Improving Abstract Interpretations by Combining Domains*

M. Codish, A. Mulkers, M. Bruynooghe[†]
Department of Computer Science
KU Leuven, Belgium
`codish@cs.kuleuven.ac.be`

M. García de la Banda, M. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid, Spain
`maria@fi.upm.es`

Abstract

In this paper we consider static analyses based on abstract interpretation of logic programs over combined domains. It is known that analyses over combined domains potentially provide more information than obtainable by performing the independent abstract interpretations. However, the construction of a combined analysis often requires redefining the basic operations for the combined domain. We demonstrate for logic programs that in practice it is possible to obtain precision in a combined analysis without redefining the basic operations. We also propose a way of performing the combination which can be more precise than the straightforward application of the classical “reduced product” approach, while keeping the original components of the basic operations. The advantage of the approach is that proofs of correctness for the new domains are not required and implementations can be reused. We illustrate our results by showing that a combined sharing analysis — constructed from “old” proposals — compares well with other “new” proposals suggested in recent literature both from the point of view of efficiency and accuracy.

*The work presented in this paper has been supported in part by CEC DGXIII ESPRIT Project “PRINCE” and CICYT project TIC91-0106-CE.

[†]Supported by the Belgian National Fund for Scientific Research

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM-PEPM’93-6/93/Copenhagen, DK

© 1993 ACM 0-89791-594-1/93/0006/0194...\$1.50

1 Introduction

The framework of abstract interpretation [8] provides the basis for a semantic approach to dataflow analysis. A program analysis is viewed as a non-standard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behaviour.

Describing program analysis as a non-standard semantics is more than a theoretical exercise in aesthetics. The semantic approach allows us to focus on the abstraction of data. The framework of abstract interpretation then determines an abstract semantic domain and an abstract semantics. Formal justification of program analyses is reduced to proving conditions on the relation between data and data descriptions and on the elementary operations defined on the data descriptions. This approach eases both the development and the justification of program analyses.

In the case of logic programming languages, “data” corresponds to substitutions and atoms. The basic operations on data typically include unification, composition of substitutions and projection of substitutions onto variables of interest. Proving the safety of an abstract unification function is the major step in proving the safety of abstractions for logic programs. Introductory material for the subject of abstract interpretation of logic programs can be found for example in [11, 10, 13, 17, 22].

It is often the case that program analyses aim to provide a combination of more basic types of information. Typical examples in the context of logic

programs are analyses for: groundness and sharing [4, 16, 20, 23], modes and types [12, 15], sharing and freeness [18, 24], etc. Typically, such combined analyses provide more information than that obtained by combining the information which might be obtained by the corresponding individual analyses. Moreover, efficiency can also improve as the increased precision reduces the number of irrelevant analysis paths which the abstract computation is obliged to follow. However, the design, implementation and formal justification of such combined analyses require new efforts which do not directly benefit from previously designed analyses.

In this paper we observe that in many cases it is possible to provide combined analyses which do benefit from previously defined analyses and which also maintain a high degree of precision. In particular this is the case when the analyses being composed contain a sufficient degree of “overlapping” information. For example, recent proposals to achieve better sharing analyses by combining together the advantages of various old analyses [5, 24] can be derived automatically with little effort.

The theoretical background for the current paper has been laid down already in 1979 by Cousot and Cousot [9]. There, the authors illustrate that although some precision can be gained by removing redundancies from combined domains, still further precision is gained by introducing new basic operations. Here, we focus first on the precision that can be gained simply by removing redundancies. We illustrate that for the case of logic programs, this often provides a practical technique for providing precise combined analyses. We also propose and illustrate an approach which allows deriving more precise information by removing redundancies and combining lower level operations in previously defined analyses.

The rest of the paper is structured as follows. Section 2 provides the theoretical background for our technique. This includes a brief introduction to abstract interpretation following [8] as well as the definitions for combining domains as given in [9]. We close Section 2 with an example for logic programs which demonstrates how Søndergaard’s domain for (pair) sharing and groundness analysis [23] can be constructed by combining corresponding groundness and sharing domains. In Section 3 we introduce an alternative domain for this type of analysis proposed in [16]. We identify the advantages of the two alternative domains and propose to provide the “ultimate” sharing analysis by combining the analyses respectively described in [4] and in [20] for these domains. Another example involves combining the sharing analysis of [4] with the **Sharing+Freeness** analysis presented in [18, 19]. Section 4 provides an experimental eval-

uation of our approach. The combined analyses have been implemented in the context of the $\&$ -Prolog compiler [14, 18, 20] by reusing components of previously defined analyses. The results obtained are at least as good as those obtained in recently developed analysers which completely redesign and reimplement the basic operations. Finally, Section 5 concludes and proposes some directions for further experimentation.

2 Background

In the following we summarize briefly the theory of abstract interpretation as defined in Cousot 77. The theory for combining domains follows the description in Cousot 79.

Abstract interpretation

We assume the standard framework of abstract interpretation as defined in [8] in terms of *Galois insertions*.

Definition 2.1 *Galois insertion*

A *Galois insertion* is a quadruple (E, α, D, γ) where:

1. $(E, \sqsubseteq_E)$ and $(D, \sqsubseteq_D)$ are complete lattices called concrete and abstract domains respectively;
2. $\alpha : E \rightarrow D$ and $\gamma : D \rightarrow E$ are monotonic functions called abstraction and concretization functions respectively; and
3. $\alpha(\gamma(d)) = d$ and $e \sqsubseteq_E \gamma(\alpha(e))$ for every $d \in D$ and $e \in E$.

In general only one of $\{\alpha, \gamma\}$ need be specified since in principle a “best possible” α can always be determined for a given γ and vice versa. For simplicity, in the following only the concretization function γ will generally be specified.

Example 1

Let $\text{Parity} = \{\perp, od, ev, \top\}$ and $\text{Sign} = \{\perp, 0, \hat{+}, \hat{-}, \top\}$ be complete lattices ordered as illustrated in Figure 1. Let

$$\gamma_{\text{parity}} = \left\{ \begin{array}{l} \perp \mapsto \emptyset, \quad ev \mapsto \{x \mid x \bmod 2 = 0\}, \\ od \mapsto \{x \mid x \bmod 2 = 1\}, \quad \top \mapsto \mathcal{Z} \end{array} \right\};$$

$$\gamma_{\text{sign}} = \left\{ \begin{array}{l} \perp \mapsto \emptyset, \quad 0 \mapsto \{0\}, \quad \hat{+} \mapsto \{x \mid x \geq 0\}, \\ \hat{-} \mapsto \{x \mid x < 0\}, \quad \top \mapsto \mathcal{Z} \end{array} \right\}.$$

The following specifies the notion of approximation which is then lifted from the primitive domains to function domains:

Figure 1: *Parity* and *Sign* lattices.

Definition 2.2 approximation

Let (E, α, D, γ) be a Galois insertion and let $\mu : E \rightarrow E$ and $\mu^A : D \rightarrow D$ be monotonic functions. We say that $d \in D$ γ -approximates $e \in E$, denoted $d \propto_\gamma e$, if $e \sqsubseteq_E \gamma(d)$ ¹. We say that μ^A γ -approximates μ , denoted $\mu^A \propto_\gamma \mu$, if $\forall d \in D. e \in E. d \propto_\gamma e \Rightarrow \mu^A(d) \propto_\gamma \mu(e)$.

Example 2

Consider the functions: $inc, dec, div : \mathbb{Z} \rightarrow \mathbb{Z}$ defined respectively by $\lambda x. x + 1$, $\lambda x. x - 1$ and $\lambda x. x \text{ div } 2$. Possible approximations for *Parity* and *Sign* are given by:

	$\perp$	od	ev	$\top$
inc_p^A	$\perp$	ev	od	$\top$
dec_p^A	$\perp$	ev	od	$\top$
div_p^A	$\perp$	$\top$	$\top$	$\top$

	$\perp$	0	$\hat{+}$	$\hat{-}$	$\top$
inc_s^A	$\perp$	$\hat{+}$	$\hat{+}$	$\top$	$\top$
dec_s^A	$\perp$	$\hat{-}$	$\top$	$\hat{-}$	$\top$
div_s^A	$\perp$	0	$\hat{+}$	$\hat{-}$	$\top$

Concrete semantics are typically defined as least fixed points of an operator on programs. Typically, the meaning of a program P may be expressed as $\llbracket P \rrbracket = lfp(f_P)$ where $f_P : Den \rightarrow Den$ is a monotonic operator on a domain of denotations Den . A program analysis will typically be defined by introducing an appropriate Galois insertion $(Den, \alpha, Den^A, \gamma)$ and constructing an approximation $f_P^A : Den^A \rightarrow Den^A$ of f_P so that the least fixed point of f_P^A is finitely computable. This construction often takes a systematic approach which involves replacing the basic operations in the concrete semantic operator f_P by corresponding abstract operations in f_P^A (e.g., [10, 21]). Given that these abstract operations approximate the concrete operations it is generally straightforward to prove that the derived abstract semantic operator approximates the concrete semantic operator. The fundamental theorem of abstract interpretation provides the following result:

¹Or alternatively if $\alpha(e) \sqsubseteq_D d$.

Theorem 2.3

Let (E, α, D, γ) be a Galois insertion and let $\mu : E \rightarrow E$ and $\mu^A : D \rightarrow D$ be monotonic functions such that μ^A γ -approximates μ . Then $lfp(\mu^A) \propto_\gamma lfp(\mu)$.

The “art” of abstract interpretation can be described as involving the following steps: (1) to choose an appropriate concrete semantics; (2) to identify a suitable notion of data-description; and (3) to provide good approximations of the basic operations in the concrete semantics. Once this is done the foundation is laid for deriving, more or less automatically, a semantic based program analysis. Applying suitable optimizations to the fixpoint algorithm used in the description domain, an analysis that is also efficient can be built essentially automatically from it [2, 20]. In the case of logic programs the main step is to provide a notion of abstract substitutions and an abstract unification algorithm. Other operations include “projection” and “composition” which safely project (i.e., on a finite set of variables) and compose descriptions.

The subject of this paper is centered around the practicality of: Given an appropriate concrete semantics –point (1) above– and having found two or more notions of description –point (2) above– together with corresponding approximations of the basic operations in the concrete semantics –point (3) above–, automatically constructing an approximation of the basic operations for a combined notion of description. Given this construction a combined analysis is derived by abstracting the concrete semantics.

Direct product analysis

Let E be a concrete domain and let $(E, \alpha_i, D_i, \gamma_i)$ $i \in \{1, 2\}$ be Galois insertions. The *direct product domain* is a quadruple $(E, \alpha_x, D, \gamma_x)$ where $D = D_1 \times D_2$, $\gamma_x : D \rightarrow E$ is defined by $\lambda(d_1, d_2). \gamma_1(d_1) \sqcap_E \gamma_2(d_2)$ and $\alpha_x : E \rightarrow D$ is defined by $\lambda e. (\alpha_1(e), \alpha_2(e))$.

The direct product domain is not a Galois insertion. Consider for example the domain *Parity* $\times$ *Sign* (see Example 1). Observe that

$$\alpha_x(\gamma_x(\top, 0)) = \alpha_x(\{0\}) = (ev, 0)$$

which is in violation of Definition 2.1. However, given a function $\mu : E \rightarrow E$ and corresponding γ_i -approximations $\mu_i^A : D_i \rightarrow D_i$ for $i \in \{1, 2\}$, the *direct product function* $\mu_x^A : D \rightarrow D$ defined by $\lambda(d_1, d_2). (\mu_1^A(d_1), \mu_2^A(d_2))$ is a γ_x -approximation of μ . The direct product function corresponds to performing the independent analyses μ_1^A and μ_2^A .

Following [9] we proceed to lift the direct product domain by considering the equivalence relation induced by γ_x . This provides a Galois insertion. Moreover, sharper analyses can be obtained by performing operations on the new *reduced product domain*.

Reduced product analysis

Let $(E, \alpha_i, D_i, \gamma_i)$, $i \in \{1, 2\}$ be Galois insertions and let $(E, \alpha_x, D, \gamma_x)$ be the corresponding direct product domain. The relation $\equiv \subseteq D \times D$ induced by γ_x is defined by $d \equiv d' \Leftrightarrow \gamma_x(d) = \gamma_x(d')$. The *reduced product domain* is a quadruple $(E, \alpha_*, D_*, \gamma_*)$ where $\alpha_* : E \rightarrow D_*$ is defined by $\lambda e. [\alpha_x(e)]_\equiv$ and $\gamma_* : D_* \rightarrow E$ is defined by $\lambda[d]. \gamma_x(d)$. It is straightforward to show that the reduced product domain is well defined and is a Galois insertion.

Let $\mu : E \rightarrow E$ be a concrete function and let $\mu_i^A : D_i \rightarrow D_i$, $i \in \{1, 2\}$, be corresponding γ_i -approximations. The *reduced product function* $\mu_*^A : D_* \rightarrow D_*$ is defined by

$$\lambda[d]_\equiv. [(\mu_1^A(d_1), \mu_2^A(d_2))]_\equiv$$

where $(d_1, d_2) = \sqcap_D [d]_\equiv$, namely the smallest representative of the equivalence class $[d]_\equiv$. It is straightforward to show that the reduced product function μ_*^A is well defined and is a γ_* -approximation of μ . Moreover, in general the reduced product function is no less precise than the direct product function. Example 3 illustrates that μ_*^A is potentially more precise than μ_x^A . The reduced product function corresponds to performing the original analyses over the reduced product domain. Namely, over a domain of representatives that contain no redundant information.

Example 3

Consider the following program fragment under the initial assumption that $x = 0$. A parity analysis will start with the abstract initial invariant $x = ev$ while a sign analysis will begin with the assumption $x = 0$. After considering the first program statement the direct and reduced product analyses give respectively $div_x^A(ev, 0) = \langle \top, 0 \rangle$ and $div_*^A(ev, 0) = \langle \top, 0 \rangle_\equiv$. In the reduced product domain $\langle \top, 0 \rangle_\equiv = \langle ev, 0 \rangle$ so after considering the second program statement the reduced product analysis gives $inc_*^A(\top, 0)_\equiv = \langle od, \dagger \rangle_\equiv$ which is more precise than the corresponding direct product analysis $inc_x^A(\top, 0) = \langle \top, \dagger \rangle$.

program fragment	direct product analysis for x	reduced product analysis for x
$\downarrow \{x=0\}$ $x := x \text{ div } 2$	$\{ \langle ev, 0 \rangle \}$	$\{ \langle ev, 0 \rangle \}_\equiv$
$\downarrow \{x=0\}$ $x := x + 1$	$\{ \langle \top, 0 \rangle \}$	$\{ \langle ev, 0 \rangle \}_\equiv$
$\downarrow \{x=1\}$	$\{ \langle \top, \dagger \rangle \}$	$\{ \langle od, \dagger \rangle \}_\equiv$

The previous example demonstrates how considering the interaction between the analysis domains can sharpen precision. However, further precision may be obtained by redefining the abstract operations:

Example 4

Define $dec^A : (Parity \times Sign)_\equiv \rightarrow (Parity \times Sign)_\equiv$ such that dec^A is the same as dec_*^A except that $dec^A(\langle od, \dagger \rangle) = \langle ev, \dagger \rangle$. This is clearly safe and provides a potentially sharper analysis.

program fragment	reduced product analysis	reduced product analysis with: $dec^A(\langle od, \dagger \rangle) = \langle ev, \dagger \rangle$
$\downarrow \{x=1\}$ $x := x - 1$	$\{ \langle od, \dagger \rangle \}_\equiv$	$\{ \langle od, \dagger \rangle \}_\equiv$
$\downarrow \{x=0\}$	$\{ \langle ev, \top \rangle \}_\equiv$	$\{ \langle ev, \dagger \rangle \}_\equiv$

We conclude this section with:

An example for logic programs

As a simple example for logic programs, we illustrate how Sndergaard's domain for sharing and groundness analysis [23] can be represented as the reduced product of corresponding sharing and groundness domains. The resulting reduced product analysis is equivalent to that derived from the abstract unification of Codish et al. [4] for this domain. First we provide some preliminary definitions:

Let Var denote an enumerable set of variables and $\text{PVar} \subseteq \text{Var}$ a distinguished (enumerable) set of variables which may occur in programs. Let Sub denote the set of idempotent substitutions. Informally, a set of program variables $\{x_1, \dots, x_n\} \subseteq \text{PVar}$ are *aliased* or *share* if in some execution of the program they may be bound to terms $t_1, \dots, t_n$, such that $\text{vars}(t_1) \cap \dots \cap \text{vars}(t_n) \neq \emptyset$. A program variable is *ground* if it is bound to a term t such that $\text{vars}(t) = \emptyset$. A program variable is *linear* if it is bound to a term which contains only single occurrences of variables.

unification	direct product analysis	reduced product analysis
$\begin{array}{c} \downarrow \\ \left\{ \begin{array}{l} X = U \\ Y = U \\ Z = g(U, V) \end{array} \right\} \\ \downarrow \end{array}$	$\langle \{ X \}, \{ \} \rangle$	$\langle \{ X \}, \{ \} \rangle$
	$\langle \{ X, Y \}, \{ (X, Y), (Y, Z), (X, Z) \} \rangle$	$\langle \{ X, Y \}, \{ \} \rangle$

Figure 2: Direct and reduced product analyses.

Definite groundness information is described by means of a set of program variables: $D_1 = 2^{\text{PVar}}$. Possible (pair) sharing information is described by symmetric binary relations on PVar : $D_2 = 2^{(\text{PVar} \times \text{PVar})}$. For a relation $R \in D_2$, xRy denotes that x and y are bound to terms which may share a variable, xRx denotes that x is bound to a possibly non-linear term. For convenience we will let an arbitrary relation R on PVar denote the smallest symmetric relation which contains R .

Example 5

Consider the (abstract) unification

$$h(U, U, g(U, V)) = h(X, Y, Z)$$

under the abstract substitutions $\{ X \} \in D_1$ and $\emptyset \in D_2$ specifying that X is definitely ground and that there is no possible sharing between the other variables which are definitely linear. A simple groundness analysis will determine that after the unification the variables $\{ X, Y, U \}$ will be ground. A (pair) sharing analysis which does not consider information in D_1 may determine that after unification there is at most sharing between

$$\left\{ \begin{array}{l} (X, Y), (Y, Z), (X, Z), (X, U), \\ (Y, U), (Z, U), (Z, V) \end{array} \right\}$$

The same analyses performed on a reduced product domain will eliminate the sharing on ground variables. Figure 2 illustrates the results of these analyses reflected on the variables $\{ X, Y, Z \}$. Such an analysis is described in [4] as a formalization of [23].

In the following section we present a more complex example and propose an approach to provide better precision by removing redundancies at intermediate lower level steps of abstract unification.

3 Combining set sharing with pair sharing

The following illustrates the practical benefit of combining domains for a somewhat more realistic example. We consider the combination of two different sharing analyses, one over the domain ASub of Søndergaard [23] and another one over the domain Sharing of Jacobs and Langen [16].

The domain ASub of Søndergaard is that described above as the reduced product $D_1 \star D_2$. An abstract substitution $(G, R) \in \text{ASub}$ describes those substitutions which make (at least) all the variables in G ground and have no more (pair) sharing than specified by R . The concretization function, $\gamma_{\text{ASub}} : \text{ASub} \rightarrow 2^{\text{Sub}}$, is defined by

$$\gamma_{\text{ASub}}(G, R) = \left\{ \theta \mid \begin{array}{l} \forall (x, y) \in \text{PVar}^2 : \\ (x \in G \Rightarrow \text{ground}(x\theta)) \wedge \\ (x \neq y \wedge \text{vars}(x\theta) \cap \text{vars}(y\theta) \neq \emptyset \Rightarrow \\ x R y) \wedge \\ (x \notin G \Rightarrow \text{linear}(x\theta)) \end{array} \right\}$$

As mentioned before a description of abstract operations for this domain can be found in [4].

The $\text{Sharing} = 2^{2^{\text{PVar}}}$ domain of Jacobs and Langen keeps track of set sharing. The concretization function is defined in terms of the occurrences of a variable U in a substitution:

$$\text{occs}(\theta, U) = \{X \in \text{dom}(\theta) \mid U \in \text{vars}(X\theta)\}.$$

If $\text{occs}(\theta, U) = V$ then θ maps the variables in V to terms which share the variable U . The concretization function $\gamma_{\text{Sharing}} : \text{Sharing} \rightarrow 2^{\text{Sub}}$ is defined as follows:

$$\gamma_{\text{Sharing}}(\kappa) = \{\theta \in \text{Sub} \mid \forall U \in \text{Var}. \text{occs}(\theta, U) \in \kappa\}.$$

unification	ASub	Sharing
$\downarrow$ $\left\{ \begin{array}{l} X = a \\ Y = f(A, B) \\ Z = C \end{array} \right\}$ $\downarrow$	$\langle \{ \}, \left\{ \begin{array}{l} (X, Y), (Y, Z), \\ (Z, X) \end{array} \right\} \rangle$ $\langle \{ X \}, \left\{ \begin{array}{l} (Y, A), (Y, B), (Z, C), \\ (A, Z), (B, Z), (Y, Z), \\ (A, C), (B, C), (Y, C) \end{array} \right\} \rangle$	$\left\{ \begin{array}{l} \emptyset, \{X, Y, Z\}, \{Y\}, \\ \{Z\}, \{A\}, \{B\}, \{C\} \end{array} \right\}$ $\left\{ \begin{array}{l} \emptyset, \{Y, A\}, \{Y, B\}, \\ \boxed{\{Y, A, B\}}, \{Z, C\} \end{array} \right\}$

Figure 3: Pair and set sharing analyses.

As mentioned before, the abstract substitution is composed of sets of program variables. Intuitively, each set in the abstract substitution containing variables $v_1, \dots, v_n$ represents the fact that there may be one or more shared variables occurring in the terms to which $v_1, \dots, v_n$ are bound. If a variable v does not occur in any set, then there is no variable that may occur in the terms to which v is bound and thus those terms are definitely ground. If a variable v appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term.

The advantage of the ASub domain is that it captures information about linearity which is not captured in Sharing. In ASub whenever a term is known to be linear it is possible to infer that the variables contained in the term do not share, while in Sharing such sharing must be assumed. On the other hand, the Sharing domain is more powerful in the way groundness is propagated between variables. The reason is that Sharing not only represents when two terms possibly share, but also which variables are possibly shared and which are definitely not shared. Thus, it can represent that a set of terms share all their variables, and therefore infer the groundness of one term from the groundness of the others.

These differences make the two abstract domains incomparable in the sense that each gives better results for some programs [7]. Several attempts have been made to enrich one domain or the other to give better results [6, 24]. Other combinations focus on adding other types of information while at the same time improving the sharing information [18]. These attempts all imply redefinition of the basic operations for the new domains. We propose the reduced product domain as the straightforward way to achieve the “best” possible information given two analyses. Moreover, we claim that reasonable precision can be obtained without redefining the abstract unification algorithms.

It is important to note that since the reduced product is defined for two abstract domains and their abstraction (or concretization) functions, once the reduced product has been determined for them, it can be used with any abstract unification algorithm defined for the original abstract domains. In fact, in the following examples and in the evaluation of the technique given in Section 4, the abstract unification algorithms used are the respectively improved versions [4, 20] of the abstract algorithms originally given for each domain.

Example 6

Consider the unification

$$p(X, Y, Z) = p(a, f(A, B), C)$$

where the call pattern is of the form

$$\{ X \mapsto U, Y \mapsto f(U, V), Z \mapsto f(U, W) \}.$$

Namely, X , Y and Z are bound to linear terms which share a common variable. Figure 3 illustrates the results of abstract unification for the domains ASub and Sharing. The square boxes indicate redundant information.

The Sharing analysis indicates that the pairs $(Y, Z), (Y, C), (A, Z), (A, C), (B, Z), (B, C)$ in the ASub analysis are redundant as they are not subsets of a set obtained by the Sharing analysis. On the other hand the ASub analysis indicates that the set $\{Y, A, B\}$ in the Sharing analysis is redundant as (A, B) is not in ASub. Hence a reduced product analysis would result in $\{ \{X\}, \{(Y, A), (Y, B), (Z, C)\} \}$ for ASub and $\{ \emptyset, \{Y, A\}, \{Y, B\}, \{Z, C\} \}$ for Sharing.

The function **Reduce** yields the minimal representative of an element of the reduced product given an arbitrary representative.

Definition 3.1 $\text{Reduce} : \text{ASub} \times \text{Sharing} \rightarrow \text{ASub} \times \text{Sharing} : \langle (G, R), S \rangle \mapsto \langle (G', R'), S' \rangle$ where

$$S' = \{s \in S \mid s \cap G = \emptyset, \text{Pairs}(s) \subseteq R\},$$

$$R' = R \cap \left(\bigcup_{s \in S'} s \times s \right),$$

$$G' = \text{ground}(S')$$

$$\text{Pairs}(s) = \{(X, Y) \in s \times s \mid X \neq Y\}.$$

The idea is that **Reduce** removes redundancies from the representation of an abstract substitution while preserving its meaning. This is achieved by: (1) eliminating from S those sets which indicate sharing not present in (G, R) , obtaining S' ; (2) eliminating from R those pairs which indicate sharing not present in S' , obtaining R' ; and (3) deriving those variables which are ground according to S' , obtaining G' .

Note that the **Reduce** function can be extended in a straightforward way for the combination of abstract domains that are enhancements of the **ASub** and **Sharing** domains respectively. This is the case, for example, with the **Sharing+Freeness** abstract domain defined in [18], in which the **Sharing** domain is enriched with freeness information by adding a second component which represents the set of definitely free variables. Both combinations of **ASub** with **Sharing** and of **ASub** with **Sharing+Freeness** are illustrated in Section 4.

An additional gain in precision can be obtained if redundant information is removed not only after each basic operation (abstract unification), but also at intermediate steps inside the implementation of the corresponding algorithms. We illustrate this point for the combination of the **ASub** and **Sharing** domains. The abstract unification algorithms for the **ASub** and **Sharing** domains defined in [4, 16] both follow the same basic strategy when solving an abstract unification $E\delta$ consisting of an equation E and an abstract substitution δ . Namely, first reducing the equation E to a solved form $\text{mgu}(E) = \{e_1, \dots, e_n\}$, and then solving each of the abstract equations $e_i\delta_i$ in turn with $\delta_1 = \delta$ and δ_{i+1} the solution of $e_i\delta_i$. Consequently, the **Reduce** function can be applied at the intermediate steps after solving each equation in the process. Intuitively one expects that choosing the finest granularity will yield the most precise results and that removing redundancies as early as possible will be beneficial computationally. The following example demonstrates this point.

Example 7 Consider

$E = \{p(W, X, Y, Z) = p(f(A, B), f(a, a), A, B)\}$, with $\text{mgu}(E) = \{W = f(A, B), X = f(a, a), Y = A, Z = B\}$, call substitutions $d_1 = \{\emptyset, \{(X, Y), (Y, Z), (Z, X)\}\} \in \text{ASub}$ and $d_2 = \{\emptyset, \{X, Y, Z, \{X\}, \{Y\}, \{Z\}, \{W\}, \{A\}, \{B\}\} \in \text{Sharing}$.

Figure 4 contains the results at the intermediate steps of the (combined) abstract unification algorithm in the case of a direct product analysis. The sharing information in boxes will be removed if the **Reduce** function is applied at the intermediate steps in the algorithm. Note that if the **Reduce** function is applied only at the end of the abstract unification then the result is less precise as the sharing between Y and Z is possible according to both the final **ASub** and **Sharing** components. Hence, removing redundancies at intermediate steps is important to improve both the precision and efficiency of the abstract unification.

4 Evaluation

This section presents and compares the analysis results obtained for the following domains:

analyser	domain
P	ASub(pair sharing)
S	Sharing(set sharing)
SF	Sharing+Freeness
$P \star S$	reduced product
$P \star SF$	reduced product

The respective analyses are based on the following implementations:

P on the algorithms defined in [4];

S on the algorithms described in [20]; and

SF on algorithms defined in [18, 19].

The $P \star S$ and $P \star SF$ reduced product analyses are implemented using the **Reduce** function. The analyses have been carried out within the framework of the &-Prolog compiler. This framework, implemented in Prolog, is based on the abstract interpretation framework of Bruynooghe described in [2], optimized with the specialized domain-independent fixpoint algorithm defined in [20]. The type of analysis to be performed is a parameter of the system passed to the fixpoint algorithm which in turn calls the appropriate abstract operators. The system allows the comparison of precision of different analyses as well as of their relative efficiency.

The abstract operations for the domains S and SF were already supported by the existing implementation of the framework. In order to provide the results for the combined analyses, an implementation of the operations for the P domain was added. Once this was done, integrating the $P \star S$ and $P \star SF$ analyses in the system required a few additional lines of code which, each time an abstract function is called by the fixpoint algorithm, calls the corresponding abstract functions (e.g., for P and for S), performs the

equations	pair × set sharing
$\downarrow$ $\left\{ \begin{array}{l} W = f(A, B), \\ X = f(a, a), \\ Y = A, Z = B \end{array} \right\}$ $\downarrow$ $\left\{ \begin{array}{l} W = f(A, B), \\ X = f(a, a), \\ Y = A, Z = B \end{array} \right\}$ $\downarrow$ $\left\{ \begin{array}{l} W = f(A, B), \\ X = f(a, a), \\ Y = A, Z = B \end{array} \right\}$ $\downarrow$ $\left\{ \begin{array}{l} W = f(A, B), \\ X = f(a, a), \\ Y = A, Z = B \end{array} \right\}$	$\emptyset, \{ (X, Y), (Y, Z), (X, Z) \}, \left\{ \begin{array}{l} \emptyset, \{X, Y, Z\}, \{X\}, \{Y\}, \\ \{Z\}, \{W\}, \{A\}, \{B\} \end{array} \right\}$ $\emptyset, \left\{ \begin{array}{l} (X, Y), (Y, Z), (X, Z), \\ (A, W), (B, W) \end{array} \right\}, \left\{ \begin{array}{l} \emptyset, \{X, Y, Z\}, \{X\}, \{Y\}, \{Z\}, \\ \{A, W\}, \{B, W\}, \boxed{\{A, B, W\}} \end{array} \right\}$ $\{X\}, \left\{ \begin{array}{l} (A, W), (B, W), \boxed{(Y, Z)} \end{array} \right\}, \left\{ \begin{array}{l} \emptyset, \{Y\}, \{Z\}, \{A, W\}, \\ \{B, W\}, \boxed{\{A, B, W\}} \end{array} \right\}$ $\{X\}, \left\{ \begin{array}{l} (A, W), (B, W), (A, Y), (W, Y), \\ \boxed{(Y, Z), (A, Z), (W, Z)} \end{array} \right\}, \left\{ \begin{array}{l} \emptyset, \{Z\}, \{B, W\}, \{A, W, Y\}, \\ \boxed{\{A, B, W, Y\}} \end{array} \right\}$ $\{X\}, \left\{ \begin{array}{l} (A, W), (B, W), (A, Y), \\ (W, Y), (B, Z), (W, Z), \\ \boxed{(Y, Z), (A, Z), (A, B), \\ (B, Y), (W, W)} \end{array} \right\}, \left\{ \begin{array}{l} \emptyset, \{B, W, Z\}, \{A, W, Y\}, \\ \boxed{\{A, B, W, Y, Z\}} \end{array} \right\}$

Figure 4: Applying Reduce at intermediate steps.

Reduce function over the information inferred by each analyser, and returns the resulting information to the fixpoint algorithm. In the present implementation the basic step after which the **Reduce** function is applied is a full abstract unification.

The framework is based on a collecting semantics which specifies not only answer substitutions for the initial goal but also the intermediate bindings of variables before and after each call in the body of a clause. In fact, the framework may be viewed as providing approximations of variable bindings at the respective “program points.” Approximations of these intermediate bindings are relevant for many applications such as, for example, program parallelization.

When comparing the accuracy of the various analyses we consider a variety of criteria including information about groundness, linearity, pair sharing and set sharing. We note that the pair sharing in an element of **Sharing** is obtained by considering all possible pairs as specified by the **Pairs** function given in Definition 3.1. Similarly, the set sharing in an element of **ASub** is obtained by computing the “closure under

union” operation of the *S* unification algorithm and removing the redundant sets as done in the computation of *S'* in Definition 3.1.

Table 1 lists the programs analysed, notes their sizes and shows the analysis times in milliseconds (Sparc-Station IPC, Sicstus 2.1, native code). The programs are:

- serialize** a program to manipulate lists of numbers;
- init-subst** initialise two abstract substitutions to have the same set of variables;
- map-color** a map coloring algorithm;
- grammar** a program which generates and recognizes a small set of english;
- browse** which is in the Gabriel benchmarks (as translated by T. Dobry and H. Touati);
- bid** which computes an opening bid for a bridge hand;
- deriv** a program which performs symbolic differentiation;

Program sizes				Analysis times in milliseconds				
Program	# clauses	# expanded clauses	# vars	S	P	SF	P*S	P*SF
serialize	12	12	44	9290	839	2620	1870	2530
init-subst	12	14	53	1250	569	660	829	1080
map-color	13	13	25	4600	1040	1629	5760	2939
grammar	16	16	17	170	140	250	269	349
browse	32	38	115	51860	1609	25559	49590	29549
bid	53	53	98	1129	1000	1259	1429	1759
deriv	62	62	170	2819	2630	3119	3550	4289
rdtok	55	68	196	5670	4450	6879	6389	11510
read	88	92	344	8790	8380	9760	11069	12919
boyer	138	146	118	11040	3949	14600	7709	10480
peephole	134	155	357	20760	7990	14890	23029	25589
ann	181	222	594	93509	16789	44639	53269	65269

Table 1: Program sizes and analysis times

rdtok O’Keefe’s public domain Prolog tokeniser;
read Warren and O’Keefe’s public domain Prolog parser;
boyer a theorem prover also in the Gabriel Benchmarks (as translated by E. Tick);
peephole SB-Prolog’s Peephole optimizer; and
ann a simplified version of &-Prolog’s parallelizing annotator.

Size measures include: the number of clauses in the program, the number of clauses after preprocessing² and the number of variables in the program³.

Example 8

The following illustrates the output from the reduced product analysis (P*S) for one clause of the **serialize** program.

```

arrange([X|L],tree(T1,X,T2)) :-
  %S [[X],[L],[T1],[T2],[L1],[L2]]
  %P [],[]
  split(L,X,L1,L2),
  %S [[X],[X,L],[L,L1],[L,L2],[T1],[T2]]
  %P [],[[L,L],[X,L],[L,L1],[L,L2]]
  arrange(L1,T1),
  %S [[X],[X,L],[L,T1,L1],[L,L2],[T2]]
  %P [],[[L,L],[L1,L1],[L,T1],[L,L1],[L,L2],
  %      [X,L],[T1,L1]]
  arrange(L2,T2).
```

²The abstract framework opens up the clauses in which “if-then-else” and “or” structures appear. Therefore, the real number of analysed clauses may be greater than in the original program.

³We do not count variables in facts because the information for facts is actually represented in the clause or clauses that call them, in the abstract substitution just before and after such calls.

```

%S [[X],[X,L],[L,T1,L1],[L,T2,L2]]
%P [],[[L,L],[L1,L1],[L2,L2],[X,L],[L,T1],
%      [L,T2],[L,L1],[L,L2],[T1,L1],[T2,L2]]
```

The results of the analysis are indicated as comments within the text of the program. In this case the %S and %P indicate respectively information from the Sharing and ASub domains. This indicates a total of: 0 (definitely) ground variables, 6 (possibly) non-linear variables, 15 (possibly) shared pairs and 9 (possible) set sharings.

In contrast, performing only the set sharing analysis S for the same clause gives the following output indicating the less precise possibility of 65 shared sets:

```

arrange([X|L],tree(T1,X,T2)) :-
  %S [[X],[X,L],[X,L,T1],[X,L,T1,T2],[X,L,T2],
  %   [X,T1],[X,T1,T2],[X,T2],[L],[L,T1],[L,T1,
  %   T2],[L,T2],[T1],[T1,T2],[T2],[L1],[L2]]
  split(L,X,L1,L2),
  %S [[X],[X,L],[X,L,T1],[X,L,T1,T2],[X,L,T1,
  %   T2,L1],[X,L,T1,T2,L1,L2],[X,L,T1,T2,L2],
  %   [X,L,T1,L1],[X,L,T1,L1,L2],[X,L,T1,L2],
  %   [X,L,T2],[X,L,T2,L1],[X,L,T2,L1,L2],[X,L,
  %   T2,L2],[X,L,L1],[X,L,L1,L2],[X,L,L2],[X,
  %   T1],[X,T1,T2],[X,T2],[L,T1,T2,L1],[L,T1,T2,
  %   L1,L2],[L,T1,T2,L2],[L,T1,L1],[L,T1,L1,L2],
  %   [L,T1,L2],[L,T2,L1],[L,T2,L1,L2],[L,T2,L2],
  %   [L,L1],[L,L1,L2],[L,L2],[T1],[T1,T2],[T2]]
  arrange(L1,T1),
  %S [[X],[X,L],[X,L,T1,T2,L1],[X,L,T1,T2,
  %   L1,L2],[X,L,T1,L1],[X,L,T1,L1,L2],[X,L,T2],
  %   [X,L,T2,L2],[X,L,L2],[X,T2],[L,T1,T2,L1],
  %   [L,T1,T2,L1,L2],[L,T1,L1],[L,T1,L1,L2],
  %   [L,T2,L2],[L,L2],[T2]]
  arrange(L2,T2),
  %S [[X],[X,L],[X,L,T1,T2,L1,L2],[X,L,T1,L1],
  %   [X,L,T2,L2],[L,T1,T2,L1,L2],[L,T1,L1],
  %   [L,T2,L2]]
```

Results

The following two tables⁴ show the total number of pair and set sharing for the *S*, *P* and *SF* analysers and for the reduced product analyses. For pair sharing the respective reduced product analyses basically do no better than the *P* analysis. Similar results are obtained when measuring the pair sharing only in answer substitutions.

Total # of shared pairs					
Program	S	P	SF	P*S	P*SF
serialize	235	35	137	35	35
init-subst	5	72	5	5	5
map-color	76	74	73	73	73
grammar	11	11	11	11	11
browse	196	104	167	104	104
bid	11	0	0	0	0
deriv	0	0	0	0	0
rdtok	185	48	51	48	48
read	11	1	1	1	1
boyer	242	93	222	93	93
peephole	386	310	310	310	310
ann	1935	1690	1694	1690	1690

Total set shared variables					
Program	S	P	SF	P*S	P*SF
serialize	502	41	208	24	24
init-subst	5	92	5	5	5
map-color	108	145	101	101	101
grammar	5	14	5	5	5
browse	671	606	628	547	547
bid	17	0	0	0	0
deriv	0	0	0	0	0
rdtok	219	57	47	44	44
read	12	1	1	1	1
boyer	417	132	375	100	100
peephole	623	579	417	417	417
ann	3230	6447	2543	2543	2543

In addition to sharing we have found that for our benchmarks, with one exception, all of the analysers give almost identical groundness information. For the **init-subst** benchmark the set sharing domain derives 126 definitely ground occurrences of variables while the pair sharing domain finds only 33 (the combined domain analysis derives also 126 ground occurrences of variables). In addition we found that the reduced product analyses do not improve the linearity derived by the *P* analysis.

⁴Note that the numbers here refer to the amount of possible sharing and hence more precise analyses indicate less sharing.

Discussion

The first observation is that the results obtained for the reduced product analyses are always equal or better than the ones obtained by the separate analyses. Thus, the combined analyses are quite successful at incorporating the good qualities of each of the components. It is also interesting to note that *P*SF* does not improve the results of *P*S* in terms of sharing (although it of course provides in addition freeness information). This is not surprising as freeness information provides only a restricted form of linearity information, i.e. that obtained from the knowledge that any free variable is also a linear term. Thus, *P*S* seems an excellent sharing analysis for the benchmarks used.

Although not of direct relevance to domain combination issues the results also allow a quite interesting comparison of the basic analyses, *P*, *S* and *SF*. Observing the results for pair sharing we see that linearity information (present in *P*, not present in *S*, and partially present in *SF*) proves to be a powerful instrument for increasing the accuracy of sharing analyses.

In terms of set sharing the situation is more complex. Despite the lack of linearity information *S* computes in some cases better set sharing information than *P*, but still *P* does better in others. Freeness information again improves the results of *S* but there are still cases in which *P* does better than *SF*. This gives more window of opportunity to the combined domain *P*S*. It is strictly more powerful than any of the others (except *P*SF*, of course) but the difference shows up most in terms of set sharing.

Although it is easy to contrive examples for which the domains which capture set sharing provide a more powerful groundness propagation, it is interesting to note that in practice (as far as we have observed) all of the domains provide almost identical groundness information. This is partially due to the fact that our analyses are goal dependent and because in most programs groundness typically propagates in a top-down, left-to-right direction. The **init-subst** benchmark is an exception as it contains a predicate in which groundness information propagates from right-to-left even when it is called with the most instantiated query mode. This is due to the way the program is written to take advantage of tail recursion. We believe that this phenomenon may actually show up more often in actual applications in which efficiency has been a major consideration during coding, and also when partially instantiated structures are used. Also, we expect the difference between the base domains with respect to groundness propagation and consequently with respect to both types of sharing to be more notable when performing goal independent analyses [1, 3]. It is our belief that for such analyses, combining domains is even more beneficial.

The analysis times also provide some interesting insight into the approach taken towards combining domains. The time taken for the combined analyses is always less than the sum of that taken by each of the analyses by themselves. This shows one more advantage of combining rather than just running each analysis separately (in addition to the possibly higher accuracy): steps of the fixpoint algorithm are shared and the total time is decreased. But it is also quite interesting to note that the total time is sometimes actually *much* less than the sum of the times. This is due to two effects. The first one is that if the unification time for each analysis is not very high, then the time for the reduced analysis is not increased much because the fixpoint algorithm steps, which are shared, will then dominate the analysis time. This is specially the case in our implementation due to the heavy use of the database that is made for the fixpoint algorithm. The second reason is more interesting: because of the additional precision obtained with the combined analysis the abstract substitutions are potentially smaller and thus less time is taken in the abstract unifications. Furthermore, the additional precision can also reduce the number of steps in the fixpoint computation. This can be observed by comparing the analysis times of $P \star S$ and $P \star SF$ for **browse**.

5 Conclusions and future work

We have shown how in practice it is possible to maintain precision in a combined analysis and obtain reasonable analysis times without redefining the basic operations. We have also proposed a way of performing the combination which can be more precise than the “reduced product” approach, while keeping the original components of the basic operations. The advantage of the general approach is that proofs of correctness for the new domains are not required and implementations can be reused. In order to illustrate this we have implemented a series of sharing analyses previously proposed and constructed two new ones as combinations of these. There are strong indications that our automatically combined analyses in fact compare well with other new proposals suggested in recent literature [5, 24] both from the point of view of efficiency and accuracy. Moreover, we plan to further improve our results by breaking abstract unification up into individual steps and applying the **Reduce** function after each simple unification step as discussed in Section 3.

An important insight acquired from this work is the realization of the ease in practice of the combination process, which certainly required a much smaller amount of work than that taken by the original analysers also implemented by us. This is of practical importance because the precision and efficiency of many

analyses can be improved by combining various standard domains such as those described in this paper. In many cases the improvement in efficiency is crucial for practical implementations. We are currently applying this approach to a number of other domains.

Acknowledgments

We would like to thank Renganathan Sundararajan for providing some of the benchmarks used in this study and Dennis Dams for his comments on an earlier version of this paper.

References

- [1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. To appear in *ACM Transactions on Programming Languages and Systems*.
- [2] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.
- [3] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. To appear in *Theoretical Computer Science*.
- [4] M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming*, pages 79–93, Paris, France, June 1991. MIT Press.
- [5] A. Cortesi and G. Filé. Comparing and improving: Abstract domains for sharing analysis. Technical Report, Department of Mathematics, University of Padova.
- [6] A. Cortesi and G. Filé. Freeness computation in abstract interpretation. *Rapporto Interno 2*, Dip. di Matematica Pura e Applicata, Università di Padova, March 1992.
- [7] A. Cortesi and G. Filé and Will Winsborough. Comparison of Abstract Interpretations. In *Nineteenth International Colloquium on Automata, Languages, and Programming*, LNCS 623, 1992.
- [8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

- [9] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.
- [10] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179. North-Holland, July 1992.
- [11] S.K. Debray, Guest editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, 13(2 and 3). North-Holland, July 1992.
- [12] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):205–258, July 1992.
- [13] M. Hermenegildo. Abstract Interpretation and its Applications: Notes for an Advanced Tutorial. 1992 Joint International Conference and Symposium on Logic Programming, University of Maryland, November 1992.
- [14] M. Hermenegildo and K. Greene. $\&$ -Prolog and its Performance: Exploiting Independent And-Parallelism. In *Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, Israel, June 1990. MIT Press.
- [15] K. Horiuchi. Less Abstract Semantics for Abstract Interpretation of FGHC Programs. In *1992 International Conference on Fifth Generation Computer Systems*, pages 897–906, Tokyo, Japan, June 1992. ICOT.
- [16] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314. North-Holland, July 1992.
- [17] N. D. Jones and H. Søndergaard. A semantic-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood, Chichester, 1987.
- [18] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 49–63, Paris, France, June 1991. MIT Press.
- [19] K. Muthukumar and M. Garcia de la Banda and M. Hermenegildo. Sharing and freeness analysis of logic programs using abstract interpretation. Technical report, U. of Madrid (UPM), Facultad Informatica UPM, Madrid, Spain, November 1992.
- [20] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347. North-Holland, July 1992.
- [21] F. Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76:29–92, 1988.
- [22] H. Søndergaard. *Semantics-based Analysis and Transformation of Logic Programs*. PhD thesis, DIKU, University of Copenhagen, 1990.
- [23] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *ESOP'86 Proceedings European Symposium on Programming*, Lecture Notes in Computer Science 213, pages 327–338. Springer-Verlag, N.Y., 1986.
- [24] R. Sundararajan and J. S. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *Twelfth FST & TCS Conference*, New Delhi, India, December 1992.