Efficient Computation of the Reduced Product of Groundness and Sharing Analyses

Lunjin Lu
Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309, USA

Abstract. We identify situations where computation of the reduced product of program analyses can be made more efficient and present a method for efficiently computing the reduced products of program analyses in these situations. The method gives rise to big efficiency gains when applied to computing the reduced product of groundness and pair sharing analyses which provide indispensable information for parallelizing compilation of logic programs.

Keywords: Reduced product, Groundness, Sharing, And-Parallelism

1 Introduction

Information about possible sharing between variables in a logic program is required for a number of important optimizations. In *Independent And-Parallelism* goals may be executed in parallel only if they do not share common variables [6, 12]. Information about definite groundness of variables is required for specializing the unification operation used in the execution of the program. There has been much research into groundness and sharing analyses and their application in parallelizing compilation of logic programs [14, 11, 7, 12, 3, 2, 13, 17]. Groundness analysis and sharing analysis are often combined using the reduced product method [9] so as to derive more precise sharing information. In this paper, we revisit the problem of computing the reduced product of two program analyses in the framework of abstract interpretation in general and the problem of computing the reduced product of the groundness and sharing analyses in particular.

Abstract interpretation is a methodology for static program analysis. Program analyses are viewed as program execution over non-standard data domains. Cousot and Cousot first laid solid logical foundations for abstract interpretation [8, 9]. Their idea is to define a collecting semantics for a program which associates each program point with the set of all storage states that are possibly obtained when the execution reaches the point. Then, an abstraction of the collecting semantics is calculated by simulating over a non-standard data domain the computation of the collecting semantics over the standard data domain. The standard and the non-standard domains are called the concrete domain and the abstract domain respectively.

Designing an analysis is a complex process. The quest for simplifying this process has resulted in systematic methods for constructing abstract domains and program analyses. One important way to compose abstract domains is the reduced product approach [9]. The reduced product of two abstract domains captures information represented by the two component abstract domains without redundancy.

The reduced product approach improves the precision of component analyses because information for one component can be used to improve the precision of another. Precision gains in the reduced product approach stem from mutual improving synergism between component analyses. However, there are situations where the reduced product approach improves one component analysis but not the other. This paper makes use of this asymmetry between component analyses to speed up the computation of their reduced product. It is based on a simple principle that the component analysis which is not improved by the other is computed the first and its result is then used to improve the other. We introduce a reduction relation between abstract domains and use the reduction relation to identify the situations where one analysis is not improved by the other analysis. The main contributions of this paper are the reduction relation and a method for computing reduced products of program analyses that makes use of the reduction relation. We also provide experimental results in computing the reduced product of groundness and pair sharing analyses which shows that our method can deliver big efficiency gains.

The rest of the paper is organised as follows. Section 2 contains basic notions that are used in later sections. Section 3 introduces the reduction relation between abstract domains and section 4 proposes a method to speed up the computation of the reduced products of analyses. Section 5 illustrates the method by computing the reduced product of groundness and pair sharing analyses. Section 6 concludes the paper with directions for future work.

2 Preliminaries

2.1 Abstract Interpretation Schemes

The semantics of a program is given by an interpretation $\langle(C,\sqsubseteq_C),\mathcal{C}\rangle$ where $(C,\sqsubseteq_C)$ is a complete lattice and $\mathcal{C}$ is a continuous function on $(C,\sqsubseteq_C)$. The semantics of the program is taken as the least fixpoint $\mathsf{lfp}\ \mathcal{C}$ of $\mathcal{C}$. The concrete semantics of the program is given by a concrete interpretation $\langle(C,\sqsubseteq_C),\mathcal{C}\rangle$ while an abstract semantics is given by an abstract interpretation $\langle(D,\sqsubseteq_D),\mathcal{D}\rangle$. Let $\circ$ denote function composition. The correspondence between $(C,\sqsubseteq_C)$ and $(D,\sqsubseteq_D)$ is formalised by a Galois connection (α_D,γ_D) where $\alpha_D : C \mapsto D$ and $\gamma_D : D \mapsto C$ are monotonic functions satisfying $\forall c \in C.c \sqsubseteq_C \gamma_D \circ \alpha_D(c)$ and $\forall d \in D.\alpha_D \circ \gamma_D(d) \sqsubseteq_D d$. The condition for $\mathsf{lfp}\ \mathcal{D}$ to be a safe abstraction of $\mathsf{lfp}\ \mathcal{C}$ is $\alpha_D \circ \mathcal{C} \circ \gamma_D \sqsubseteq_D \mathcal{D}$ [10]. In the Galois connection approach, an analysis can be formalised as an abstract interpretation scheme.

Definition 2.1 *An abstract interpretation scheme is a tuple*

$$\mathsf{D} = (\langle(C,\sqsubseteq_C),\mathcal{C}\rangle,\alpha_D,\gamma_D,\langle(D,\sqsubseteq_D),\mathcal{D}\rangle)$$

such that $\langle(C,\sqsubseteq_C),\mathcal{C}\rangle$ *and* $\langle(D,\sqsubseteq_D),\mathcal{D}\rangle$ *are interpretations,* (α_D,γ_D) *is a Galois connection between* $(C,\sqsubseteq_C)$ *and* $(D,\sqsubseteq_D)$*, and* $\alpha_D \circ \mathcal{C} \circ \gamma_D \sqsubseteq_D \mathcal{D}$.

2.2 Galois Insertion/Upper Closures

Instead of a Galois connection, a Galois insertion from D to C can be used. A Galois insertion from D to C is a Galois connection between D and C such that $\forall d \in D.\alpha_D \circ \gamma_D(d) = d$. A Galois insertion can be constructed from every Galois connection. The Galois insertion (α_D,γ_D) from D to C is denoted as $(\gamma_D, C, D, \alpha_D)$.

An alternative but equivalent approach to formalising an abstraction is by an upper closure operator on the domain C of data [9]. An upper closure operator ρ on C is a monotonic, idempotent and extensive function on C. $\rho(C)$ - the set of the fixpoints of ρ - is a Moore family. Let $U = uco(C)$ be the set of upper closure operators on C. Then $< U,\sqsubseteq_U,\sqcup_U,\sqcap_U,\perp_U,\top_U >$ is a complete lattice where $\rho_1 \sqsubseteq_U \rho_2 \stackrel{def}{=} \forall c \in C.\rho_1(c) \sqsubseteq_C \rho_2(c)$, and $\sqcup_U,\sqcap_U,\perp_U$ and $\top_U$ are respectively the least upper bound, the greatest lower bound, the infimum and the supremum.

2.3 Direct and Reduced Products

The reduced product approach is the most important one to designing a new abstract domain from two existing abstract domains. The reduced product of two Galois insertions $(\gamma_D, C, D, \alpha_D)$ and $(\gamma_E, C, E, \alpha_E)$ is $(\alpha_R, C, R, \gamma_R)$ obtained as follows. The denotation of a pair $(d,e) \in D \times E$ is $\gamma_D(d) \sqcap_C \gamma_E(e)$. The abstract domain $R \stackrel{def}{=} (D \times E)_{/\equiv_{D\times E}}$ is the quotient of $D \times E$ with respect to the equivalence $\equiv_{D\times E}$ of denotation with

$$\begin{array}{lcl} (d,e) \equiv_{D\times E} (d',e') & \stackrel{def}{=} & \\ & & \gamma_D(d) \sqcap_C \gamma_E(e) = \gamma_D(d') \sqcap_C \gamma_E(e') \\ \alpha_R(c) & \stackrel{def}{=} & [(\alpha_D(c),\alpha_E(c))]_{\equiv_{D\times E}} \\ \gamma_R([(d,e)]_{\equiv_{D\times E}}) & \stackrel{def}{=} & \gamma_D(d) \sqcap_C \gamma_E(e) \end{array}$$

Research into analysis design goes beyond the composition of abstract domains. The composition of semantic operators for the component analyses is also studied. [9, 5] studied two ways to combine component analyses into a new analysis, namely direct product analysis and reduced product analysis. Let

$$\begin{array}{lcl} \mathsf{D} & = & (\langle(C,\sqsubseteq_C),\mathcal{C}\rangle,\alpha_D,\gamma_D,\langle(D,\sqsubseteq_D),\mathcal{D}\rangle) \\ \mathsf{E} & = & (\langle(C,\sqsubseteq_C),\mathcal{C}\rangle,\alpha_E,\gamma_E,\langle(E,\sqsubseteq_E),\mathcal{E}\rangle) \end{array}$$

be two abstract interpretation schemes. The direct product of D and E is $\mathsf{P} = (\langle(C,\sqsubseteq_C),\mathcal{C}\rangle,\alpha_P,\gamma_P,\langle(P,\sqsubseteq_P),\mathcal{P}\rangle)$ obtained as follows. The abstract domain of P is the Cartesian product of those of D and E ordered pairwise.

$$\begin{array}{lcl} P & \stackrel{def}{=} & D \times E \\ \sqsubseteq_P & \stackrel{def}{=} & \lambda(d,e).\lambda(d',e').((d \sqsubseteq_D d') \wedge (e \sqsubseteq_E e')) \end{array}$$

The abstraction function of P is the product of those of D and E. $\alpha_P \stackrel{def}{=} \langle\alpha_D,\alpha_E\rangle$. The concretisation function of P applies those of D and

E separately and then takes the greatest lower bound of the denotations given by them. $\gamma_P \stackrel{def}{=} \sqcap_C \circ (\gamma_D \times \gamma_E)$. $\mathcal{P} \stackrel{def}{=} \mathcal{D} \times \mathcal{E}$. $(\gamma_P, C, P, \alpha_P)$ is a Galois connection but not necessarily a Galois insertion even if $(\gamma_D, C, D, \alpha_D)$ and $(\gamma_E, C, E, \alpha_E)$ are Galois insertions.

The abstract domain of the reduced product R of two analyses D and E [9, 5] is the reduced product of those of D and E. For an equivalence class r of $\equiv_{D\times E}$, the abstract operator $\mathcal{R}$ of R first chooses the least representative of r which is $\sqcap_P r$, and then applies $\mathcal{D} \times \mathcal{E}$ to $\sqcap_P r$ and returns the equivalence class of its result with respect to $\equiv_{D\times E}$. Formally, $\mathcal{R} = [\ ]_{\equiv_{D\times E}} \circ (\mathcal{D} \times \mathcal{E}) \circ \sqcap_P$. If an equivalence class r of $\equiv_{D\times E}$ is represented by its least representative $\sqcap_P r$, the reduced product abstract operator $\mathcal{R}$ on the equivalence class r has the same effect as $\sqcap_P \circ [\ ]_{\equiv_{D\times E}} \circ (\mathcal{D} \times \mathcal{E})$ on the least representative $\sqcap_P r$. The role of $red_{D\times E} \stackrel{def}{=} \sqcap_P \circ [\ ]_{\equiv_{D\times E}}$ is to reduce a pair in $D \times E$ into its least equivalent with respect to $\equiv_{D\times E}$ and is called a reduction function. The need for $\mathcal{R}$ to apply $\mathcal{D}\times\mathcal{E}$ to the least representative of an equivalence class of $\equiv_{D\times E}$ stems from the fact that $\equiv_{D\times E}$ are often not congruent with respect to $\mathcal{D} \times \mathcal{E}$. We have $red_{D\times E} = \alpha_P \circ \gamma_P$, which implies that $red_{D\times E}$ is a lower closure operator.

3 Reduction between abstract domains

Sometimes, one component description reduces the other but not vice verse. As will be shown later, the combination of pair sharing and groundness is such an example. The groundness description may be used to remove those pairs in the sharing description that have one ground variable according to the groundness description. However, the sharing description does not reduce the groundness description. We now formalise the fact that one component description *reduces* another component description, i.e., one component description makes another component description smaller while preserving the denotation of the combined description. Let $\mathsf{fst} : D \times E \mapsto D$ be defined as $\mathsf{fst} \stackrel{def}{=} \lambda(d,e).d$ and $\mathsf{snd} : D \times E \mapsto E$ be defined as $\mathsf{snd} \stackrel{def}{=} \lambda(d,e).e$.

Definition 3.1 *Let $d \in D$ and $e \in E$. If $\mathsf{snd} \circ red_{D\times E}(d,e) \neq e$ then we say d reduces e and write $d \searrow e$. Similarly, if $\mathsf{fst} \circ red_{D\times E}(d,e) \neq d$ then we say e reduces d and write $e \searrow d$. Formally,*

$$d \searrow e \stackrel{\text{def}}{=} \mathsf{snd} \circ red_{D\times E}(d,e) \neq e$$

$$e \searrow d \stackrel{\text{def}}{=} \mathsf{fst} \circ red_{D\times E}(d,e) \neq d$$

A domain D reduces another domain E iff some member in D reduces some member in E. In other words, $D \searrow E \stackrel{\text{def}}{=} \exists d \in D.\exists e \in E.(d \searrow e)$. $E \searrow D$ is defined similarly.

The reduced product approach improves component analyses because there is a reduction relation between their abstract domains. One component analysis improves another only if the abstract domain of the former reduces the abstract domain of the latter. Since $red_{D\times E} = \langle \rho_D, \rho_E \rangle \circ \sqcap_C$, we have

$$d \searrow e \leftrightarrow \rho_E(d \sqcap_C e) \neq e$$

$$D \not\searrow E \leftrightarrow \forall d \in D.\forall e \in E.\rho_E(d \sqcap_C e) = e$$

where $D \not\searrow E \stackrel{def}{=} \neg(D \searrow E)$. Since there is a one-to-one correspondence between abstract domains and upper closure operators, the notion of reduction naturally extends to upper closure operators.

The following lemma states that, in order to decide if one abstract domain E is reduced by another domain D, it is sufficient to check if the least element $\bot_D$ in D reduces any element e in E.

Lemma 3.1 $D \not\searrow E \leftrightarrow \forall e \in E.(\bot_D \not\searrow e)$ *where* $\bot_D = \rho_D(\bot_C)$.

The relation $\not\searrow$ between abstract domains is monotonic in both of its arguments, and hence its reverse $\searrow$ is antitonic in both of its arguments.

Lemma 3.2 *Let $\rho_D, \rho_{D'}, \rho_E, \rho_{E'} \in U$. If $\rho_D \sqsubseteq_U \rho_{D'}$, $\rho_E \sqsubseteq_U \rho_{E'}$ and $\rho_D(C) \not\searrow \rho_E(C)$ then $\rho_{D'}(C) \not\searrow \rho_{E'}(C)$.*

4 Method

This section presents a method for computing the reduced product of two program analyses in the case that one component analysis D does not improve the other E. We show that the reduced product of D and E can be computed with improved efficiency as follows. The component analysis E is first computed since it is not improved by D. The component analysis D is then computed with the result of E being used to reduce the intermediate results of D in each step.

Theorem 4.1 *Let*

$$\begin{aligned} d' &= \mathsf{lfp}\ \lambda d.\mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d), \mathsf{lfp}\ \mathcal{E}) \\ d'' &= \mathsf{fst}(\mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})) \end{aligned}$$

If $D \searrow E$ then $d' = d''$.

Proof: Since $D \searrow E$, we have $(d'', \mathsf{lfp}\ \mathcal{E}) = \mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})$. Therefore,

$$\begin{aligned}(d'', \mathsf{lfp}\ \mathcal{E}) &= red_{D\times E}(\mathcal{D}(d''), \mathcal{E}(\mathsf{lfp}\ \mathcal{E})) \\ &= red_{D\times E}(\mathcal{D}(d''), \mathsf{lfp}\ \mathcal{E})\end{aligned}$$

Hence, $d'' = \mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d''), \mathsf{lfp}\ \mathcal{E})$ is a fixpoint of $\lambda d.\mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d), \mathsf{lfp}\ \mathcal{E})$. Therefore, $d' \sqsubseteq_D d''$ since d' is the least fixpoint of $\lambda d.\mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d), \mathsf{lfp}\ \mathcal{E})$. Since $red_{D\times E}$ is monotonic, we can show by induction that, for any $i \geq 0$,

$$\begin{aligned}&\mathsf{fst} \circ [red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})]^i(\bot_D, \bot_E) \\ &\quad \sqsubseteq_D \quad [\lambda d.\mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d), \mathsf{lfp}\ \mathcal{E})]^i(\bot_D)\end{aligned}$$

Therefore, $d'' \sqsubseteq_D d'$. We have $d'' = d'$ since both $d'' \sqsubseteq_D d'$ and $d' \sqsubseteq_D d''$. This completes the proof of the lemma. ∎

So, instead of computing directly $\mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})$, we first compute $\mathsf{lfp}\ \mathcal{E}$. Since $D \searrow E$, $\mathsf{snd} \circ \mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E}) = \mathsf{lfp}\ \mathcal{E}$. $d' = \mathsf{lfp}\ \lambda d.\mathsf{fst} \circ red_{D\times E}(\mathcal{D}(d), \mathsf{lfp}\ \mathcal{E})$ is then computed. Theorem 4.1 guarantees $(d', \mathsf{lfp}\ \mathcal{E}) = \mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})$. The method for computing the reduced products of analyses is incremental in that the E component is first computed and then the D component is computed with the result of the E component being used to reduce the D component. The non-incremental method which directly computes $\mathsf{lfp}\ red_{D\times E} \circ (\mathcal{D} \times \mathcal{E})$ uses intermediate results of the E component to reduce the intermediate results of the D component. This is necessary when the two component domains reduce each other. However, this slows down the convergence to the least fixpoint of the D component if the D does not reduce the E component, because reduction at each intermediate step is over zealous. The incremental method gains efficiency by using the final result $\mathsf{lfp}\ \mathcal{E}$ of the E component to reduce the intermediate results of the D component. As shown in the proof of theorem 4.1, each intermediate results of the D component in the incremental method is no less than the corresponding intermediate results in the non-incremental method. This means that incremental method takes less steps to reach the least fixpoint of the D component.

The proof of theorem 4.1 indicates that the incremental method computes a safe approximation of the reduced product of analyses even if component analyses improve each other. The result in this case will be less precise than the non-incremental method in [9, 5]. Therefore, it can be safely used to trade precision for efficiency and design effort in the case that component analyses improve each other. When no component analysis improves the other, the reduced product of analyses degenerates to their direct product. In this case, the incremental method will compute the same result as the non-incremental one possibly with a little loss in efficiency because it does book-keepings for component analyses separately.

We now illustrate the incremental method with the computation of the reduced product of groundness and pair sharing analyses of logic programs.

5 Groundness/Pair Sharing

In logic programming, a program state is a substitution that is an almost everywhere identity function mapping a variable into a term. Furthermore, substitutions constituting program states are idempotent. Let Sub be the set of idempotent substitutions. The concrete domain $(\wp(\mathsf{Sub}), \subseteq)$ is the set of subsets of Sub ordered by set inclusion. Let Vars be the set of all variables, PVar be the set of program variables of interest and $\mathbf{V}_O$ be the set of variables occurring in a syntactic object O.

5.1 Pair Sharing

Søndergaard [18] proposes an abstract domain for occur check analysis which corresponds to the reduced product of an abstract domain for pair sharing and an abstract domain for groundness. The abstract domain for pair sharing captures sharing (independence) and linearity information and is the set of symmetric relations on PVar ordered by set inclusion.

Two distinct variables x, y share under a substitution θ if $\mathbf{V}_{\theta(x)} \cap \mathbf{V}_{\theta(y)} \neq \emptyset$. Otherwise, they are independent. Let $\mathsf{indep} : \mathsf{Sub} \mapsto \mathsf{PVar}^2 \setminus \{(x,x) \mid x \in \mathsf{PVar}\} \mapsto \mathsf{Bool}$ be $\mathsf{indep}\ \theta\ x\ y \stackrel{def}{=} \mathbf{V}_{\theta x} \cap \mathbf{V}_{\theta y} = \emptyset$. A variable x is linear under a substitution θ if no variable occurs more than once in $\theta(x)$. Define $\mathsf{linear} : \mathsf{Sub} \mapsto \mathsf{PVar} \mapsto \mathsf{Bool}$ as $\mathsf{linear}\ \theta\ x \stackrel{def}{=}$ "x is linear under θ".

The possible sharing among variables in PVar and possible non-linearity of variables in PVar under a substitution θ are represented as a symmetric relation $\mathcal{P}$ on PVar. A substitution θ satisfies $\mathcal{P}$ if, for any $x, y \in \mathsf{PVar}$,

$$\begin{aligned}&(x \neq y \rightarrow ((x,y) \notin \mathcal{P} \rightarrow \mathsf{indep}\ \theta\ x\ y)) \\ \wedge\quad &((x,x) \notin \mathcal{P} \rightarrow \mathsf{linear}\ \theta\ x)\end{aligned}$$

Let PSH be the set of symmetric relations on PVar.

$< PSH, \subseteq >$ is a complete lattice. The Galois insertion from $< PSH, \subseteq >$ to $(\wp(Sub), \subseteq)$ is as follows.

$$\begin{array}{l} \alpha_{PSH}(\Theta) \stackrel{def}{=} \{(x,x) \mid \exists\theta \in \Theta.\neg \mathsf{linear}\ \theta\ x\} \\ \quad \cup \{(x,y) \mid (x \neq y) \wedge \exists\theta \in \Theta.\neg\mathsf{indep}\ \theta\ x\ y\} \\ \gamma_{PSH}(\mathcal{P}) \stackrel{def}{=} \\ \left\{ \theta \left| \begin{array}{l} \forall x,y \in \mathsf{PVar}. \\ (x \neq y \rightarrow ((x,y) \notin \mathcal{P} \rightarrow \mathsf{indep}\ \theta\ x\ y)) \\ \wedge ((x,x) \notin \mathcal{P} \rightarrow \mathsf{linear}\ \theta\ x) \end{array} \right. \right\} \end{array}$$

5.2 Groundness

In groundness analysis, we are interested in knowing which variables in PVar will be definitely instantiated to ground terms. Therefore, a set of substitutions is approximated naturally by a subset of PVar. The abstract domain for groundness analysis is thus $GRD \stackrel{def}{=} \wp(\mathsf{PVar})$. The partial order on GRD induced by $\subseteq$ on $\wp(\mathsf{Sub})$ is $\supseteq$. $(GRD, \supseteq)$ is a complete lattice. Let $\mathsf{ground} : \mathsf{Sub} \mapsto \mathsf{PVar} \mapsto \mathsf{Bool}$ be $\mathsf{ground}\ \theta\ x \stackrel{def}{=} (\mathbf{V}_{\theta x} = \emptyset)$. The Galois insertion from $(GRD, \supseteq)$ to $(\wp(\mathsf{Sub}), \subseteq)$ is as follows.

$$\begin{array}{lcl} \alpha_{GRD}(\Theta) & \stackrel{def}{=} & \{x \in \mathsf{PVar} \mid \forall\theta \in \Theta.\mathsf{ground}\ \theta\ x\} \\ \gamma_{GRD}(g) & \stackrel{def}{=} & \{\theta \mid \forall x \in g.\mathsf{ground}\ \theta\ x\} \end{array}$$

5.3 Computing Reduced Product of Groundness and Pair Sharing

It is well known that groundness analysis can be used to improve other analyses such as pair sharing analysis. If $\mathcal{P} \in PSH$ and $\mathcal{P} \neq \emptyset$ then it can be reduced by some groundness description. A stronger statement is as follows.

Theorem 5.1 *For all $\mathcal{P}', \mathcal{P} \in PSH$, if $\mathcal{P} \neq \emptyset$ and $\mathcal{P}' \subset \mathcal{P}$ then there is a $g \in GRD$ such that*

$$(\gamma_{PSH}(\mathcal{P}) \cap \gamma_{GRD}(g) = \gamma_{PSH}(\mathcal{P}') \cap \gamma_{GRD}(g))$$

The following theorem states that pair sharing analysis doesn't reduce groundness analysis.

Theorem 5.2 *For all $\mathcal{P} \in PSH$ and $g_1, g_2 \in GRD$, if $g_1 \neq g_2$ then*

$$\gamma_{PSH}(\mathcal{P}) \cap \gamma_{GRD}(g_1) \neq \gamma_{PSH}(\mathcal{P}) \cap \gamma_{GRD}(g_2)$$

The method in section 4 is applicable to computing the reduced product of groundness and pair sharing analyses because the above two theorems ensures $GRD \searrow PSH$ and $PSH \not\searrow GRD$.

We have implemented both the non-incremental and the incremental methods in SWI Prolog and using the fixpoint abstract semantics in [15]. The fixpoint abstract semantics extends [16] with negation as failure and built-in predicates. The abstract interpretation of built-in predicates is done as in [4]. Exactly the same implementations of the abstract operators for the groundness and the sharing analyses and the reduction function for these two analyses are used.

Some experiment results are listed in table 1. Each row compares the performance of the incremental method with that of the non-incremental method on a given program, a given goal and a given input abstract substitution. The first entry is the name of the program, the second the goal, and the third the input abstract substitution. Each of the fourth to seventh entries lists a ratio of a measurement of the incremental method by the same measurement of the non-incremental method. The fourth entry measures the time, the fifth the number of calls to the abstract unification operator for pair shaing, and the sixth the number of calls to the abstract least upper bound operator for pair sharing, and the seventh the number of calls to the reduction function. The table doesn't measure the number of calls to the abstract unification operator for groundness or the number of calls to the abstract least upper bound operator for groundness as the incremental and the non-incremental methods do not differ significantly on these two measurements.

We make the following observations of the experiment results. Firstly, the incremental method behaves better than the non-incremental method on every input consisting of a program, a goal and an abstract substitution. Secondly, incremental method is more than 60 percent faster than the non-incremental method for some inputs while it is only slightly better than the non-incremental method for some other inputs. In average, the non-incremental method is 21 percent faster, makes 22 percent less calls to the abstract unification operator for pair sharing, 22 percent less calls to the abstract least upper bound operator for pair sharing, and 55 percent less calls to the reduction function for the groundness and the pair sharing analyses.

6 Conclusion

A reduction relation between abstract domains is defined and is used to identify situations where the reduced product of two analyses can be computed more efficiently. An incremental method for computing the reduced product is presented. A substantial speed-up in computing the reduced product of groundness and pair sharing analyses is reported.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 2001. To appear.

[2] R. Bagnara, E. Zaffanella, and P. M. Hill. Enhancing Sharing for precision. In M. C. Meo and M. Vilares Ferro, editors, *Proceedings of the "APPIA-GULP-PRODE'99 Joint Conference on Declarative Programming"*, pages 213–227, L'Aquila, Italy, 1999.

[3] R. Bagnara, E. Zaffanella, and P. M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 103–114, Montreal, Canada, 2000. ACM Press.

[4] M. Bruynooghe. A practical framework for the abstract interpretation of logic progams. *Journal of Logic Programming*, 10(2):91–124, 1991.

[5] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 194–205. The ACM Press, 1993.

[6] M. Codish and E. Shapiro. Compiling OR-parallelism into AND-parallelism. *Lecture Notes in Computer Science*, 225:283–297, 1986.

[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*, pages 52–61, New Haven, Connecticut, USA, 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on Principles of programming languages*, pages 238–252. The ACM Press, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the sixth annual ACM symposium on Principles of programming languages*, pages 269–282. The ACM Press, 1979.

[10] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[11] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, USA, 1989.

[12] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1–4):291–314, 1992.

[13] Dean Jacobs and Anno Langen. Compilation of logic programs for restricted and-parallelism. *Lecture Notes in Computer Science*, pages 300:284–297, 1988.

[14] N.D. Jones and H. Søndergarrd. A semantics-based framework for abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, 1987.

[15] L. Lu. Abstract interpretation, bug detection and bug diagnosis in normal logic programs. PhD thesis, University of Birmingham, 1994.

[16] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82. Springer-Verlag, 1988.

[17] Enrico Pontelli and Gopal Gupta. Implementation mechanisms for dependent and-parallelism. In Lee Naish, editor, *Logic programming: Proceedings of the fourteenth International Conference on Logic Programming*, pages 123–137. The MIT Press, July 1997.

[18] H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. *Lecture Notes in Computer Science*, 213:324–338, 1986.

Table 1: Performance Result on Groundness and Pair-Sharing

Program	Input	Goal	Time	s_unify	s_lub	gs_red
Buggy Quick Sort	([Li], [])	qs(Li, Lo)	0.9	0.87	0.87	0.51
List Difference	([Y, Z], [])	diff(X, Y, Z)	0.92	0.87	0.88	0.53
Interchange Sort	([Xs], [])	sort(Xs, Ys)	0.73	0.54	0.54	0.32
QuickSort with D-List	([Xs], [])	quicksort(Xs, Ys)	0.86	0.82	0.83	0.48
append	([L3], [])	append(L1, L2, L3)	0.39	0.25	0.25	0.14
rotate	([X], [])	rotate(X, Y)	0.63	0.57	0.56	0.33
eager	([], [])	go	0.85	0.72	0.72	0.49
press	([], [])	test_press(X, Y)	0.67	0.53	0.53	0.28
read	([], [])	read(X, Y)	0.91	0.86	0.86	0.44
peep	([Pi], [])	comppeepopt(Pi, OP, Pr)	0.94	0.72	0.72	0.39
gabriel	([], [])	main(V1, V2)	0.84	0.72	0.72	0.39
simple_analyzer	([], [])	main	0.77	0.68	0.69	0.47
browse	([], [])	q	0.91	0.71	0.74	0.44
tsp	([N, V], [])	tsp(N, V, M, S, C)	0.91	0.82	0.82	0.46
neural	([X], [])	test(X, N)	0.7	0.78	0.8	0.43
Heapify Binary Trees	([H], [])	heapify(T, H)	0.71	0.61	0.61	0.34
rev	([Ys], [(Xs, Xs)])	rev(Xs, Ys)	0.81	0.83	0.83	0.49
Exponentiation	([N, Y], [])	exp(N, X, Y)	0.79	0.61	0.61	0.38
Exponentiation	([N, X], [])	exp(N, X, Y)	0.98	1	1	0.58
meta	([G], [])	interpret(G)	0.95	0.94	0.94	0.5
reducer	([], [])	main	0.94	0.93	0.94	0.53
serialize	([], [])	go(S)	0.91	0.95	0.93	0.51
naughts-and-crosses	([], [(R, R)])	play(R)	0.92	0.94	0.92	0.51
semi	([], [])	go(N, T)	0.92	0.96	0.91	0.5
ann	([G], [])	go(G)	0.94	0.97	0.97	0.82
asm	([P, O], [])	asm_PIL(P, O)	0.93	0.97	0.97	0.53
Factorial	([F], [])	factorial(N, F)	0.91	0.91	0.91	0.54
AVERAGE	-	-	0.79	0.78	0.78	0.45