star\bigr).$$

Combining cases 1 and 2 we obtain

$$S \in \rho\bigl(sh_2 \setminus \mathrm{rel}(v_x \cup v_t, sh_2)\bigr) \cup \mathrm{bin}\bigl(\mathrm{rel}(v_x, sh_2)^\star, \mathrm{rel}(v_t, sh_2)^\star\bigr).$$

Hence as ρ is extensive and monotonic

$$S \in \rho\Bigl(\bigl(sh_2 \setminus \mathrm{rel}(v_x \cup v_t, sh_2)\bigr) \cup \mathrm{bin}\bigl(\mathrm{rel}(v_x, sh_2)^\star, \mathrm{rel}(v_t, sh_2)^\star\bigr)\Bigr),$$

and hence $S \in \rho\Bigl(\mathrm{amgu}\bigl(sh_2, \{x \mapsto t\}\bigr)\Bigr)$. $\square$

Theorem 21 *Suppose $d_1, d_2 \in \mathit{SS}$. Then, for each $\sigma \in \mathit{Subst}$,*

$$\rho(d_1) = \rho(d_2) \implies \rho\Bigl(\mathrm{Amgu}\bigl(d_1, \sigma\bigr)\Bigr) = \rho\Bigl(\mathrm{Amgu}\bigl(d_2, \sigma\bigr)\Bigr).$$

PROOF. There are three cases:

(1) $d_1 = \bot$. Then, by hypothesis, $d_2 = \bot$. Straightforward.

(2) $d_1 = \top$. Then, by hypothesis, $d_2 = \top$. Straightforward.

(3) $d_1 = (sh_1, U)$. Then, by hypothesis, $d_2 = (sh_2, U)$ and $\rho(sh_1) = \rho(sh_2)$. Let $V \stackrel{\mathrm{def}}{=} \mathit{vars}(\sigma) \setminus U$ and $s_V \stackrel{\mathrm{def}}{=} \bigl\{ \{x\} \bigm| x \in V \bigr\}$; by definition of Amgu and ρ we have

$$\begin{aligned}\rho\Bigl(\mathrm{Amgu}\bigl((sh_1, U), \sigma\bigr)\Bigr) &= \rho\Bigl(\Bigl(\mathrm{amgu}\bigl(sh_1 \cup s_V, \sigma\bigr), U \cup V\Bigr)\Bigr) \\ &= \Bigl(\rho\Bigl(\mathrm{amgu}\bigl(sh_1 \cup s_V, \sigma\bigr)\Bigr), U \cup V\Bigr),\end{aligned}$$

and, similarly,

$$\rho\Bigl(\mathrm{Amgu}\bigl((sh_2, U), \sigma\bigr)\Bigr) = \Bigl(\rho\Bigl(\mathrm{amgu}\bigl(sh_2 \cup s_V, \sigma\bigr)\Bigr), U \cup V\Bigr).$$

We conclude the proof by observing that

$$\rho(sh_1) = \rho(sh_2) \implies \rho(sh_1 \cup s_V) = \rho(sh_2 \cup s_V)$$

and then applying Lemma 20. □

Lemma 22 *Suppose $sh_1, sh_2 \in SH$. Then*

$$\rho(sh_1 \cup sh_2) = \rho\bigl(\rho(sh_1) \cup \rho(sh_2)\bigr).$$

PROOF. This is a classical property of upper closure operators [58]. We prove it here for completeness. By monotonicity of ρ,

$$\rho(sh_1) \cup \rho(sh_2) \subseteq \rho(sh_1 \cup sh_2).$$

Hence, by monotonicity and idempotency of ρ,

$$\rho\bigl(\rho(sh_1) \cup \rho(sh_2)\bigr) \subseteq \rho(sh_1 \cup sh_2).$$

By extensiveness of ρ,

$$sh_1 \cup sh_2 \subseteq \rho(sh_1) \cup \rho(sh_2),$$

and hence, by monotonicity of ρ,

$$\rho(sh_1 \cup sh_2) \subseteq \rho\bigl(\rho(sh_1) \cup \rho(sh_2)\bigr).$$

□

Theorem 23 *Suppose that $d_1, d_2 \in SS$. Then, for all $d' \in SS$,*

$$\rho(d_1) = \rho(d_2) \implies \rho(d' \sqcup d_1) = \rho(d' \sqcup d_2).$$

PROOF. There are three cases:

(1) $d_1 = \bot$. Then, by hypothesis, $d_2 = \bot$. Straightforward.
(2) $d_1 = \top$. Then, by hypothesis, $d_2 = \top$. Straightforward.
(3) $d_1 = (sh_1, U)$. Then, by hypothesis, $d_2 = (sh_2, U)$ and $\rho(sh_1) = \rho(sh_2)$. Let $d' \in SS$. Again, if $d' \in \{\bot, \top\}$ the proof is straightforward. Thus, let $d' = (sh', U')$; if $U \neq U'$ then the proof is straightforward, so we consider $U = U'$. By definition of $\sqcup$ we have

$$\begin{aligned} \rho\Bigl((sh', U) \sqcup (sh_1, U)\Bigr) &= \rho\Bigl((sh' \cup sh_1, U)\Bigr) \\ &= \Bigl(\rho(sh' \cup sh_1), U\Bigr), \end{aligned}$$

and, similarly,

$$\rho\Bigl((sh', U) \sqcup (sh_2, U)\Bigr) = \Bigl(\rho(sh' \cup sh_2), U\Bigr).$$

We conclude the proof by applying Lemma 22. □

Lemma 24 *Suppose $sh_1, sh_2 \in SH$. Then, for each $V \in \wp_{\mathrm{f}}(Vars)$,*

$$\rho(sh_1) = \rho(sh_2) \implies \rho\Bigl(\mathrm{proj}(sh_1, V)\Bigr) = \rho\Bigl(\mathrm{proj}(sh_2, V)\Bigr).$$

PROOF. We show that

$$sh_1 \subseteq \rho(sh_2) \implies \mathrm{proj}(sh_1, V) \subseteq \rho\Bigl(\mathrm{proj}(sh_2, V)\Bigr).$$

The result then follows from (A.7).

Suppose $sh_1 \subseteq \rho(sh_2)$ and $S \in \mathrm{proj}(sh_1, V)$. Then, as proj is monotonic, we have $S \in \mathrm{proj}\Bigl(\rho(sh_2), V\Bigr)$. By definition of proj and Theorem 8 there exists $S' \in \rho(sh_2)$ such that $S = S' \cap V$ and

$$\forall x \in S' : \exists T_1, \ldots, T_k \in \mathrm{rel}\Bigl(\{x\}, sh_2\Bigr) \;.\; S = \left(\bigcup_{i=1}^{k} T_i\right) \cap V,$$

hence

$$\forall x \in S : \exists T_1, \ldots, T_k \in \mathrm{rel}\Bigl(\{x\}, sh_2\Bigr) \;.\; S = \left(\bigcup_{i=1}^{k} T_i\right) \cap V,$$

and hence

$$\forall x \in S : \exists T_1, \dots, T_k \in \operatorname{rel}\bigl(\{x\}, sh_2\bigr) \;.\; S = \bigcup_{i=1}^{k} (T_i \cap V).$$

However,

$$\forall x \in S : (T_1 \cap V), \dots, (T_k \cap V) \in \operatorname{rel}\bigl(\{x\}, \operatorname{proj}(sh_2, V)\bigr),$$

and thus $S \in \rho\bigl(\operatorname{proj}(sh_2, V)\bigr)$. $\square$

Theorem 25 *Suppose that $d_1, d_2 \in SS$ and $V \in \wp_{\mathrm{f}}(\mathit{Vars})$. Then*

$$\rho(d_1) = \rho(d_2) \implies \rho\bigl(\operatorname{Proj}(d_1, V)\bigr) = \rho\bigl(\operatorname{Proj}(d_2, V)\bigr).$$

PROOF. There are three cases:

(1) $d_1 = \bot$. Then, by hypothesis, $d_2 = \bot$. Straightforward.
(2) $d_1 = \top$. Then, by hypothesis, $d_2 = \top$. Straightforward.
(3) $d_1 = (sh_1, U)$. Then, by hypothesis, $d_2 = (sh_2, U)$ and $\rho(sh_1) = \rho(sh_2)$. By definition of Proj we have

$$\begin{aligned} \rho\Bigl(\operatorname{Proj}\bigl((sh_1, U), V\bigr)\Bigr) &= \rho\Bigl(\bigl(\operatorname{proj}(sh_1, V), U \cap V\bigr)\Bigr) \\ &= \Bigl(\rho\bigl(\operatorname{proj}(sh_1, V)\bigr), U \cap V\Bigr), \end{aligned}$$

and, similarly,

$$\rho\Bigl(\operatorname{Proj}\bigl((sh_2, U), V\bigr)\Bigr) = \Bigl(\rho\bigl(\operatorname{proj}(sh_2, V)\bigr), U \cap V\Bigr).$$

We conclude the proof by applying Lemma 24. $\square$

Proof of Theorem 10.

(1) Proved as Theorem 21.
(2) Proved as Theorem 23.
(3) Proved as Theorem 25. $\square$

Lemma 26 *Let $\sigma \stackrel{\mathrm{def}}{=} \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, where, for each $i = 1, \dots, n$, t_i is a ground term. Then, for all $sh \in SH$ we have*

$$\operatorname{amgu}(sh, \sigma) = sh \setminus \operatorname{rel}\bigl(\{x_1, \dots, x_n\}, sh\bigr).$$

PROOF. By induction on the cardinality n of σ. If $n = 0$, the statement is obvious. Suppose $n = 1$. Then

$$\begin{aligned} \mathrm{amgu}(sh, x_1 \mapsto t_1) &= sh \setminus \mathrm{rel}\bigl(\{x_1\}, sh\bigr) \cup \mathrm{bin}\Bigl(\mathrm{rel}\bigl(\{x_1\}, sh\bigr)^\star, \mathrm{rel}\bigl(\emptyset, sh\bigr)^\star\Bigr) \\ &= sh \setminus \mathrm{rel}\bigl(\{x_1\}, sh\bigr). \end{aligned}$$

For the inductive step, suppose $n > 1$ and let

$$\sigma' \stackrel{\mathrm{def}}{=} \{x_1 \mapsto t_1, \ldots, x_{n-1} \mapsto t_{n-1}\}.$$

By definition of amgu we have

$$\begin{aligned} \mathrm{amgu}(sh, \sigma) &= \mathrm{amgu}\bigl(sh, \{x_n \mapsto t_n\} \cup \sigma'\bigr) \\ &= \mathrm{amgu}\Bigl(\mathrm{amgu}\bigl(sh, \{x_n \mapsto t_n\}\bigr), \sigma'\Bigr) \\ &= \mathrm{amgu}\Bigl(sh \setminus \mathrm{rel}\bigl(\{x_n\}, sh\bigr), \sigma'\Bigr) \\ &= \Bigl(sh \setminus \mathrm{rel}\bigl(\{x_n\}, sh\bigr)\Bigr) \setminus \mathrm{rel}\Bigl(\{x_1, \ldots, x_{n-1}\}, sh \setminus \mathrm{rel}\bigl(\{x_n\}, sh\bigr)\Bigr) \\ &= sh \setminus \Bigl(\mathrm{rel}\bigl(\{x_n\}, sh\bigr) \cup \mathrm{rel}\Bigl(\{x_1, \ldots, x_{n-1}\}, sh \setminus \mathrm{rel}\bigl(\{x_n\}, sh\bigr)\Bigr)\Bigr) \\ &= sh \setminus \mathrm{rel}\bigl(\{x_1, \ldots, x_n\}, sh\bigr). \end{aligned}$$

□

Theorem 27 *Let $d_1 \stackrel{\mathrm{def}}{=} (sh_1, U)$ and $d_2 \stackrel{\mathrm{def}}{=} (sh_2, U)$ be two elements of SS. Then $\rho(d_1) \neq \rho(d_2)$ implies*

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

PROOF. Suppose $\rho(d_1) \neq \rho(d_2)$. Then it follows that $\rho(sh_1) \neq \rho(sh_2)$. We assume that $S \in \rho(sh_1) \setminus \rho(sh_2)$. (If such an S does not exist we simply swap sh_1 and sh_2.)

Let a be a constant and let

$$\sigma_a \stackrel{\mathrm{def}}{=} \{\, x \mapsto a \mid x \in U \setminus S \,\}.$$

Then, by Lemma 26, for $i = 1$, 2, we define $\mathrm{amgu}(sh_i, \sigma_a) \stackrel{\mathrm{def}}{=} sh_i^S$ where

$$sh_i^S \stackrel{\mathrm{def}}{=} \{\, T \mid T \subseteq S, T \in sh_i \,\}.$$

Suppose first that $\# S \geq 2$. For this case, let $\sigma \stackrel{\mathrm{def}}{=} \sigma_a$. Then,

$$\mathrm{Amgu}\bigl((sh_i, U), \sigma\bigr) = (sh_i^S, U).$$

Since $S \notin \rho(sh_2)$, there exists a pair $P \subseteq S$ such that

$$\forall T \in sh_2 : T \subseteq S \implies P \nsubseteq T.$$

However, by definition of sh_2^S, this is the same as saying $\forall T \in sh_2^S : P \nsubseteq T$ or, equivalently, $P \notin \alpha_{PS}(sh_2^S, U)$. Also, since $S \in \rho(sh_1)$, there exists $T' \in sh_1$ such that $T' \subseteq S$ and $P \subseteq T'$, and, moreover, $T' \in sh_1^S$. Thus

$$\alpha_{PS}(sh_1^S, U) \neq \alpha_{PS}(sh_2^S, U).$$

Second, suppose that $\# S = 1$ and let $U' = U \cup \{z\}$ and $\sigma = \sigma_a \cup \{x \mapsto z\}$ where $S = \{x\}$ and $z \in \mathit{Vars} \setminus U$. By applying the definition of Amgu we obtain, for each $i \in \{1, 2\}$,

$$\begin{aligned}
\mathrm{Amgu}\Big((sh_i, U), \sigma\Big) &= \bigg(\mathrm{amgu}\Big(sh_i \cup \big\{\{z\}\big\}, \sigma\Big), U'\bigg), \\
&= \bigg(\mathrm{amgu}\Big(sh_i^S \cup \big\{\{z\}\big\}, \{x \mapsto z\}\Big), U'\bigg).
\end{aligned}$$

Also, by definition of sh_1^S and sh_2^S, we have $sh_1^S = \big\{\{x\}\big\}$ and $sh_2^S = \emptyset$. Hence

$$\begin{aligned}
\mathrm{Amgu}\Big((sh_1, U), \sigma\Big), &= \Big(\big\{\{x, z\}\big\}, U'\Big), \\
\mathrm{Amgu}\Big((sh_2, U), \sigma\Big) &= (\emptyset, U').
\end{aligned}$$

Thus,

$$\alpha_{PS}\bigg(\mathrm{Amgu}\Big((sh_1, U), \sigma\Big)\bigg) = \Big(\big\{\{x, z\}\big\}, U'\Big)$$

is distinct from

$$\alpha_{PS}\bigg(\mathrm{Amgu}\Big((sh_2, U), \sigma\Big)\bigg) = (\emptyset, U').$$

□

Proof of Theorem 11. We have four cases. If one of the following holds

1. $d_1 = \bot$ or $d_2 = \bot$,
2. $d_1 = \top$ or $d_2 = \top$,
3. $d_1 = (sh_1, U_1)$, $d_2 = (sh_2, U_2)$, and $U_1 \neq U_2$,

then we simply take $\sigma = \emptyset$. The last case,

4. $d_1 = (sh_1, U)$ and $d_2 = (sh_2, U)$,

has been proved as Theorem 27. □

Proof of Theorem 12 From $sh_1 \cap sh_2 \subseteq sh_2$ and the monotonicity of ρ, we have that $\rho(sh_1 \cap sh_2) \subseteq \rho(sh_2)$.

In order to prove the reverse inclusion, we will prove $\rho(sh_1 \cap sh_2) \supseteq sh_1 \cup sh_2$. Then, by ρ monotonicity and idempotency, we obtain

$$\begin{aligned}\rho(sh_1 \cap sh_2) &= \rho\big(\rho(sh_1 \cap sh_2)\big) \\ &\supseteq \rho(sh1 \cup sh2) \\ &\supseteq \rho(sh2).\end{aligned}$$

Let $S \in sh_1 \cup sh_2$. We will prove $S \in \rho(sh_1 \cap sh_2)$ by induction on the cardinality of S.

Let $\# S \leq 2$. As $S \in \rho(sh_1) = \rho(sh_2)$ we have, by definition of ρ, both $S \in sh_1$ and $S \in sh_2$. Thus $S \in sh_1 \cap sh_2$ and the result follows by ρ extensivity.

Let now $\# S = k > 2$. There are three cases:

a) If $S \in sh_1 \cap sh_2$ then the result follows, as before, by ρ extensivity.
b) Let $S \in sh_1 \setminus sh_2$. As $S \in \rho(sh_1) = \rho(sh_2)$, we have $S \in \rho(sh_2) \setminus sh_2$. Thus, by the definition of ρ, $\mathrm{pairs}(S) = \bigcup\big\{ \mathrm{pairs}(T) \bigm| T \in sh_2, T \subset S \big\}$. Note that, for all such T, we have $\# T < k$ and thus, by the inductive hypothesis, $T \in \rho(sh_1 \cap sh_2)$. Hence, by the definition of ρ and ρ idempotency, we have $S \in \rho\big(\rho(sh_1 \cap sh_2)\big) = \rho(sh_1 \cap sh_2)$.
c) The case for $S \in sh_2 \setminus sh_1$ is symmetric to case b) above. □

Lemma 28 *S is redundant for sh if and only if S is redundant for $sh \setminus \{S\}$.*

PROOF. We have that S is redundant for sh if and only if $\# S > 2$ and

$$\mathrm{pairs}(S) = \bigcup\Big\{ \mathrm{pairs}(T) \Bigm| T \in sh, T \subset S \Big\}$$

if and only if $\# S > 2$ and

$$\mathrm{pairs}(S) = \bigcup\Big\{ \mathrm{pairs}(T) \Bigm| T \in sh \setminus \{S\}, T \subset S \Big\}$$

if and only if S is redundant for $sh \setminus \{S\}$. □

Corollary 29 *Let $S \in sh$. Then S is redundant for sh if and only if $\rho(sh) = \rho\big(sh \setminus \{S\}\big)$.*

PROOF. By the monotonicity of ρ we have $\rho(sh) \supseteq \rho\big(sh \setminus \{S\}\big)$. Assume S is redundant for sh. Then by Lemma 28, S is redundant for $sh \setminus \{S\}$ and thus $S \in \rho\big(sh \setminus \{S\}\big)$. By the extensivity of ρ we have also $sh \setminus \{S\} \subseteq \rho\big(sh \setminus \{S\}\big)$, and thus $sh \subseteq \rho\big(sh \setminus \{S\}\big)$. By the monotonicity and idempotency of ρ we can conclude that $\rho(sh) \subseteq \rho\big(sh \setminus \{S\}\big)$

Assume now $\rho(sh) = \rho\big(sh \setminus \{S\}\big)$. Since $S \in sh$ then $S \in \rho(sh) = \rho\big(sh \setminus \{S\}\big)$. Thus S is redundant for $sh \setminus \{S\}$. $\square$

Proof of Theorem 13. Let

$$sh_{\mathrm{red}} \stackrel{\mathrm{def}}{=} sh \setminus \{\, S \in SG \mid S \text{ is redundant for } sh \,\}.$$

We first prove that

$$\rho(sh_{\mathrm{red}}) = \rho(sh). \tag{A.9}$$

To this end, for each $S \in SG$ such that S redundant for sh, let $sh_S \stackrel{\mathrm{def}}{=} sh \setminus \{S\}$ and note that $sh_{\mathrm{red}} = \bigcap\{\, sh_S \mid S \text{ is redundant for } sh \,\}$. By Corollary 29, we have $\rho(sh_S) = \rho(sh)$. Thus we can apply Theorem 12 and obtain $\rho(sh_{\mathrm{red}}) = \rho(\bigcap\{\, sh_S \mid S \text{ is redundant for } sh \,\}) = \rho(sh)$.

Having proved (A.9), we only need to prove

$$sh_{\mathrm{red}} = \bigcap\Big\{\, sh' \in SH \;\Big|\; \rho(sh') = \rho(sh_{\mathrm{red}}) \,\Big\}.$$

The inclusion $sh_{\mathrm{red}} \supseteq \bigcap\Big\{\, sh' \in SH \;\Big|\; \rho(sh') = \rho(sh_{\mathrm{red}}) \,\Big\}$ is obvious, since sh_{red} is one of the sets that are intersected in the right hand side.

For the reverse inclusion, let $sh' \in SH$ such that $\rho(sh') = \rho(sh_{\mathrm{red}})$. We have:

$$
\begin{array}{lll}
S \in sh_{\mathrm{red}} & & \\
\quad\iff S \in sh \setminus \{\, T \in SG \mid T \text{ is redundant for } sh \,\} & & \text{by def. of } sh_{\mathrm{red}} \\
\quad\iff S \in \rho(sh) \setminus \{\, T \in SG \mid T \text{ is redundant for } sh \,\} & & \text{by def. of } \rho \\
\quad\iff S \in \rho(sh) \setminus \Big\{\, T \in SG \;\Big|\; T \text{ is redundant for } \rho(sh) \,\Big\} & & \text{by Lemma 17} \\
\quad\iff S \in \rho(sh') \setminus \Big\{\, T \in SG \;\Big|\; T \text{ is redundant for } \rho(sh') \,\Big\} & & \text{as } \rho(sh) = \rho(sh') \\
\quad\iff S \in sh' \setminus \Big\{\, T \in SG \;\Big|\; T \text{ is redundant for } \rho(sh') \,\Big\} & & \text{by def. of } \rho \\
\quad\iff S \in sh' \setminus \{\, T \in SG \mid T \text{ is redundant for } sh' \,\} & & \text{by Lemma 17} \\
\quad\implies S \in sh' & &
\end{array}
$$

Hence $sh_{\mathrm{red}} \subseteq \bigcap\Big\{\, sh' \in SH \;\Big|\; \rho(sh') = \rho(sh_{\mathrm{red}}) \,\Big\}$. $\square$

Proof of Theorem 14. The fact that $sh^\star \supseteq \rho\bigl(\mathrm{bin}(sh, sh)\bigr)$ follows from Theorem 8 and the definition of bin. We now prove that $sh^\star \subseteq \rho\bigl(\mathrm{bin}(sh, sh)\bigr)$. Let $S \in sh^\star$. Then

$$\exists T_1, \dots, T_n \in sh \;.\; S = \bigcup_{i=1}^{n} T_i, \qquad \text{with } n \geq 1.$$

If $S \in \mathrm{bin}(sh, sh)$, then, by definition, $S \in \rho\bigl(\mathrm{bin}(sh, sh)\bigr)$, as required. Suppose $S \notin \mathrm{bin}(sh, sh)$. Then $\# S > 1$ and there exists $\{x, y\} \in \mathrm{pairs}(S)$. Then there must exist $i, j \in \{1, \dots, n\}$ (i and j need not be distinct) such that $x \in T_i$ and $y \in T_j$. This implies $\{x, y\} \in \mathrm{pairs}(T_i \cup T_j)$. However, $T_i \cup T_j \in \mathrm{bin}(sh, sh)$. Hence, as $S \notin \mathrm{bin}(sh, sh)$, $T_i \cup T_j \subset S$. Since the choices of $\{x, y\} \in \mathrm{pairs}(S)$ and $i, j \in \{1, \dots, n\}$ such that $x \in T_i$ and $y \in T_j$ were arbitrary, S is redundant for $\mathrm{bin}(sh, sh)$. $\square$

Lemma 30 *For each $sh_1, sh_2 \in SH$,*

$$\rho\bigl(\mathrm{bin}(sh_1, sh_2)\bigr) = \rho\Bigl(\mathrm{bin}\bigl(\rho(sh_1), \rho(sh_2)\bigr)\Bigr).$$

PROOF. By the monotonicity of bin and ρ, we have

$$\rho\bigl(\mathrm{bin}(sh_1, sh_2)\bigr) \subseteq \rho\Bigl(\mathrm{bin}\bigl(\rho(sh_1), \rho(sh_2)\bigr)\Bigr).$$

Thus, we must show that

$$\rho\Bigl(\mathrm{bin}\bigl(\rho(sh_1), \rho(sh_2)\bigr)\Bigr) \subseteq \rho\bigl(\mathrm{bin}(sh_1, sh_2)\bigr).$$

Since ρ is monotonic and idempotent, we just need to show that

$$\mathrm{bin}\bigl(\rho(sh_1), \rho(sh_2)\bigr) \subseteq \rho\bigl(\mathrm{bin}(sh_1, sh_2)\bigr).$$

Using Theorem 8 we have:

$$\begin{aligned}
&S \in \mathrm{bin}\bigl(\rho(sh_1), \rho(sh_2)\bigr)\\
&\quad\iff S = S_1 \cup S_2 \text{ where } S_1 \in \rho(sh_1) \text{ and } S_2 \in \rho(sh_2)\\
&\quad\iff S = S_1 \cup S_2 \text{ where, for each } i = 1,\, 2,\\
&\qquad\qquad \forall x \in S_i : S_i = \bigcup \Bigl\{\, T_i \subseteq S \Bigm| T_i \in \mathrm{rel}\bigl(\{x\}, sh_i\bigr) \,\Bigr\}\\
&\quad\implies \forall x \in S : S = \bigcup \Bigl\{\, T \subseteq S \Bigm| T \in \mathrm{rel}\bigl(\{x\}, \mathrm{bin}(sh_1, sh_2)\bigr) \,\Bigr\}\\
&\quad\iff S \in \rho\bigl(\mathrm{bin}(sh_1, sh_2)\bigr).
\end{aligned}$$

$\square$

Proof of Theorem 15. By Definition 2 for amgu,

$$\rho\Big(\mathrm{amgu}(sh, x \mapsto t)\Big) = \rho\Big(\big(sh \setminus (A \cup B)\big) \cup \mathrm{bin}(A^\star, B^\star)\Big).$$

So that, by Lemma 22,

$$\rho\Big(\mathrm{amgu}(sh, x \mapsto t)\Big) = \rho\Big(\rho\big(sh \setminus (A \cup B)\big) \cup \rho\big(\mathrm{bin}(A^\star, B^\star)\big)\Big).$$

However, by Theorem 14,

$$\mathrm{bin}(A^\star, B^\star) = \mathrm{bin}\Big(\rho\big(\mathrm{bin}(A, A)\big), \rho\big(\mathrm{bin}(B, B)\big)\Big).$$

Thus, by Lemma 30 and the idempotency of ρ,

$$\rho\Big(\mathrm{bin}(A^\star, B^\star)\Big) = \rho\Big(\mathrm{bin}\big(\mathrm{bin}(A, A), \mathrm{bin}(B, B)\big)\Big).$$

Therefore, by Lemma 22,

$$\rho\Big(\mathrm{amgu}(sh, x \mapsto t)\Big) = \rho\Big(\big(sh \setminus (A \cup B)\big) \cup \mathrm{bin}\big(\mathrm{bin}(A, A), \mathrm{bin}(B, B)\big)\Big).$$

□

Lemma 31 *Let $sh = sh_1 \cup \{S\}$. Then $sh^\star = sh_1^\star \cup \{S\} \cup \{\, S \cup T \mid T \in sh_1^\star \,\}$.*

PROOF. We start by proving that $sh^\star \supseteq sh_1^\star \cup \{S\} \cup \{\, S \cup T \mid T \in sh_1^\star \,\}$. By monotonicity of $(\cdot)^\star$ we have $sh_1^\star \subseteq sh^\star$, whereas, by extensivity of $(\cdot)^\star$, we have $\{S\} \subseteq sh^\star$. Let $T \in sh_1^\star$. Then, by definition of $(\cdot)^\star$, there exist $S_1, \dots, S_n \in sh_1$ such that $T = S_1 \cup \cdots \cup S_n$. Thus $S \cup T = S \cup S_1 \cup \cdots \cup S_n \in sh^\star$.

We now prove that $sh^\star \subseteq sh_1^\star \cup \{S\} \cup \{\, S \cup T \mid T \in sh_1^\star \,\}$. Let $S' \in sh^\star$. Then there exist $S_1, \dots, S_n \in sh$ such that $S' = S_1 \cup \cdots \cup S_n$.

Suppose first $n = 1$ and $S_1 = S$. Then $S' = S$.

Suppose now $n > 1$ and there exists $i \in \{1, \dots, n\}$ such that $S_i = S$. Then, if we let

$$T = \bigcup_{\substack{j=1 \\ j \neq i}}^{n} S_j,$$

we have $S' = S \cup T$ and $T \in sh_1^\star$, thus $S' \in \{\, S \cup T \mid T \in sh_1^\star \,\}$.

Finally, if $\nexists i \in \{1, \dots, n\} \,.\, S_i = S$, then $S' \in sh_1^\star$. □

Lemma 32 *Let* $sh_k = \{S_1, \ldots, S_k\}$ *for* $k = 1, \ldots, n$. *Let also* sh^k_{star} *and* sh^k_{done} *denote the values of the variables* sh_{star} *and* sh_{done}, *respectively, just after the k-th evaluation of the* **if** *statement (i.e., after line 7) in the algorithm of Figure 1. Then* $sh^\star_k \subseteq sh^k_{\mathrm{star}}$ *and* $sh^k_{\mathrm{done}} \subseteq sh_k$.

PROOF. We reason by induction on k. For the case where $k = 1$ we have $sh_k = \{S_1\} = sh^\star_k = sh^k_{\mathrm{star}} = sh^k_{\mathrm{done}}$.

Next assume $1 < k \leq n$. Then $sh_k = sh_{k-1} \cup \{S_k\}$.

Suppose $S_k \neq \bigcup\{\, S_j \in sh_{k-1} \mid S_j \subset S_k \,\}$. By the inductive hypothesis, since $sh^{k-1}_{\mathrm{done}} \subseteq sh_{k-1}$, this implies that the **if** condition in line 4 evaluates to *true* and lines 5 and 6 are executed. Clearly, after the execution of line 5, we have $sh^k_{\mathrm{done}} \subseteq sh_{k-1} \cup \{S_k\} = sh_k$. Moreover, after execution of line 6, we have $sh^k_{\mathrm{star}} = sh^{k-1}_{\mathrm{star}} \cup \{S_k\} \cup \{\, S_k \cup T \mid T \in sh^{k-1}_{\mathrm{star}} \,\}$. By the inductive hypothesis, this implies $sh^k_{\mathrm{star}} \supseteq sh^\star_{k-1} \cup \{S_k\} \cup \{\, S_k \cup T \mid T \in sh^\star_{k-1} \,\}$. Thus, by Lemma 31, $sh^\star_k \subseteq sh^k_{\mathrm{star}}$.

Suppose now $S_k = \bigcup\{\, S_j \in sh_{k-1} \mid S_j \subset S_k \,\}$. Then, $sh^k_{\mathrm{done}} = sh^{k-1}_{\mathrm{done}}$. By the inductive hypothesis, $sh^{k-1}_{\mathrm{done}} \subseteq sh_{k-1}$. Hence $sh^k_{\mathrm{done}} \subseteq sh_k$. We now show that $sh^\star_k = sh^\star_{k-1}$. Clearly, by monotonicity of $(\cdot)^\star$ we have $sh^\star_k \supseteq sh^\star_{k-1}$. Assume now $T \in sh^\star_k$, i.e., $T = T_1 \cup \cdots \cup T_m$ where $T_i \in sh_k$ for $i = 1, \ldots, m$. If for no $j \in \{1, \ldots, m\}$ we have $T_j = S_k$, then $T \in sh^\star_{k-1}$. On the other hand, if there exists $j \in \{1, \ldots, m\}$ such that $T_j = S_k$, then

$$\begin{aligned}
T &= T_1 \cup \cdots \cup T_{j-1} \cup T_j \cup T_{j+1} \cup \cdots \cup T_m \\
&= T_1 \cup \cdots \cup T_{j-1} \cup \bigcup\{\, S_j \in sh_{k-1} \mid S_j \subset S_k \,\} \cup T_{j+1} \cup \cdots \cup T_m \\
&\in sh^\star_{k-1}.
\end{aligned}$$

The inductive hypothesis implies that $sh^{k-1}_{\mathrm{star}} \supseteq sh^\star_{k-1} = sh^\star_k$. This concludes the proof, as the algorithm only adds to sh_{star} and thus $sh^k_{\mathrm{star}} \supseteq sh^{k-1}_{\mathrm{star}}$. □

Proof of Theorem 16. An invariant of the algorithm is that if $S \in sh_{\mathrm{star}}$ then $S = \bigcup_{i \in I} S_i$ for some $I \subseteq \{1, \ldots, n\}$ with $I \neq \emptyset$. The invariant is established in line 1 and preserved by any step. In particular, line 6, which is the only one to change sh_{star}, maintains the invariant. Thus, at the end of the algorithm, if S is an element of sh_{star} then $S \in sh^\star$, hence $sh_{star} \subseteq sh^\star$. The reverse inclusion is trivially satisfied for $sh = \emptyset$, while it is proven by Lemma 32 otherwise. □

B The Tested Programs

Several of the tested programs have become more or less standard for the evaluation of data-flow analyzers. Others, the more interesting ones, are real applications whose analysis has never before been reported in the literature. The suite comprises the following programs: [12]

`action.pl`: an interpreter for *action* semantics written by S. Diehl.

`aircraft.pl`: a program for reasoning on aircraft routes and profiles (author unknown).

`ann.pl`: a simplified clause annotator by M. Hermenegildo, R. Warren, and M. Muthukumar.

`aqua_c.pl`: the Aquarius Prolog compiler, by P. Van Roy.

`arch1.pl`: a machine learning program implementing Winston's incremental learning procedure, by S. Wrobel.

`bmtp.pl`: a sophisticated Boyer-Moore's theorem prover, apparently by H. Fujita, copyrighted by the Institute for New Generation Computer Technology.

`boyer.pl`: a Boyer-Moore theorem prover written by E. Tick after the Lisp version by R. P. Gabriel. `bp0-6.pl`a graphs search program by E. Tick

`bryant.pl`: a Prolog implementation of ROBDDs by P. Schachte.

`bup-all.pl`: the BUP system, a parser generator from Definite Clause Grammars by Y. Matsumoto.

`caslog.pl`: a semi-automatic complexity analysis system for logic programs, by N.-W. Lin.

`cg_parser.pl`: a natural deduction CG parser with semantics, by B. Carpenter.

`chasen-all.pl`: ChaSen version 1.51, a Japanese morphological analysis system, by Y. Den, O. Imaichi, Y. Matsumoto, and T. Utsuro.

`chat80.pl`: a famous query answering system by F. Pereira and D. H. D. Warren.

`chat_parser.pl`: a parser for a set of English sentences taken from Chat80.

`chess.pl`: a Prolog chess program, originally the result of the Artificial Intelligence Project at the "Computer-Club der RWTH Aachen" (KI — Gruppe 89/90), ported to standard Prolog by M. Ostermann.

`cobweb.pl`: a machine learning program implementing Gennari's incremental concept formation algorithm, by J.-U. Kietz.

`crip.pl`: a program to compute counter-models to intuitionistic propositional formulae using a calculus without loop-checking, by L. Pinto and R. Dyckhoff (based on a paper by the same authors at the Symposia Gaussiana conference, Munich, 1993).

`cugini_ut.pl`: the so-called "Cugini Utilities", by J. Cugini.

[12] We did our best to credit all the authors of the programs listed. However, some programs in the test-suite do not give any indication in this respect.

dpos_an.pl: a (buggy version of a) data-flow analyzer for groundness employing *Pos* and some abstractions of it, by A. King.

eliza.pl: a Prolog version by V. Patel of the famous Eliza program.

ftfsg.pl: a compiler and parser for flexible typed feature structure grammars, by M. Dahllof.

ftfsg2.pl: a compiler and parser for flexible typed feature structure grammars with a graphical interface, by M. Dahllof.

ga.pl: a simple genetic algorithm implemented in Prolog.

ileanTAP.pl: an intuitionistic theorem prover by J. Otten.

ime_v2-2-1.pl: a program by L. Zhuhai for solving linear multiple equations using Gauss method.

jugs.pl: a program to solve the "Jugs" puzzle, i.e., given two jugs of known capacities, you have to obtain a specified amount by filling jugs, emptying jugs, and pouring the contents of one jug into the other jug.

lc.pl: a theorem prover for propositional Dummett logic LC, i.e., intuitionistic logic plus $(A \to B) \vee (B \to A)$, written by R. Dyckhoff.

ldl-all.pl: version 3.4.7 of LDL, the Language Development Laboratory, copyrighted by the University of Rostock, Germany.

leanTAP.pl: a tableau-based theorem prover for formulae in negation normal form by B. Beckert and J.Posegga.

lg_sys.pl: a huge program for computational linguistics, LexGram version 0.9.2 by E. Koenig, which includes the entire CUF system version 2.31. The CUF system implements the Comprehensive Unification Formalism, a formalism for unification grammars that was developed within the ESPRIT project DYANA and extended within projects DYANA-2 (ESPRIT) and B5 (SFB 340) at the Institut für Maschinelle Sprachverarbeitung (IMS), Universität Stuttgart, who holds the copyright.

linTAP.pl: a theorem prover for Multiplicative Linear Logic M?LL, i.e., multiplicative linear logic with positive '?' and negative '!', written by J. Otten.

ljt.pl: an intuitionistic theorem prover using LWB syntax, by R. Dyckhoff. For more information see "Contraction-free calculi for intuitionistic logic" by the same author, appeared in the Journal of Symbolic Logic, 1992.

llprover.pl: a linear logic theorem prover for SICStus Prolog by N. Tamura with TEX form output by E. Sugiyama.

log_interp.pl: version 3.0.8 of the {log} interpreter by A. Dovier, E. Omodeo, C. Piazza, E. Pontelli, and G. Rossi.

lojban.pl: the Lojban semantic analyzer by N. Nicholas.

metutor.pl: the Metutor means-ends tutoring system by N. C. Rowe, with the "firefighting tutor" example application.

mixtus-all.pl: an automatic partial evaluator for full Prolog, by D. Sahlin.

nand.pl: a program by B. Holmer implementing a rough approximation to the algorithm presented in E. S. Davidson, "An Algorithm for NAND Decomposition Under Network Constraints", IEEE Trans. Comp., vol. C-18, no. 12, Dec. 1969, p. 1098.

nbody.pl: a program solving the n-body problem for star clusters by E. Tick.

`oldchina.pl`: a prehistoric version of CHINA, when it was written in Prolog.

`parser_dcg.pl`: a parser by R. Bagnara, based on *Definite Clause Grammars*, for both an imperative and a functional higher-order languages.

`peephole1.pl`: the peephole optimizer of SB-Prolog 3.0.

`pets_an.pl`: a groundness data-flow analyzer for Prolog, using the *Pos* domain implemented by means of ROBDDs, by P. Schachte.

`peval.pl`: a self-applicable partial evaluator for the flow-chart language described in Chapter 4 of *Partial Evaluation and Automatic Program Generation*, by N. D. Jones, C. K. Gomard, and P. Sestoft, Prentice-Hall, 1993.

`plaiclp.pl`: a version of PLAI, the UPM-CLIP framework and environment for developing global analyzers based on abstract interpretation.

`pmatch.pl`: a text-dictionary matcher by A. Michiels.

`press.pl`: a program for solving equations taken from Chapter 23 of *The Art of Prolog*, by L. Sterling and E. Y. Shapiro, The MIT Press, 1986.

`puzzle.pl`: a version of the houses (zebra) program as presented in *Constraint Satisfaction in Logic Programming*, by P. Van Hentenryck, The MIT Press, 1989.

`quot_an.pl`: a data-flow analyzer for groundness employing a quotient of Sharing, by A. King.

`read.pl`: a Prolog reader by D. H. D. Warren and R. O'Keefe, with modifications by A. Mycroft.

`reducer.pl`: a graph reducer for T-combinators by P. Van Roy.

`reg.pl`: the "Regular Approximation Tool" by J. Gallagher.

`rubik.pl`: a Rubik's cube solver written by D. Merritt as described in *Building Expert Systems in Prolog*, Springer-Verlag, 1989.

`sax-all.pl`: the SAX natural language parsing system by Y. Matsumoto and Y. Den.

`scc.pl`: a program written by S. Diehl for computing the strongly-connected components of a graph.

`sdda.pl`: a data-flow analyzer that represents aliasing.

`semi.pl`: a program for playing with semigroups, by M. Carlsson.

`sim.pl`: a simulator for OR-parallel Prolog, by K. Shen.

`sim_v5-2.pl`: an algebra simplifier by L. Zhuhai.

`simple_an.pl`: a simple data-flow analyzer written by P. Van Roy.

`slice-all.pl`: an interpreter, based on the SOS approach, for both an imperative and a functional higher-order languages, written by R. Bagnara.

`spsys.pl`: the SP system, comprises a collection of programs for performing specialization (partial evaluation), some other program transformations, and some program analyses. Copyright by J. Gallagher.

`trs.pl`: an automated prover for first-order classical predicate logic based on a modified version of Gentzen's sequent calculus LK, by K. Sakai.

`unify.pl`: a compiler code generator for unification written by P. Van Roy.

`warplan.pl`: the famous "Warplan" robot problem solver written by D. H. D. Warren.