An Algebraic Approach to Sharing Analysis of Logic Programs

Michael Codish[1] Vitaly Lagoon[1] Francisco Bueno[2]

[1] Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, PoB 653, Beer-Sheba. 84105, Israel.
[2] Department of Artificial Intelligence, Universidad Politécnica de Madrid (UPM), Campus de Montegancedo. 28660, Boadilla del Monte, Madrid, Spain.

Abstract. This paper describes an algebraic approach to the sharing analysis of logic programs based on an abstract domain of *set logic programs*. Set logic programs are logic programs in which the terms are sets of variables and unification is based on an associative, commutative, and idempotent equality theory. We show that the proposed domain is isomorphic to the set-sharing domain of Jacobs and Langen and argue that there are good reasons to adopt our representation: (1) the abstract domain and the abstract operations defined are based on a theory for sets and set unification, resulting in a more intuitive approach to sharing analysis; (2) the abstract substitutions are like substitutions and can be applied to (abstract) atoms. This facilitates program analyses performed as abstract compilation. Finally (3) our representation makes explicit the "domain" of interest of an abstract substitution – which solves some technical problems in defining the domain of Jacobs and Langen.

1 Introduction

Two variables in a logic program are said to be *aliased* if in some execution of the program they are bound to terms which contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a ground term in every execution of the program. Aliasing and groundness information, often called *sharing* information in the logic programming community, provide the basis for a wide range of program optimizations and other useful applications. Such information can be used to identify circumstances in which the occur check may be safely dispensed with [21, 23] or to determine run-time goal independence which can be used to eliminate costly run-time checks in and-parallel execution of logic programs [20, 16, 14]. In the context of concurrent logic programming languages, sharing information can be used to identify *single-writer* properties (e.g., structures which are constructed by a single process). One of the more widely applied sharing analyses reported in the literature is the so called *set-sharing* analysis due to Jacobs and Langen [16], first implemented by Muthukumar and Hermenegildo [20]. This analysis plays a central role in the And-Parallel Prolog compiler described in [14]. The analysis of Jacobs and Langen as well as many of its extensions are developed within the framework of *abstract interpretation* [8] which provides the basis for a semantic approach to dataflow analysis.

This paper presents an algebraic approach for the sharing analysis of logic programs using *set logic programs*. In set logic programs terms are sets of variables, and standard unification is replaced by a suitable unification for sets based on the well studied notion of ACI1-unification [1, 17]. Namely, unification in the presence of an associative, commutative, and idempotent equality theory with a unit element (the empty set). Analyses are semantic based and formalized in terms of abstract atoms in which the arguments are sets of variables that specify the possible sharing and definite groundness between the argument positions of concrete atoms. Abstract atoms can also be viewed as abstract substitutions applied to atoms. Abstract substitutions are *set substitutions*, which are mappings from variables (argument positions) to sets of variables (arguments). We show that the domain of set substitutions is isomorphic to the set-sharing domain. We argue that set logic programs provide a more natural and intuitive means for describing sharing analyses.

The main advantage of our approach is that the operations on the abstract domain and their formal justification follow from simple and obvious algebraic properties of set substitutions. The *composition* of set substitutions, *application* of a set substitution to an (abstract) atom, and the *projection* of a set substitution to a set of relevant variables, all maintain their standard definitions (just as composition, application and projection of usual substitutions). We introduce a natural ordering on abstract syntactic objects which reflects a notion of "less sharing" (similar to "less general" on concrete syntactic objects). The abstract unification of a pair of abstract terms corresponds to finding their most general unifier with respect to this ordering. Unification of abstract atoms is defined as solving sets of unification equations between abstract terms.

Another advantage of the domain we propose is that it enables us to perform program analysis by combining program abstraction (replacing terms by sets of variables) with concrete evaluation (enhanced by ACI1-unification). This approach is often termed *abstract compilation* and derived from ideas presented in [15, 7, 13] and has been applied in a variety of applications [11, 12, 2, 9]. To our knowledge, no previous work has provided an abstract compilation scheme for sharing information. This is no surprise, since the complexity involved in an accurate analysis for such information makes it a non-trivial task. We attack this problem by formulating abstract compilation as a transformation of logic programs into (abstract) set logic programs. The underlying algebraic framework could be cast in terms of a *generalized constraint system*, following [13]. This is for example the approach in [22], where (groundness and type) analyses are designed as constraint solving. We prefer to follow a more compact presentation. A similar approach based on ACI-unification has recently been applied to derive polymorphic type dependencies [4].

2 Preliminaries

In the following we assume a familiarity with the standard definitions and notation for logic programs as described in [18]. We assume a first order language with a fixed set of predicate symbols Π, a fixed signature Σ and a countable set of variables $\mathcal{V}$. The set of atoms constructed using predicate symbols from Π and terms from $T(\Sigma, \mathcal{V})$ is denoted by $\mathcal{B}_{\mathcal{V}}$. The set of variables occurring in a syntactic object o is denoted $vars(o)$.

We assume the standard ordering on terms and other syntactic objects. We let $t_1 \leq t_2$ denote that t_1 is less general than t_2. A *substitution* θ is a mapping from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$ which acts as the identity almost everywhere, i.e., its domain $dom(\theta) = \{x \in \mathcal{V} \mid x\theta \neq x\}$ is finite. The range of a substitution is defined as $range(\theta) = \{x\theta \mid x \in dom(\theta)\}$. Composition of substitutions θ and ψ is defined as usual and denoted $\theta \circ \psi$. A substitution ψ is *idempotent* if $\psi \circ \psi = \psi$ (or $dom(\psi) \cap range(\psi) = \emptyset$). The set of idempotent substitutions is denoted Sub. For substitutions we write $\theta_1 \leq \theta_2$ to denote that θ_1 is less general than θ_2 if there exists a substitution θ such that $\theta_2 \circ \theta = \theta_1$. The empty (identity) substitution is denoted ε. A substitution extends to apply to any syntactic object in the usual way.

We say that a set of variables S *share* through a substitution θ if there exists a variable v such that $S = \left\{ x \in dom(\theta) \,\middle|\, v \in vars(x\theta) \right\}$. In this case we say that the variables in S share in θ through v. Jacobs and Langen denote this as $occs(\theta, v) = S$ [16]. Similarly, for an atom $p(t_1, \ldots, t_n)$ we say that the set of argument positions $I \subseteq \{1, \ldots, n\}$ *share* if there exists a variable v such that $I = \left\{ i \,\middle|\, v \in vars(t_i) \right\}$. In this case we might say that the argument positions I in $p(t_1, \ldots, t_n)$ share through v.

3 Set Logic Programs

The sharing analyses described in this paper are constructed using a first order language, similar to that of logic programs, which we call *set logic programs*. Intuitively, set logic programs are logic programs in which the terms are sets of variables, which we call abstract terms.

Abstract terms: Syntactically, we assume a set of variables $\mathcal{V}$ and an underlying alphabet $\Sigma^{\oplus} = \{\oplus, \emptyset\}$ consisting of a binary function symbol $\oplus$ which "glues" elements together and a single constant symbol $\emptyset$ to represent the empty set. Abstract terms, or *set expressions*, are elements of the term algebra $T(\Sigma^{\oplus}, \mathcal{V})$ modulo an equality theory consisting of the following axioms:

$$
\begin{array}{llll}
(x \oplus y) \oplus z = x \oplus (y \oplus z) & (\textit{associativity}) & x \oplus x = x & (\textit{idempotence}) \\
x \oplus y = y \oplus x & (\textit{commutativity}) & x \oplus \emptyset = x & (\textit{unit element})
\end{array}
$$

This equality theory is sometimes referred to as ACI1 and the corresponding equivalence relation on terms denoted $=_{ACI1}$. This notion of equivalence suggests that abstract terms can be viewed as sets of variables. For example, the terms $x_1 \oplus x_2 \oplus x_3$, $x_1 \oplus x_2 \oplus x_3 \oplus \emptyset$, and $x_1 \oplus x_2 \oplus x_3 \oplus x_2$ can each be viewed as representing the set $\{x_1, x_2, x_3\}$ of three variables. In the following we do not distinguish between set expressions and sets of variables, often referring to the set of variables in a term as a set expression.

Abstract atoms: Abstract atoms are entities of the form $p(\tau_1, \ldots, \tau_n)$ where $p/n \in \Pi$ and $\tau_1, \ldots, \tau_n$ are abstract terms. Abstract atoms are also viewed modulo ACI1-equivalence. Namely, two abstract atoms are equivalent, denoted $\pi_1 =_{ACI1} \pi_2$, if their corresponding arguments are equivalent. We denote the set of abstract atoms modulo ACI1-equivalence by $\mathcal{B}_{\mathcal{V}}^{\oplus}$. In order to simplify notation we often write an abstract atom π instead of its corresponding equivalence class $[\pi]_{=_{ACI1}}$. We also denote ACI1-equality ("$=_{ACI1}$") of abstract atoms by equality sign ("$=$") when it is unambiguous.

While our abstract domain is defined in terms of abstract atoms, abstract substitutions also play a role in the presentation. Moreover, abstract substitutions are used later to show the isomorphism between our domain and the set-sharing domain of Jacobs and Langen.

Abstract substitutions: Abstract substitutions, or *set substitutions*, are substitutions which map variables of $\mathcal{V}$ to abstract terms. We denote the set of idempotent abstract substitutions by $Sub^{\oplus}$. The application of an abstract substitution μ to an abstract term τ is defined as usual by replacing occurrences of each variable x in τ by the abstract term $\mu(x)$. We say that two abstract substitutions μ_1 and μ_2 are ACI1-equivalent if they map variables to equivalent abstract terms,

$$\mu_1 =_{ACI1} \mu_2 \quad \Leftrightarrow \quad \forall x \in \mathcal{V} : \; x\mu_1 =_{ACI1} x\mu_2.$$

The standard operations on abstract substitutions such as projection and composition are defined just as for usual substitutions.

We say that an abstract substitution ψ is *linear* if

$$\forall x, y \in dom(\psi) : \quad x \neq y \;\Rightarrow\; x\psi \cap y\psi = \emptyset. \tag{1}$$

Our interest in linear substitutions is due to the fact that they do not introduce additional sharing dependencies when applied to a syntactic object. Namely, if x and y are distinct variables and ψ is a linear abstract substitution then $x\psi$ and $y\psi$ do not have any variables in common. Linear substitutions induce an ordering on syntactic objects which reflects the "amount" of sharing they contain.

An ordering: We say that an abstract atom π_1 is less general than π_2, denoted $\pi_1 \preceq \pi_2$, if and only if there is a linear substitution ψ such that $\pi_1 =_{ACI1} \pi_2\psi$. It is straightforward to show that $\pi_1 \preceq \pi_2$ implies that the arguments of π_2 contain more sharing and less groundness than the arguments of π_1. Namely, if I is a set of argument positions which share in π_1 (through some variable), then I also share in π_2 (through some variable). Moreover, if π_2 binds a variable x to the empty set, then so must π_1.

Example 1. Consider the following abstract atoms:

$$\pi_1 = p(\{A, B\}, \{B, C\}, \{A, B, D\}),\; \pi_2 = p(\{X\}, \{X, Y\}, \{X, Z\}),$$
$$\pi_3 = p(\{U\}, \{V\}, \{U, W\}).$$

The first and the third arguments of π_1 share through A, while all three arguments share through B. In π_2 there is sharing between all three arguments through X, and in π_3 there is sharing between the first and the third argument. Thus, π_1 contains more sharing than each of π_2 and π_3. In our domain this is captured as $\pi_2 \preceq \pi_1$ and $\pi_3 \preceq \pi_1$. This because $\pi_2 = \pi_1\psi_1$ and $\pi_3 = \pi_1\psi_2$ where:

$$\begin{aligned}\psi_1 &= \{A \mapsto \emptyset,\ B \mapsto X,\ C \mapsto Y,\ D \mapsto Z\}\\ \psi_2 &= \{A \mapsto U,\ B \mapsto \emptyset,\ C \mapsto V,\ D \mapsto W\}.\end{aligned}$$

Note also that the ordering "$\preceq$" naturally induces the ACI1-equivalence of atoms, i.e., $(\pi_1 \preceq \pi_2) \wedge (\pi_2 \preceq \pi_1) \Leftrightarrow (\pi_1 =_{ACI1} \pi_2)$.

A similar ordering is defined for abstract substitutions. An abstract substitution μ_1 is less general than μ_2, denoted $\mu_1 \preceq \mu_2$, if and only if there exists a linear substitution ψ such that $\mu_2 \circ \psi =_{aci} \mu_1$. Note that if $\mu_1 \preceq \mu_2$ and $\mu_2 \preceq \mu_1$ then μ_1 and μ_2 map variables to ACI1-equivalent terms and are hence ACI1-equivalent.

Least upper bound: The least upper bound of two abstract atoms with the same predicate symbol and arity is induced by the ordering of abstract objects.

Proposition 1. *For two abstract atoms $p(\tau_1, \ldots, \tau_n)$ and $p(\tau'_1, \ldots, \tau'_n)$ of the same predicate symbol p/n:*

$$p(\tau_1, \ldots, \tau_n) \sqcup p(\tau'_1, \ldots, \tau'_n) = p(\tau_1 \oplus \tau'_1, \ldots, \tau_n \oplus \tau'_n).$$

Proof. Technical.

The notion of least upper bound extends to sets of abstract atoms by combining all of the atoms with the same predicate symbol. Let $\mathcal{I} \subseteq \mathcal{B}^{\oplus}_{\mathcal{V}}$, then

$$\sqcup\, \mathcal{I} = \left\{\ \sqcup\{p(\tau_1, \ldots, \tau_n) \in \mathcal{I}\} \,\middle|\, p/n \in \Pi\ \right\}. \tag{2}$$

4 An Abstract Domain for Sharing Analysis

We propose set logic programs as a formal basis for studying sharing properties of logic programs. The sets of variables in the arguments of an abstract atom represent possible sharing information in corresponding concrete arguments. The formal relation between concrete atoms and abstract atoms is given in terms of an abstraction function on atoms which replaces the concrete terms in an atom by the set of variables it contains.

$$\begin{aligned}&\sigma : T(\Sigma, \mathcal{V}) \to T(\Sigma^{\oplus}, \mathcal{V})\\ &\sigma(t) = \begin{cases}\emptyset & \text{if } vars(t) = \emptyset\\ x_1 \oplus \cdots \oplus x_n & \text{if } vars(t) = \{x_1, \ldots, x_n\},\ n > 0\end{cases}\end{aligned} \tag{3}$$

The abstraction of atoms is obtained by application of term abstraction separately to each argument:

$$\begin{aligned}&\sigma : \mathcal{B}_{\mathcal{V}} \to \mathcal{B}^{\oplus}_{\mathcal{V}}\\ &\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))\end{aligned} \tag{4}$$

Example 2. Consider the concrete atom $a = p([X,Y|Xs], f(Ys), g(X,Y,Z))$. Its abstraction is:

$$\begin{aligned}\sigma(a) &= \sigma(p([X,Y|Xs], f(Ys), g(X,Y,Z)) = p(X \oplus Y \oplus Xs,\ Ys,\ X \oplus Y \oplus Z)\\ &= p(X' \oplus Xs,\ Ys,\ X' \oplus Z)\end{aligned}$$

where the equivalence of the last two atoms is provided by the pair of linear substitutions: $\psi_{\preceq} = \{Y \mapsto \emptyset,\ X \mapsto X'\}$ and $\psi_{\succeq} = \{X' \mapsto (X \oplus Y)\}$.

We say that an abstract atom π describes a concrete atom a if $\sigma(a) \preceq \pi$, denoted $\pi \propto a$. Namely, if the arguments of π contain more sharing than those of a, and if any argument position of π which is ground is also ground in a.

Abstraction of substitutions: Abstract atoms and abstract substitutions are in fact two equivalent representations of the same dependency information. Moreover, abstract atoms are no more than "syntactic sugar" for $\Pi \times Sub^{\oplus}$. However, abstract substitutions are sometimes more convenient objects to reason about. Thus, similarly to the case of atoms, we define the notions of abstraction and approximation for substitutions. A substitution is abstracted by abstracting the terms in its range:

$$\begin{aligned}&\sigma : Sub \to Sub^{\oplus}\\ &\sigma(\theta) = \left\{\ x \mapsto \sigma(t) \,\middle|\, [x \mapsto t] \in \theta\ \right\}.\end{aligned} \tag{5}$$

We say that an abstract substitution μ describes a concrete substitution θ, denoted $\mu \propto \theta$, if $\sigma(\theta) \preceq \mu$.

Abstract interpretations: An abstract domain for sharing analysis and the corresponding abstraction and concretization functions are obtained by considering *abstract interpretations*, i.e., sets of abstract atoms. We view sets of abstract atoms as being downwards-closed. A set $S \in \wp(\mathcal{B}^{\oplus}_{\mathcal{V}})$ is downwards-closed if $\pi \in S$ and $\pi' \preceq \pi$ implies $\pi' \in S$. We equip $\mathcal{B}^{\oplus}_{\mathcal{V}}$ with an ordering defined as

$$\mathcal{I}_1 \preceq \mathcal{I}_2 \Leftrightarrow \forall \pi_1 \in \sqcup\mathcal{I}_1\ \exists \pi_2 \in \sqcup\mathcal{I}_2 :\ \pi_1 \preceq \pi_2. \tag{6}$$

The corresponding equivalence relation:

$$\mathcal{I}_1 \approx \mathcal{I}_2 \ \Leftrightarrow\ (\mathcal{I}_1 \preceq \mathcal{I}_2) \wedge (\mathcal{I}_2 \preceq \mathcal{I}_1) \tag{7}$$

provides a basis for relating abstract interpretations to downwards-closed sets of (abstract) atoms (also called *c-interpretations* in [10]). Let $S\Downarrow$ denote a minimal downwards-closed set containing S. Equation (7) provides $S \approx S\Downarrow$ for any $S \in \wp(\mathcal{B}^{\oplus}_{\mathcal{V}}\ )$, i.e., any member of $\wp(\mathcal{B}^{\oplus}_{\mathcal{V}})$ is equivalent to some downwards-closed set. The quotient $[\wp(\mathcal{B}^{\oplus}_{\mathcal{V}}\ )]_{\approx}$ of the equivalence relation will be denoted in the following by an abuse of notation as $\wp(\mathcal{B}^{\oplus}_{\mathcal{V}})$.

Lemma 2. *$(\wp(\mathcal{B}^{\oplus}_{\mathcal{V}}), \preceq)$ is a complete lattice.*

Proof. If L is a set of downwards-closed sets then $\cap L$ and $\sqcup L$ are downwards-closed, therefore $lub(L) \approx \sqcup L$ and $glb(L) \approx \cap L$.

The relation between concrete and abstract interpretations is formalized as usual in terms of a pair of abstraction and concretization functions lifted from the abstraction function σ on atoms in the standard way:

$$\begin{array}{ll} \alpha : \wp(\mathcal{B}_{\mathcal{V}}) \to \wp(\mathcal{B}_{\mathcal{V}}^{\oplus}) & \gamma : \wp(\mathcal{B}_{\mathcal{V}}^{\oplus}) \to \wp(\mathcal{B}_{\mathcal{V}}) \\ \alpha(I) = \sqcup \left\{ \sigma(a) \,\middle|\, a \in I \right\} & \gamma(\mathcal{I}) = \bigcup \left\{ I \,\middle|\, \alpha(I) \preceq \mathcal{I} \right\} \end{array} \tag{8}$$

Theorem 3. *$\langle \wp(\mathcal{B}_{\mathcal{V}}), \alpha, \wp(\mathcal{B}_{\mathcal{V}}^{\oplus}), \gamma \rangle$ is a Galois insertion.*

Proof. Straightforward from the above definitions.

In order to ensure terminating analysis and finite approximations we establish finiteness of our abstract domain modulo the equivalence relation of Equation (7). But first note that for any abstract interpretation $\mathcal{I}$, $\mathcal{I} \approx \sqcup\mathcal{I}$. This follows by Equation (6). Intuitively, this means that we view abstract interpretations as "lubbed". It is worth noting that $\sqcup\mathcal{I}$ is an abstract interpretation with minimal cardinality among those equivalent to $\mathcal{I}$, containing at most one abstract atom for each predicate symbol in Π.

Theorem 4. *$\mathcal{B}_{\mathcal{V}}^{\oplus}$ is finite.*

Proof. The elements of $\mathcal{B}_{\mathcal{V}}^{\oplus}$ are equivalence classes of abstract atoms $[\pi]_{=_{aci}}$. Therefore it suffices to prove that for any predicate symbol p/n the number of associated equivalence classes of abstract atoms $[\pi]_{=_{aci}}$ is finite. We prove the claim of the theorem demonstrating that each equivalence class of abstract atoms constructed using p/n has a representative with a number of variables bounded by $2^n - 1$. Assume an atom π has more than $2^n - 1$ distinct variables. Then there are at least two variables x and y occurring in exactly the same set of argument positions of π. Consider the atom $\pi' = \pi \cdot \{x \mapsto \emptyset,\ y \mapsto z\}$ where z is a fresh variable. By construction we have $\pi' \preceq \pi$ and it is easy to see that $\pi \preceq \pi'$ with $\pi = \pi' \cdot \{z \mapsto (x \oplus y)\}$. Thus, π and π' are in the same equivalence class and $|vars(\pi')| = |vars(\pi)| - 1$. It follows that for any atom having two or more variables in the same set of argument positions we can find an equivalent atom with a smaller set of variables. So, for any atom constructed using p/n there exists an equivalent atom with all variables occurring in distinct subsets of argument positions, i.e., an atom with at most $2^n - 1$ variables.

The abstract domain $\langle \wp(\mathcal{B}_{\mathcal{V}}), \alpha, \wp(\mathcal{B}_{\mathcal{V}}^{\oplus}), \gamma \rangle$ provides a basis for the sharing analysis of logic programs. When constructing a program analysis for logic programs three main operations must be defined: abstract unification, abstract application (or projection[3]), and abstract least upper bound. We show that all of these operations are straightforward to define in our domain.

[3] The role of projection is to restrict a substitution to a set of variables of interest. Using application instead facilitates the design of the analysis.

Abstract Unification: Formally speaking, the unification of abstract atoms is similar to the well studied notion of ACI1-unification [1, 17]. Recall that an ACI1-unifier of two terms τ_1, τ_2 is a substitution ψ such that $\tau_1\psi =_{ACI1} \tau_2\psi$. However, there are two important differences: (1) because the underlying alphabet contains only one binary function symbol and only one constant, abstract terms always unify[4] and have exactly one most general unifier (in the general case there may be a finite number of most general unifiers and the unification problem is NP-complete). And (2) because we define the ordering on terms using the notion of linear substitutions, the most general unifier is non-standard (though well defined).

The abstract unification (for sharing analysis) of a pair of terms τ_1, τ_2, denoted $mgu_{ACI1}(\tau_1, \tau_2)$, is the most general substitution μ such that $\tau_1\mu =_{ACI1} \tau_2\mu$. Figure 1 describes a simple algorithm to compute the most general unifier of a pair of abstract terms. Note in the figure that if τ_1 or τ_2 is ground then $S = \emptyset$ and $Z = \emptyset$ which implies that μ binds all variables to $\emptyset$.

Lemma 5. *Let t_1 and t_2 be concrete terms with $\theta = mgu(t_1, t_2)$. Let $\xi = mgu_{ACI1}(\sigma(t_1), \sigma(t_2))$, computed by the algorithm in Figure 1. Then $\xi \propto \theta$.*

Proof. Technical, by observing that the algorithm in Figure 1 enables all possible sharing which might occur in the unification of t_1 and t_2. If θ implies less sharing than that, a linear abstract substitution ψ can be constructed such that it maps out the extra sharing, and therefore $\xi \circ \psi = \sigma(\theta)$.

The abstract unification $mgu^{\mathcal{A}}(\pi_1, \pi_2)$ of atoms π_1 and π_2 is defined in terms of the set $\mathcal{E}$ of equations between the terms in the corresponding argument positions by:

$$mgu^{\mathcal{A}}(\mathcal{E}) = \begin{cases} \emptyset & \text{if } \mathcal{E} = \emptyset \\ \mu \circ mgu^{\mathcal{A}}(\mathcal{E}'\mu) & \begin{array}{l}\text{if } \mathcal{E} = \{(\tau = \tau')\} \cup \mathcal{E}' \\ \text{and } \mu = mgu_{ACI1}(\tau, \tau')\end{array} \end{cases} \tag{9}$$

Abstract unification is thus defined much the same as in the concrete case. It is parameterized on abstract unification of terms and abstract composition of substitutions. The following result about abstract composition is instrumental in proving correctness of abstract unification.

Lemma 6. *Let θ_1 and θ_2 be concrete substitutions, and μ_1 and μ_2 abstract substitutions such that $\mu_1 \propto \theta_1$ and $\mu_2 \propto \theta_2$. Then $(\mu_1 \circ \mu_2) \propto (\theta_1 \circ \theta_2)$.*

Proof. Since $\mu_1 \propto \theta_1$ and $\mu_2 \propto \theta_2$, we have that $\exists\psi_1 : \ \mu_1 \circ \psi_1 = \sigma(\theta_1)$ and $\exists\psi_2 : \ \mu_2 \circ \psi_2 = \sigma(\theta_2)$. We have to prove that $\exists\psi : \ \mu_1 \circ \mu_2 \circ \psi = \sigma(\theta_1 \circ \theta_2)$. It is easy to see that this holds with $\psi = \psi_1 \circ \psi_2$.

---

[4] Observe that the substitution which binds all variables in two terms to $\emptyset$ is always a unifier.

Input: abstract terms τ_1 and τ_2
Output: most general ACI1-unifier μ

$$
\begin{array}{l}
v_1 = vars(\tau_1) \\
v_2 = vars(\tau_2) \\
S = \left\{ \, s \subseteq (v_1 \cup v_2) \,\middle|\, s \cap v_1 \neq \emptyset,\ s \cap v_2 \neq \emptyset \, \right\} \\
\textbf{let } S = \{s_1, \ldots s_k\} \\
\quad Z = \{z_1, \ldots, z_k\} \qquad \text{// fresh variables corresponding to members of } S \\
\mu = \left\{ \, x \mapsto z_1 \oplus \cdots \oplus z_n \,\middle|\, \begin{array}{l} x \in (v_1 \cup v_2) \\ \{z_1, \ldots, z_n\} = \{z_i | x \in s_i\} \end{array} \right\}
\end{array}
$$

Fig. 1. ACI1-unification of two terms

Theorem 7 abstract unification is correct.
Let a_1 and a_2 be concrete atoms such that $mgu(a_1, a_2) = \theta$. Let π_1 and π_2 be abstract atoms such that $\pi_1 \propto a_1$ and $\pi_2 \propto a_2$. Let $\mu = mgu^{\mathcal{A}}(\pi_1, \pi_2)$. Then $\mu \propto \theta$.

Proof. By induction on the length of corresponding tuples of abstract terms with base case provided by Lemma 5 and the induction step provided by Lemma 6.

The result of Theorem 7 is one of the main points in our presentation. It shows that there is a natural ordering (based on linear substitutions) for set-sharing analysis for which abstract unification is defined as inductive solving equations similarly to the case of concrete unification.

Abstract application: The application of an abstract substitution to an abstract atom is defined just as for the case of a standard substitution. The correctness for sharing analysis follows as a corollary from Theorem 7.

Corollary 8 application of abstract mgu is correct.
Let a_1 and a_2 be concrete atoms, and π_1 and π_2 be abstract atoms. Let $\pi_1 \propto a_1$ and $\pi_2 \propto a_2$, with $mgu(a_1, a_2) = \theta$ and $mgu^{\mathcal{A}}(\pi_1, \pi_2) = \mu$. Let h be a concrete atom. Then $\sigma(h)\mu \propto h\theta$.

Proof. From Theorem 7 it holds $\sigma(a_1\theta) \preceq \pi_1\mu$, which implies that $\sigma(\langle h, a_1\rangle\theta) \preceq \langle\sigma(h), \pi_1\rangle\mu$. Thus, $\sigma(h\theta) \preceq \sigma(h)\mu$.

Abstract lub: It is easy to see that the least upper bound operation on abstract atoms of Section 3 is safe, and precise, for sharing analysis.

Lemma 9 abstract lub is correct.
For abstract atoms π_1 and π_2 and concrete atoms a_1 and a_2,

$$(\pi_1 \propto a_1) \wedge (\pi_2 \propto a_2) \;\Rightarrow\; (\pi_1 \sqcup \pi_2 \propto a_1) \wedge (\pi_1 \sqcup \pi_2 \propto a_2).$$

Proof. Straightforward. Since $\pi_1 \sqcup \pi_2$ is an upper bound of π_1 and π_2 we have $\pi_1 \preceq (\pi_1 \sqcup \pi_2)$ and $\pi_2 \preceq (\pi_1 \sqcup \pi_2)$. Because $\pi_1 \propto a_1$ and $\pi_2 \propto a_2$ we have $\sigma(a_1) \preceq \pi_1$ and $\sigma(a_2) \preceq \pi_2$. Thus, $\sigma(a_1) \preceq (\pi_1 \sqcup \pi_2)$ and $\sigma(a_2) \preceq (\pi_1 \sqcup \pi_2)$, i.e., $\pi_1 \sqcup \pi_2 \propto a_1$ and $\pi_1 \sqcup \pi_2 \propto a_2$.

When considering the lub of abstract interpretations, the operation is trivially precise as the ordering of Equation (7) already views interpretations as "lubbed". It follows that γ is indeed well defined, as $\mathcal{I} \approx \sqcup\mathcal{I}$ and $\gamma(\mathcal{I}) = \gamma(\sqcup\mathcal{I})$.

5 Set Logic Programs vs Set-Sharing

We show that the abstract domain based on set logic programs is isomorphic to the well-known set-sharing representation of Jacobs and Langen [16]. In the following we denote by *Sharing* the abstract domain of [16].

We say that two abstract domains of a given concrete domain are isomorphic if (1) they are isomorphic as partial orders; and (2) the interpretation of the corresponding abstract objects is the same. Namely, the closure operators $\gamma \circ \alpha$ defined by the corresponding abstraction and concretization functions are the same.

The domain of set-sharing: Recall the original definition of the *Sharing* domain which consists of sets of sets of program variables ordered by set inclusion. Sharing information is characterized using a notion of the occurrences of a variable through a substitution,

$$\begin{array}{l} occs : (Sub \times \mathcal{V}) \to \wp(\mathcal{V}) \\ occs(\theta, v) = \left\{ \, x \in dom(\theta) \,\middle|\, v \in vars(x\theta) \, \right\}. \end{array}$$

If $occs(\theta, v) = S$ then S is the set of variables which θ maps to a term containing v. Each set S in an abstract substitution κ represents the possibility of a variable v occurring through the variables of S in the range of a substitution described by κ. The abstraction function for the sharing domain is defined in terms of the *sharing groups* of a substitution, formalized by:

$$\begin{array}{l} \mathcal{A} : Sub \to Sharing \\ \mathcal{A}(\theta) = \left\{ \, occs(\theta, v) \,\middle|\, v \in vars(range(\theta)) \, \right\}. \end{array}$$

A Galois insertion is then constructed:

$$\begin{array}{ll} \alpha^{Sh} : \wp(Sub) \to Sharing & \gamma^{Sh} : Sharing \to \wp(Sub) \\ \alpha^{Sh}(\Theta) = \bigcup \left\{ \, \mathcal{A}(\theta) \,\middle|\, \theta \in \Theta \, \right\} & \gamma^{Sh}(\kappa) = \left\{ \, \theta \in Sub \,\middle|\, \mathcal{A}(\theta) \subseteq \kappa \, \right\} \end{array} \tag{10}$$

The following example illustrates the description of concrete substitutions by set-sharing substitutions.

Example 3. Let $\kappa = \big\{ \{A, B\}, \{B, C\}, \{A\}, \{B\}, \{C\}, \emptyset \big\}$ be an abstract substitution in the *Sharing* domain. The substitutions $\theta_1 = \{A \mapsto f(X, Y), B \mapsto g(Y, Z), C \mapsto f(Z, V)\}$ and $\theta_2 = \{A \mapsto f(X), B \mapsto g(Y), C \mapsto f(Z)\}$ are described by κ: In θ_1, X occurs through $\{A\}$, Y occurs through $\{A, B\}$, Z occurs through $\{B, C\}$ and V occurs through $\{C\}$, and in θ_2 there are variables which occur through $\{A\}$, $\{B\}$ and $\{C\}$ — and these occurrences are all specified in κ.

An isomorphism: In principle the domain based on set logic programs is formalized in terms of a Galois insertion of *abstract atoms* while the set-sharing domain of Jacobs and Langen is based on a Galois insertion of *abstract substitutions*. This introduces some technicalities into the formal proofs. The reader should notice that in fact set-sharing analyses, such as those used in [16, 20], are actually based on pairs consisting of a concrete atom of the form $p(x_1, \ldots, x_n)$ together with an abstract substitution.[5] Note that an abstract atom $p(\tau_1, \ldots, \tau_n)$ in our domain can also be viewed as a pair $p(x_1, \ldots, x_n)$ together with a set substitution $\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$.

Lemma 10. *There exists a set isomorphism between $Sub^{\oplus}$ and $Sharing$.*

Proof. Note that the abstraction function $\mathcal{A} : Sub \to Sharing$ extends naturally to a function $\mathcal{A} : Sub^{\oplus} \to Sharing$ viewing sets of variables as ordinary terms. Hence, we prove the lemma demonstrating that $\mathcal{A} : Sub^{\oplus} \to Sharing$ is a bijective function for which an inverse function $\mathcal{A}^{-1} : Sharing \to Sub^{\oplus}$ can be provided.

Let $\kappa = \{S_1, \ldots, S_n\}$ be an element of the $Sharing$ domain defined for a set D of variables of interest. Assume without loss of generality that the domain of substitutions in $Sub^{\oplus}$ is D.[6] Let $z_1, \ldots, z_n$ be a set of fresh variables, one for each sharing group in κ. The inverse function yielding the set substitution μ which corresponds to κ is defined by:

$$\mathcal{A}^{-1} : Sharing \to Sub^{\oplus}$$
$$\mathcal{A}^{-1}(\kappa) = \left\{ \, x \mapsto \bigoplus_{x \in S_i} z_i \,\middle|\, x \in D \, \right\}.$$

It is straightforward to see that $\mathcal{A} \circ \mathcal{A}^{-1}$ and $\mathcal{A}^{-1} \circ \mathcal{A}$ correspond to identity functions in $Sub^{\oplus}$ and $Sharing$ respectively.

The following lemma establishes the relation between the ordering of set substitutions and the ordering in the $Sharing$ domain:

Lemma 11. *There is an order embedding between $\langle Sub^{\oplus}, \preceq \rangle$ and $\langle Sharing, \subseteq \rangle$.*

Proof. Let μ_1 and μ_2 be two abstract substitutions and let D be a set of variables of interest. Assume without loss of generality that $dom(\mu_1) = dom(\mu_2) = D$. We prove that $\mu_1 \preceq \mu_2 \Leftrightarrow \mathcal{A}(\mu_1) \subseteq \mathcal{A}(\mu_2)$.

($\Rightarrow$) We have to demonstrate that if μ is a set substitution and ψ is a linear substitution then $\mathcal{A}(\mu\psi) \subseteq \mathcal{A}(\mu)$. Each member of the set $\mathcal{A}(\mu\psi)$ is of the form: $occs(\mu\psi, v) = \left\{ \, x \,\middle|\, v \in vars(x\mu\psi) \, \right\}$. Assume without loss of generality that $range(\mu) = dom(\psi)$. Since ψ is linear, there is exactly one variable $v' \in range(\mu)$

[5] However, set-sharing substitutions can not be applied to atoms, since they are in fact an encoding of sharing information rather than "true" substitutions.

[6] Note that the domain of an abstract substitution $\kappa \in Sharing$ must be explicitly specified, as any variable of interest not occurring in κ is considered ground. In contrast, the variables of interest for a set substitution are those in its domain.

mapped by ψ to a set expression containing v. Thus, all the variables occurring in $\mu\psi$ through v occur in μ through v', i.e., $occs(\mu\psi, v) = occs(\mu, v')$. And so, we have $occs(\mu\psi, v) \in \mathcal{A}(\mu)$.

($\Leftarrow$) Given $\mathcal{A}(\mu_1) \subseteq \mathcal{A}(\mu_2)$ we construct a linear substitution ψ as follows. Consider the set of variables: $S = \{v | occs(\mu_2, v) \in \mathcal{A}(\mu_2) \setminus \mathcal{A}(\mu_1)\}$. For any variable $v \in S$ we add a mapping $\{v \mapsto \emptyset\}$ to ψ. Thus, we have $\mathcal{A}(\mu_2\psi) = \mathcal{A}(\mu_1)$ and consequently, by Lemma 10, $\mu_1\psi = \mu_2$.

Thus, set substitutions of $Sub^{\oplus}$ and abstract substitutions of $Sharing$ form isomorphic partial orders. Considering abstract substitutions together with predicate names we establish that both abstract domains approximate the same sharing information.

Theorem 12 domain isomorphism.

$$\langle \Pi \times Sharing, \subseteq, \alpha^{Sh} \rangle \;\cong\; \langle \wp(\mathcal{B}^{\oplus}_{\mathcal{V}}), \preceq, \alpha \rangle$$

Proof. Technical, by Lemmas 10 and 11 and by observing that $\mathcal{A} : Sub \to Sharing$ and $\sigma : Sub \to Sub^{\oplus}$ map concrete substitutions to isomorphic abstract substitutions. Namely, that $\forall \theta \in Sub : \; \mathcal{A}(\theta) = \mathcal{A}(\sigma(\theta))$, which follows by definition.

The following example illustrates the isomorphism of the two representations of sharing information.

Example 4. Recall the abstract substitution κ and the concrete substitutions θ_1 and θ_2 of Example 3. Consider the set substitution, $\mu = \{A \mapsto \{X, U\}, B \mapsto \{X, Y, V\}, C \mapsto \{Y, W\}\}$. The substitutions θ_1 and θ_2 are described by μ: X indicates the possible sharing of A and B, Y indicates that of B and C, and U, V and W the possible presence in A, B and C of variables not shared with other variables. The abstract substitution κ and the set substitution μ also describe the substitutions $\theta_3 = \{A \mapsto f(X), B \mapsto g(X)\}$ and $\theta_4 = \{A \mapsto f(X, Y), B \mapsto g(X, Y), C \mapsto Z\}$.

6 Sharing Analysis with Set Logic Programs

The abstract operations defined in Section 4 (unification, application, least upper bound) provide the building blocks to construct an abstract semantics for the sharing analysis of logic programs. We have constructed several such program analyses. A bottom-up approach is described in [5]. A top-down approach based on tabulation using XSB is described in [3]. In this section we illustrate as an example a bottom-up approach based on an abstract immediate consequences operator $\mathcal{T}_{\mathcal{P}} : \wp(\mathcal{B}^{\oplus}_{\mathcal{V}}) \to \wp(\mathcal{B}^{\oplus}_{\mathcal{V}})$ for set logic programs. For a logic program P the least fixed point of $\mathcal{T}_{\sigma(P)}$ provides the sharing analysis for P.

$$\mathcal{T}_{\mathcal{P}}(\mathcal{I}) = \sqcup \left\{ h\mu \;\middle|\; \begin{array}{l} h \leftarrow b_1, \ldots, b_n \in \mathcal{P}, \;\; a_1, \ldots, a_n \in \mathcal{I} \\ \mu = mgu^{\mathcal{A}}(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\}. \tag{11}$$

Let us consider the analysis of the well-known *append* program depicted in Figure 2 (left) using the technique discussed above. The analysis is obtained

(1) $append([\,], Ys, Ys).$ (2) $append([X|Xs], Ys, [X|Zs]) \leftarrow$
$append(Xs, Ys, Zs).$

(1′) $append(\emptyset, \{Ys\}, \{Ys\}).$ (2′) $append(\{X, Xs\}, \{Ys\}, \{X, Zs\}) \leftarrow$
$append(\{Xs\}, \{Ys\}, \{Zs\}).$

Fig. 2. The append program and its set based abstraction.

as a least fixed point of $\mathcal{T}_{\sigma(P)}$ applied to the abstract version of *append*, depicted in Figure 2 (right). In the first iteration of the evaluation we collect an abstract atom of the form $\pi_1 = append(\emptyset, \{Ys\}, \{Ys\})$ corresponding to fact (1′) in Figure 2, characterizing the set of atoms in which the first argument is ground and the second and third arguments are equal terms. In the second iteration a renaming of π_1 is unified with the body of clause (2′) in Figure 2. Abstract unification in this case specifies that Xs is bound to $\emptyset$ (a ground term) and that Ys and Zs are bound to the same set (variable Ys'). Consequently the head of clause (2′) under such bindings can be represented as $\pi_2 = append(\{X\}, \{Ys'\}, \{X, Ys'\})$. This abstract atom describes the concrete atoms of the form $append(t_1, t_2, t_3)$ which exhibit sharing between t_1 and t_3 and between t_2 and t_3. Note that $\{\pi_2\} \approx \{\pi_1, \pi_2\}$ since $\pi_1 \preceq \pi_2$ with $\pi_1 = \pi_2 \cdot \{X \mapsto \emptyset\}$. An additional iteration results in a new abstract atom of the form $\pi_3 = append(\{X, X'\}, \{Ys'\}, \{X, X', Ys'\})$, which is equivalent to π_2. Thus, the fixed point is reached with $lfp(\mathcal{T}_{\sigma(P)}) = \{\, append(\{X\}, \{Y\}, \{X, Y\}) \,\}$ providing an approximation of sharing information for *append*.

7 Extensions to Set Logic Programs

Traditionally, sharing analyses have been enhanced with other kinds of information like linearity and freeness. This information is interesting in its own right, since it allows for further compile-time optimizations. Besides, it can significantly improve sharing information.

It is often said that properties such as linearity are hard to deal with in program analysis because they are not downwards-closed. Namely, t may be a linear term but $t\theta$ non-linear. Note that our domain is downwards-closed with respect to the ordering "$\preceq$" (not with respect to the standard instantiation ordering). Consequently, downwards closure of the abstract domain preserves linearity. As a result, linearity can be captured straightforwardly by annotating variables in an adequate way, i.e., by distinguishing between *linear* and *non-linear* abstract terms. Unification is extended accordingly (see [5]). Freeness can also be added in a similar way.

Incorporating term structure to sets substitutions can also be done in a natural way. Abstract substitutions can be obtained by considering terms constructed very much like in the concrete case, but using (annotated) set expressions instead of variables. In order to guarantee finiteness of the domain, and termination of the analysis, the classical depth-k bound on terms can be imposed. Similar extensions to sharing analysis with structural information have been proposed in

[19], based on the formalism of abstract equation systems, and in the $Pat(\mathcal{R})$ domain [6]. Arguably, we expect that the extension sketched here offers advantages of easy design and formal justification.

8 Conclusion

We have described an alternative and isomorphic representation of an abstract domain for sharing analysis. The new domain based on set logic programs leads to intuitive definitions for the abstract operations needed to provide for sharing analyses. This in itself is an advantage over the existing definitions which are hard to motivate and justify. The definitions given in this paper have been extended also for linearity information. It is straightforward to also extend them for term structure information. Top-down and bottom-up analyses based on these domains have been fully implemented. The current prototypes are not as efficient as carefully handcrafted interpreter-based analyzers for goal-dependent analysis, but are comparable for goal-independent analysis. Current efforts are being devoted to more efficient unification algorithms.

References

1. F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford Science Publications, 1994.
2. M. Codish and B. Demoen. Analysing logic programs using "`prop`"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, December 1995.
3. M. Codish, B. Demoen, and K. Sagonas. Xsb as the natural habitat for general purpose program analysis. Technical report, Ben-Gurion University of the Negev, 1997. `ftp://ftp.cs.bgu.ac.il/pub/people/codish/absxsb.ps`.
4. M. Codish and V. Lagoon. Type dependencies for logic programs using aci-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996.
5. M. Codish, V. Lagoon, and F. Bueno. Sharing analysis for logic programs using set logic programs. In *Proceedings of the 1996 Conference on Declarative Programming – APPIA-GULP-PRODE*, pages 29–40, July 1996.
6. A. Cortesi, B. L. Charlier, and P. V. Hentenryck. Combinations of abstract domains for logic programming. In *21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239. ACM Press, 1994.
7. A. Cortesi and G. Filé. Abstract interpretation of logic programs: An abstract domain for groundness, sharing, freeness and compoundness analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. SIGPLAN NOTICES 26 (9)*, pages 52–61. ACM Press, 1991.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc., Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, January 1977.

9. S. Dawson, C. Ramakrishnan, and D. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
11. T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 15–18 1991. IEEE Computer Society Press.
12. J. P. Gallagher and D. A. Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Massachusetts Institute of Technology, 1994. The MIT Press.
13. R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995.
14. M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.
15. M. V. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):349–366, 1992.
16. D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming*, 13(2 and 3):291–314, 1992.
17. P. Lincoln and T. Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8(1,2):217–240, 1989.
18. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2^{nd} edition, 1987.
19. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *International Conference on Logic Programming*. MIT Press, June 1995.
20. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(2 and 3):315–347, 1992.
21. D. A. Plaisted. The occur-check problem in Prolog. In *Proceedings of the International Symposium on Logic Programming*, pages 272–280, Atlantic City, 1984. IEEE, Computer Society Press.
22. C. Ramakrishnan, I. Ramakrishnan, and R. Sekar. A symbolic constraint solving framework for analysis of logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–23, New York, USA, 1995. ACM Press.
23. H. Søndergaard. An application of abstract interpretation of logic programs: Occur-check reduction. In *Proceedings of ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1986. (Extended Abstract).

This article was processed using the LaTeX macro package with LLNCS style