A Survey on Product Operators in Abstract Interpretation

Agostino Cortesi
Ca' Foscari University
Venice, Italy
`cortesi@dsi.unive.it`

Giulia Costantini
Ca' Foscari University
Venice, Italy
`costantini@dsi.unive.it`

Pietro Ferrara
ETH
Zurich, Switzerland
`pietro.ferrara@inf.ethz.ch`

The aim of this paper is to provide a general overview of the product operators introduced in the literature as a tool to enhance the analysis accuracy in the Abstract Interpretation framework. In particular we focus on the Cartesian and reduced products, as well as on the reduced cardinal power, an under-used technique whose features deserve to be stressed for their potential impact in practical applications.

1   Introduction

Abstract interpretation [6] has been widely applied as a general technique for the sound approximation of the semantics of computer programs. In particular, abstract domains (to represent data) and semantics (to represent data operations) approximate the concrete computation. When analyzing a program and trying to prove some property on it, the quality of the result is determined by the abstract domain choice. There is always a trade-off between accuracy and efficiency of the analysis. During the years, various abstract domains have been developed. An interesting feature of the abstract interpretation theory is the possibility to combine different domains in the same analysis. In fact, the abstract interpretation framework offers some standard ways to compose abstract domains, ensuring the preservation of the theoretical properties needed to guarantee the soundness of the analysis. These compositional methods are called *domain refinements*. A systematic treatment of abstract domain refinements has been given in [12, 14], where a generic refinement is defined to be a lower closure operator on the lattice of abstract interpretations of a given concrete domain. These kinds of operators on abstract domains provide high-level facilities to tune a program analysis in terms of accuracy and cost. Two of the most well-known domain refinements are the *disjunctive completion* [6, 9, 13, 15, 18] and the *reduced product* [6], but they are not the only ones. The reduced product can be seen as the most precise refinement of the simple Cartesian product. Moreover, the reduced cardinal power is introduced by [6]. While the other domain refinements have been, since their introduction, widely used and explored, the reduced cardinal power has seen definitely less further developments since 1979, with the exception of [16]. To verify our assertion, we looked for the number of scientific citations (in the abstract interpretation context) to some domain refinements in Google Scholar. We depicted the results of this search in Figure 1. In particular, we focused on the Cartesian product, the reduced product and the reduced cardinal power. Throughout the years, the number of citations increases in all three cases, but the absolute numbers are very different: just consider that the total citations of "Cartesian product" are 964, while the ones to "reduced cardinal power" are only 38.

However, we think that the reduced cardinal power refinement has a potential which has not been completely exploited yet. Consider, for example, that such refinement is used by ASTRÉE[10], the well-known static program analyzer that proves the absence of run-time errors in safety-critical embedded
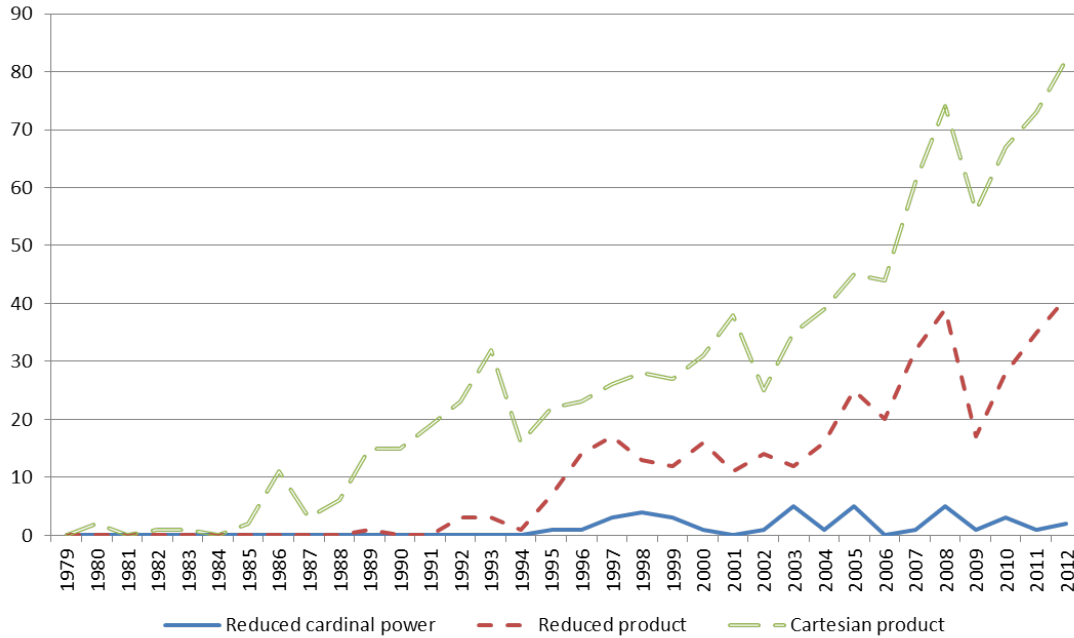
Figure 1: Number of annual citations since 1979, for the searches "reduced cardinal power", "reduced product" and "cartesian product" (in the abstract interpretation field)

applications written or automatically generated in C. This paper aims at giving a survey of the different product operators introduced in the literature by providing a uniform terminology, an analysis of their complexity, and of the implementation effort they require.

2 Formal definitions, complexity, and implementation

In this Section, we introduce the three main ways of combining various abstract domains in the abstract interpretation theory (namely, the Cartesian product, the reduced product, and the reduced cardinal power). For the sake of simplicity, we will focus on the combination of *two* abstract domains. Therefore, we suppose that two abstract domains $\overline{A}$ and $\overline{B}$ are given, and that they are equipped with lattice operators: $\langle \overline{A}, \leq_{\overline{A}}, \sqcup_{\overline{A}}, \sqcap_{\overline{A}} \rangle$ and $\langle \overline{B}, \leq_{\overline{B}}, \sqcup_{\overline{B}}, \sqcap_{\overline{B}} \rangle$.

In addition, let C be the concrete domain. We suppose that this domain is equipped with lattice operators as well: $\langle C, \leq_C, \sqcup_C, \sqcap_C \rangle$. We suppose that both $\overline{A}$ and $\overline{B}$ are sound abstractions of C, that is, they form a Galois connection: $\langle C, \leq_C \rangle \xleftrightarrow[\alpha_{\overline{A}}]{\gamma_{\overline{A}}} \langle \overline{A}, \leq_{\overline{A}} \rangle$ and $\langle C, \leq_C \rangle \xleftrightarrow[\alpha_{\overline{B}}]{\gamma_{\overline{B}}} \langle \overline{B}, \leq_{\overline{B}} \rangle$, where $\alpha_{\overline{A}}, \gamma_{\overline{A}}$ and $\alpha_{\overline{B}}, \gamma_{\overline{B}}$ are the abstraction and concretization functions of $\overline{A}$ and $\overline{B}$, respectively. [1]

Finally, abstract domains provide abstract semantic transformers. Formally, we suppose that $\overline{A}$ provides $\mathbb{S}_{\overline{A}} : \overline{A} \to \overline{A}$, and $\overline{B}$ provides $\mathbb{S}_{\overline{B}} : \overline{B} \to \overline{B}$. These are sound approximation of the concrete semantics $\mathbb{S}_C : C \to C$. Formally, $\forall \overline{a} \in \overline{A} : \mathbb{S}_C [\![\gamma_{\overline{A}}(\overline{a})]\!] \leq_C \gamma_{\overline{A}}(\mathbb{S}_{\overline{A}}[\![\overline{a}]\!])$ and $\forall \overline{b} \in \overline{B} : \mathbb{S}_C [\![\gamma_{\overline{B}}(\overline{b})]\!] \leq_C \gamma_{\overline{B}}(\mathbb{S}_{\overline{B}}[\![\overline{b}]\!])$.

---

[1] There exist other approaches which can be used as well (e.g., when the best abstraction function does not exist [8]). However, the Galois connection-based approach is definitely the most commonly used [7].

2.1 Cartesian product

The elements of this domain are elements in the Cartesian product of the two domains, and the operators are defined as the component-wise application of the operators of the two domains.

Formally, let $\mathbb{C} = \overline{A} \times \overline{B}$ be the Cartesian product. The partial order is defined as the conjunction of the partial orders of the two domains $((\overline{a}_1, \overline{b}_1) \leq_{\mathbb{C}} (\overline{a}_2, \overline{b}_2) \Leftrightarrow \overline{a}_1 \leq_{\overline{A}} \overline{a}_2 \wedge \overline{b}_1 \leq_{\overline{B}} \overline{b}_2)$. Similarly, the least upper bound and the greatest lower bound operators are defined as the component-wise application of the operators of the two domains $((\overline{a}_1, \overline{b}_1) \sqcup_{\mathbb{C}} (\overline{a}_2, \overline{b}_2) = (\overline{a}_1 \sqcup_{\overline{A}} \overline{a}_2, \overline{b}_1 \sqcup_{\overline{B}} \overline{b}_2)$ and $(\overline{a}_1, \overline{b}_1) \sqcap_{\mathbb{C}} (\overline{a}_2, \overline{b}_2) = (\overline{a}_1 \sqcap_{\overline{A}} \overline{a}_2, \overline{b}_1 \sqcap_{\overline{B}} \overline{b}_2)$, respectively). This way, we obtain that the Cartesian product $\langle \mathbb{C}, \leq_{\mathbb{C}}, \sqcup_{\mathbb{C}}, \sqcap_{\mathbb{C}} \rangle$ forms a lattice. The pairwise approach to combine operators holds also in the case of widening. In fact, given the widening operators ∇_A and ∇_B on the domains A and B, respectively, the operator $\nabla_{A \times B}((a_1, b_1), (a_2, b_2)) = (a_1 \nabla_A a_2, b_1 \nabla_B b_2)$ is a widening operator on $\mathbb{C}$ [3].

In addition, the abstraction function $\alpha_{\mathbb{C}}$ consists in the component-wise application of the abstraction functions of the two domains $(\alpha_{\mathbb{C}}(c) = (\alpha_{\overline{A}}(c), \alpha_{\overline{B}}(c)))$, while the concretization function $\gamma_{\mathbb{C}}$ consists in the intersection of the results obtained by the concretization functions of the two domains on the corresponding component $(\gamma_{\mathbb{C}}(\overline{a}, \overline{b}) = \gamma_{\overline{A}}(\overline{a}) \sqcap_C \gamma_{\overline{B}}(\overline{b}))$. Then, the Cartesian product forms a Galois connection with the concrete domain (formally, $\langle C, \leq_C \rangle \xleftarrow[\alpha_{\mathbb{C}}]{\gamma_{\mathbb{C}}} \langle \mathbb{C}, \leq_{\mathbb{C}} \rangle$).

Finally, also the semantic operator $\mathbb{S}_{\mathbb{C}} : \mathbb{C} \to \mathbb{C}$ is defined as the component-wise application of the abstract semantics of the two domains (formally, $\mathbb{S}_{\mathbb{C}}[\![(\overline{a}, \overline{b})]\!] = (\mathbb{S}_{\overline{A}}[\![\overline{a}]\!], \mathbb{S}_{\overline{B}}[\![\overline{b}]\!]))$. This way, the semantics of the Cartesian product is a sound over-approximation of the concrete semantics $(\forall (\overline{a}, \overline{b}) \in \mathbb{C} : \mathbb{S}_C[\![\gamma_{\mathbb{C}}(\overline{a}, \overline{b})]\!] \leq_C \gamma_{\mathbb{C}}(\mathbb{S}_{\mathbb{C}}[\![\overline{a}, \overline{b}]\!]))$.

As pointed out by Patrick Cousot [4], "the Cartesian product discovers in one shot the information found separately by the component analyses", but "we do not learn more by performing all analyses simultaneously than by performing them one after another and finally taking their conjunctions".

In addition, the Cartesian product may contain several abstract elements that represent the same information. For instance, consider the Cartesian product of the Interval and the Parity domains, and in particular the elements $([2..4], O)$, $([2..3], O)$, $([3..4], O)$, and $([3..3], O)$, where O represents the odd element of the Parity domain. All these elements concretize to the singleton $\{3\}$, but some of them are not minimal[2].

2.1.1 Complexity

When applying lattice or semantic operators, the complexity of the operator defined on $\mathbb{C}$ is exactly the sum of the complexity of the corresponding operators on $\overline{A}$ and $\overline{B}$. Instead, the height of the lattice of $\mathbb{C}$ (that is important to estimate the complexity of computing a fixpoint using this domain) is the multiplication of the heights of $\overline{A}$ and $\overline{B}$.

2.1.2 Implementation

Given the implementations of $\overline{A}$ and $\overline{B}$, the implementation of $\mathbb{C}$ is completely straightforward, and it could be used to combine any existing abstract domain in a completely generic way. In fact, the implementation only requires the existence of the operators, and there is no need to develop anything specific on such domains.

---

[2]An abstract element $\overline{a}$ is minimal w.r.t. a property $c \in C$ if and only if (i) $\gamma(\overline{a}) \geq_C c$ and (ii) $\nexists \overline{a}' : \gamma(\overline{a}') \geq_C c \wedge \overline{a}' < \overline{a}$.

2.2   Reduced product

Even if the Cartesian product is a quite effective way to cheaply combine two domains in terms of both formalization and implementation, it is clear that one may want to let the information flow among the two domains to mutually refine them. Already in one of the foundative papers of abstract interpretation [6], Patrick and Radhia Cousot introduced the reduced product exactly with the purpose of refining the information tracked by $\overline{A}$ and $\overline{B}$. In particular, when we have an abstract state that is non-minimal, we can take the smallest element which represents the same information by *reducing* it. A reduction improves the precision of the abstract representation with respect to the order in the Cartesian product without affecting its concrete meaning. Intuitively, a reduction exploits the information tracked by one of the two domains involved in the product to refine the information tracked by the other one (and viceversa). Let (a,b) be an element of a reduced product (where a and b belong respectively to the two domains combined in the product: $a \in \overline{A}$, $b \in \overline{B}$). Let c_1 be the set of concrete values associated to a and c_2 be the set of concrete values associated to b. Then, the element (a,b) represents the set of concrete elements $c_1 \cap c_2$. The reduction tries to find the smallest element (a',b') such that the concretizations of a' and b' are subsets of those of a,b (respectively), but their intersection remains the same as the original one $(c_1 \cap c_2)$.

The lattice and semantic structures of the reduced product are exactly the same as those of the Cartesian product. In addition, a reduction operator aimed at refining the information tracked by the two domains is introduced, and it is used after each lattice or semantic operator application. Formally, the reduction operator $\rho : \mathbb{C} \to \mathbb{C}$ is defined by $\rho(\overline{c}) = \bigsqcap_{\mathbb{C}} \{\overline{c}' \in \mathbb{C} : \gamma_{\mathbb{C}}(\overline{c}) \leq_{\mathbb{C}} \gamma_{\mathbb{C}}(\overline{c}')\}$. Nevertheless, such definition is not computable in general, and often one wants to have a relaxed version of this operator that is not expensive to compute. In general, a reduction operator has to satisfy the following two properties: (i) $\rho(\overline{c}) \leq_{\mathbb{C}} \overline{c}$ (the result of its application is a more precise abstract element); (ii) $\gamma(\rho(\overline{c})) = \gamma(\overline{c})$ (an abstract element and its reduction represent the same property).

Consider again the example of the product of the Interval and Parity domains. A simple reduction operator may increase by one the lower bound (or decrease by one the upper bound) of the interval if the bound does not respect the information tracked by the Parity domain (e.g., it is odd while the parity tracks that the value is even). This way, the reduction of $([2..4], \mathsf{O})$, $([2..3], \mathsf{O})$, and $([3..4], \mathsf{O})$ yields in all cases the abstract value $([3..3], \mathsf{O})$. Note that the reduction operator does not always obtain the minimal information. For instance, if we reduce $([1..1], \mathsf{E})$ (where E represents the even element of the Parity domain), we would obtain $(\perp_I, \mathsf{E})$ (where $\perp_I$ is the bottom element of the Intervals domain), that could be further reduced to $(\perp_I, \perp_P)$ (where $\perp_P$ is the bottom element of the Parity domain). Therefore, the reduction operator usually requires to compute a fixpoint [4]. As two other examples, consider the reduced product of Interval and Congruences domains. Firstly, the reduction of the abstract value $([2..2], 3)$ produces the abstract value $(\perp, 3)$ which is not a minimal element. We need to iterate the reduction to obtain $(\perp, \perp)$. Secondly, the reduction of $([4..5], 2)$ produces the abstract value $([4..4], 2)$ which can be reduced again to $([4..4], 4)$.

Observe that the widening operator on the reduced product cannot be derived "for free" as refinement on the the widening operators of the components. As proved in [3], this is true only under the (quite strict) condition that $\forall a_1, a_2 \in A, \forall b_1, b_2 \in B, (a_1 \nabla_A a_2, b_1 \nabla_B b_2) \in \rho(A \times B)$, where $\rho(A \times B)$ represents the elements of the reduced product. This property does not often hold in practice. A far simpler (and naive) solution consists in applying the widening component-wise and refraining from reducing the result before feeding it back as left argument of the next iteration's widening. This subsumes the above condition (as the reduction becomes idempotent on the iterates), but it also allows converging when the condition does not hold.

2.2.1 Complexity

In addition to the complexity of the Cartesian product, the reduced product requires to compute the reduction operator. Therefore, the complexity of an operator of the reduced product is the sum of the complexity of the operators defined on $\overline{A}$ and $\overline{B}$ and of the reduction operator. Since this operator may require computing a fixpoint, the final cost of a generic operator could be rather expensive. Therefore, usually it is more convenient to define a reduction operator that refines only partially the information tracked by the two domains [19].

2.2.2 Implementation

The implementation of the reduction operator has to be specific for the domains we are refining. Therefore, while the Cartesian product was completely generic and automatic, the reduced product requires one to define and implement how two domains let the information flow among them. This means that each time we want to combine two domains in a reduced product we have to implement such operator. On the other hand, all the other lattice and semantic operators are defined exactly as in the Cartesian product, except that they have to call the reduction operator at the end, but this can be implemented generically w.r.t. the combined domains.

2.2.3 Granger product

Granger [17] proposed an elegant solution to compute an approximation of the reduction operator. Granger based his new product on the definition of two operators $\rho_1 : \mathbb{C} \to \overline{A}$ and $\rho_2 : \mathbb{C} \to \overline{B}$. The idea is that each operator refines one of the two domains involved in the product. The final reduction is obtained by iteratively applying ρ_1 and ρ_2. In order to have a sound reduction operator, ρ_1 and ρ_2 have to satisfy the following conditions:

- $\rho_1(\overline{a},\overline{b}) \leq_{\overline{A}} \overline{a} \wedge \gamma_{\mathbb{C}}(\rho_1(\overline{a},\overline{b}),\overline{b}) = \gamma_{\mathbb{C}}(\overline{a},\overline{b})$

- $\rho_2(\overline{a},\overline{b}) \leq_{\overline{B}} \overline{b} \wedge \gamma_{\mathbb{C}}(\overline{a},\rho_2(\overline{a},\overline{b})) = \gamma_{\mathbb{C}}(\overline{a},\overline{b})$

The intuition behind Granger's product is that dealing with only one flow of information at a time is simpler. Each of the two operators ρ_1, ρ_2 tries to descend in one of the lattices: given the abstract element made by the pair $(\overline{a},\overline{b})$, the ρ_1 operator uses the information from $\overline{b}$ to go down the lattice of $\overline{A}$, while the ρ_2 operator uses the information from $\overline{a}$ to go down the lattice of $\overline{B}$. After each application of ρ_1 or ρ_2 we get a smaller element. The descent is iteratively repeated until the operators cannot recover any more precision: the reduction operator $\rho(\overline{a},\overline{b})$ is then defined as the fixpoint of the decreasing iteration sequence obtained by applying ρ_1 and ρ_2. This is defined by the sequence $(\overline{a}^n,\overline{b}^n)_{n\in\mathbb{N}}$ as follows:

$$(\overline{a}^0,\overline{b}^0) = (\overline{a},\overline{b})$$
$$(\overline{a}^{n+1},\overline{b}^{n+1}) = (\rho_1(\overline{a}^n,\overline{b}^n),\rho_2(\overline{a}^n,\overline{b}^n))$$

The Granger product has exactly the same complexity we discussed for the reduced product. The main practical advantage of the Granger product is that one only needs to define and implement ρ_1 and ρ_2, that is, how the information flows from one domain to the other in one step. Then the reduction operator relying on the fixpoint computation comes for free.

2.2.4   Open product

Cortesi et al. [2] proposed a further refinement of the Cartesian product. Its purpose is to let the domains interact with each other during *and* after operations by making explicit the domains' interaction through (abstract) queries. The open product is orthogonal to Granger's product and the two proposals can be combined, by incorporating Granger's idea of refinement inside the open product. The open product is orthogonal also to other methods such as down-set completion, and tensor product.

2.3   Reduced cardinal power

The reduced cardinal power was introduced by Cousot and Cousot in [6], but the literature concerning it has been relatively poor on both the theoretical and the practical level. The main feature of the cardinal power is that it allows one to track disjunctive information over the abstract values of the analysis. For instance, given the Interval and the Parity domain, one could track information like "when x is odd, y is in $[0..10]$". Some examples of the application of the cardinal power are the example 10.2.0.2 of [6], and examples 3 and 4 of [11]. In addition, a detailed explanation with various examples has been proposed by Giacobazzi and Ranzato [16]. Let us look at the example in [11], where the following slice of code is analyzed (typical of data transfer protocols where even and odd numbered packets contain data of different types):

```
1: n := 10;  i := 0;  A := new int[n];
2: while (i < n) do {
3:   A[i] := 0;
4:   i := i + 1;
5:   A[i] := −16;
6:   i := i + 1;
7: }
```

To analyze it, the authors combine *Parity* (where the lattice is made by the abstract elements $\bot, o, e, \top$) and *Intervals*. The reduced cardinal power of Intervals by Parity tracks abstract properties of the form $(o \rightarrow i_o, e \rightarrow i_e)$, which means that the interval associated to some variable is i_o (resp. i_e) when the parity associated to another variable (which could be the same) is o (resp. e). First of all, the authors show a non-relational analysis of the listing above, where they use:

- the reduced product of Parity and Intervals for simple variables;

- the reduced cardinal power of Parity by Interval for array elements (hence ignoring their relationship to indexes)

For example $(o \rightarrow \bot, e \rightarrow [-16,0])$ means that the indexed array elements must be even with value included between -16 and 0. The result of this analysis is: $i : (e, [10,10])$ (variable i is even and has value 10), $n : (e, [10,10])$ (variable *n* is even and has value 10), and $A : (o \rightarrow \bot, e \rightarrow [-16,0])$, which represents that array elements are abstracted by $(o \rightarrow \bot, e \rightarrow [-16,0])$ (i.e., they are even and with values in $[-16,0]$). The precision of this analysis can be greatly improved by using again the reduced cardinal power of Intervals by Parity, but this time relating the parity of an *index* of the array to the interval of the *elements* of the array at that index. The new result is $A : (o \rightarrow [-16,-16], e \rightarrow [0,0])$, which means that the array elements at odd indexes are equal to -16 while those at even indexes are 0.

The reduced cardinal power has been formalized as follows in [6]. Given two abstract domains $\overline{A}$ and $\overline{B}$, the cardinal power $\mathfrak{P} = \overline{B}^{\overline{A}}$ with base $\overline{B}$ and exponent $\overline{A}$ is the set of all isotone maps $\mathfrak{P} : \overline{A} \rightarrow \overline{B}$.

Roughly, the combination of two abstract domains in $\overline{B}^{\overline{A}}$ means that a state in $\overline{A}$ implies the abstract state of $\overline{B}$ it is in relation with. The partial ordering $\leq_{\mathfrak{p}}$ is defined by $\overline{f} \leq_{\mathfrak{p}} \overline{g} \Leftrightarrow \forall \overline{x} \in \overline{A} : \overline{f}(\overline{x}) \leq_{\overline{B}} \overline{g}(\overline{x})$. Similarly, the least upper bound and greatest lower bound operators are defined as the pointwise application of the operators of $\overline{B}$. This way, $\langle \mathfrak{P}, \leq_{\mathfrak{p}}, \sqcup_{\mathfrak{p}}, \sqcap_{\mathfrak{p}} \rangle$ forms a lattice.

Let f_1, f_2 be two abstract elements in $\mathfrak{P}$. Then, the widening operator $\nabla_{\mathfrak{p}}$ on (f_1, f_2) can be defined as:

$$\forall x \in \overline{A} : \nabla_{\mathfrak{p}}(f_1, f_2)(x) = \nabla_{\overline{B}}(f_1(x), f_2(x))$$

Observe that the operator above can be effectively applied only if $\overline{A}$ is finite.

By defining $\alpha_{\mathfrak{p}}(c) = \lambda \overline{x}.\alpha_{\overline{B}}(c \sqcap_C \gamma_{\overline{A}}(\overline{x}))$ and $\gamma_{\mathfrak{p}}(\overline{p})$ consequently, we have that $\langle C, \leq_C \rangle \xleftrightarrow[\alpha_{\mathfrak{p}}]{\gamma_{\mathfrak{p}}} \langle \mathfrak{P}, \leq_{\mathfrak{p}} \rangle$. We refer the interested reader to [16] (and in particular to Theorem 3.6 and Proposition 3.7, where $\odot$ corresponds to $\sqcap_C$) for more details and formal proofs.

A correctness result was presented in [6] as well (Theorem 10.2.0.1). In this work, the authors focused on a collecting semantics defined by a lattice of assertions which is a Boolean algebra. Afterwards, Cousot and Cousot did not broaden their theoretical definition to a more general setting.

Let us recall the example used in [6] to show the expressiveness of this domain.

```
1: x := 100; b := true;
2: while b do {
3:      x := x − 1;
4:      b := (x > 0);
5: }
```

The exponent of the cardinal power we use to analyze this example is the Boolean domain for variable b, while the base is the Sign domain for variable x tracking values $+, -, 0$ as well as $0+$ (meaning that the values are ≥ 0), $0-$ (meaning that the values are ≤ 0), $\neq 0$ (meaning that the values are different from 0). This way, we track that when variable b has a particular Boolean value, then the sign of variable x has a particular sign. Before entering the while loop, we know that $b = \text{true} \Rightarrow x = +$, while $b = \text{false} \Rightarrow x = \bot$. After the application of the semantics of statement 3, we will have that $b = \text{true} \Rightarrow x = 0+$, and $b = \text{false} \Rightarrow x = \bot$, because the value of b is unchanged (it is certainly true) while the value of x has been decreased by one (so it could become equal to zero or remain greater than zero). After line 4, we obtain that $b = \text{true} \Rightarrow x = +$, and $b = \text{false} \Rightarrow x = 0$, since the new condition $x > 0$ is assigned to b. In fact, b equals to true implies that x must be greater than zero. In addition, we knew that the value of x was ≥ 0. Then, if b is now false, we are sure that x will be equal to zero (but not less than zero). The fixpoint computation over the while loop stabilizes immediately (because, if we enter the loop again, we know that b is true and, as a consequence, x is positive, thus returning to the same conditions of the first iteration), and so we obtain that at the end of the program we have that $b = \text{true} \Rightarrow x = \bot$, and $b = \text{false} \Rightarrow x = 0$, since we have to assume the negation of b to terminate the execution of the while loop.

The cardinal power effectiveness is compromised when $\overline{A}$ is infinite (i.e., intervals in $\mathbb{Z}$), and can become costly when $\overline{A}$ is finite but non-trivial (i.e., intervals of machine integers). For this reason, some restricted forms of cardinal power can be used, where only a finite subset of $\overline{A}$ is represented.

Summarizing, the main difficulties in constructing a reduced cardinal power domain are: (i) the choice of elements in $\overline{A}$ to use; and (ii) the efficient design of abstract operators.

2.3.1 Complexity

Each time a lattice or semantic operator has to be applied to the abstract state, the cardinal power requires to apply it to all the elements of the base. Consider the cardinal power $\overline{B}^{\overline{A}}$. In a state of our domain, we will track a state of $\overline{B}$ for each possible state of $\overline{A}$. Let n be the number of states of $\overline{A}$, a the cost of an operator on $\overline{A}$ and b the cost on $\overline{B}$. Then the overall cost over $\overline{B}^{\overline{A}}$ is $n * (a + b)$, since, for any element in $\overline{A}$ we have to apply the operator both on $\overline{A}$ and $\overline{B}$.

Let h_a and h_b be the height of the lattice of $\overline{A}$ and $\overline{B}$, respectively. Then the height of $\overline{B}^{\overline{A}}$ is $h_b^{h_a}$.

It is then clear that the cardinal power causes a significant increase in the complexity of the analysis w.r.t. the complexity of the two original analyses. If there is already practical evidence that the reduction operator in the reduced product may induce an analysis that is too complex [19], it is even more important to carefully choose the two domains combined in a cardinal power. Nevertheless, particular instances of the cardinal power are already used to analyze industrial software, and in particular in ASTRÉE [1]. ASTRÉE exploits the Boolean relation domain, which applies the cardinal power using the values of some particular Boolean program variables as exponent. In this way, the analysis tracks precise disjunctive information w.r.t. these variables. In addition, ASTRÉE contains trace partitioning [20]. This can be seen as a cardinal power in which the exponent is a set of manually provided tokens on which the analysis tracks disjunctive information. There are various types of tokens the user can provide: particular abstract values of a variable, the begin of an if statement, etc. It is proved that, in practice, if an expert user provides the *right* tokens, the resulting analysis can be quite precise preserving its performances at the same time.

2.3.2 Implementation

The implementation can be rather simple when using a programming language providing functional constructs. In fact, the most part of the cardinal power (namely, elements of the domain, and lattice and semantic operators) can be defined as the functional point-wise application of the operators on the base and the exponent. Instead, the implementation of the cardinal power may be more verbose using an imperative programming language, but we do not expect it represents a significant challenge.

2.3.3 Reduced relative power

Giacobazzi and Ranzato generalized the reduced cardinal power [16]. For this purpose, the authors introduce the operation of *reduced relative power* on abstract domains. As it happened with the cardinal power, the reduced relative power is based on two domains $\overline{A}$ and $\overline{B}$ (respectively, the exponent and the base) and is defined in a general and standard abstract interpretation setting. Its formal definition is $\overline{A} \xrightarrow{\odot} \overline{B}$, where $\odot$ is a generic operator used to combine concrete denotations. It is called "reduced *relative* power" because it is parametric with respect to $\odot$. The operator $\odot$ should be thought of as a kind of combinator of concrete denotations: the glb is a typical example, but another less restrictive combinator could be needed for some non-trivial applications. An example comes from the field of logic program semantics. The reduced relative power can be used to systematically derive new declarative semantics for logic programs by composing the domains of interpretation of some well-known semantics. In this case a concrete domain of sets of program execution traces is endowed with an operator of trace-unfolding that does not behave like a meet-operation (in particular, it is not even commutative). For more details, see Section 7 of [16]. The definition of the reduced relative power is as follows. $\overline{A} \xrightarrow{\odot} \overline{B}$ consists of all the monotone functions from $\overline{A}$ to $\overline{B}$ having the shape $\lambda \overline{x}.\alpha_B(d \odot \gamma_A(\overline{x}))$, where: (i) d ranges over

concrete values, (ii) γ_A is the concretization function of $\overline{A}$ and (iii) α_B is the abstraction function of $\overline{B}$. These monotone functions establish a dependency between the values of $\overline{A}$ and $\overline{B}$, and for this reason are called *dependencies*. Intuitively, a dependency encodes how the abstract domain $\overline{B}$ is able to represent the "reaction" of the concrete value d whenever it is combined via $\odot$ with an object described by $\overline{A}$.

3 Examples

In this Section, we discuss the application of the Cartesian product, the reduced product, and the cardinal power to some examples dealing with arrays. This way, we show the main features and limits of each combination of domains. The two abstract domains we will combine are Intervals [5] and a relational domain that tracks constraints of the form $x < y + c$.

3.1 Cartesian product

As a first example, we consider a quite standard program that initializes to 0 all the elements of a given array.

```
for(i=0; i < arr.length; i++)
        arr[i] = 0;
```

We want to prove that the array accesses are safe, that is, $i \geq 0$ and $i <$ arr.length, in particular when we perform arr[i] = 0.

If we run the analysis using only Intervals, we obtain that $i = [0.. + \infty]$. In fact, this domain cannot infer any information from the loop guard $i <$ arr.length since it does not have any upper bound for arr.length. This result suffices to prove the first part of our property ($i \geq 0$) but not the second part ($i <$ arr.length). If we run the analysis using only a relational domain, we obtain that $i <$ arr.length $+ 0$ when we analyze the statement arr[i] = 0 thanks to the loop guard. In this way, we can prove the second part of the property, but not the first part.

Therefore, the two domains alone cannot prove the property of interest, while the Cartesian product can. In fact, it runs the two analyses in parallel, and at the end it takes from both domains the most precise result they get regarding the property to verify. Combining the two results, the entire property is proved to hold.

3.2 Reduced product

Let us introduce a more complex example. It receives as input an integer variable l, and it creates an array with one element if $l \leq 0$, and of l elements otherwise. Then it initializes the first l elements to zero.

```
if(l<=0)
        arr = new Int[1];
else
        arr = new Int[l];
for(i=0; i < l; i++)
        arr[i] = 0;
```

As before, we want to prove that when we perform arr[i] = 0 we have that $i \geq 0$ and $i <$ arr.length. In particular, the critical property is the second one, since the first one is already proved by Intervals as explained before.

If we analyze this example with the Cartesian product defined before, we obtain that (i) ($\{$arr.length $\mapsto$ $[1..1], l \mapsto [-\infty..0]\}, \emptyset$) in the then branch, and (ii) ($\{$arr.length $\mapsto [1..+\infty], l \mapsto [1..+\infty]\}, \{$arr.length $<$ $l+1, l <$ arr.length $+1\}$) in the else branch. Then, when we compute the upper bound of these two states, we obtain only ($\{$arr.length $\mapsto [1..+\infty], l \mapsto [-\infty..+\infty]\}, \emptyset$). This leads to infer that ($\{$arr.length $\mapsto [1..+\infty], l \mapsto [-\infty..+\infty], i \mapsto [0..+\infty],\}, \{i < l+0\}$) inside the loop, but this cannot prove that i $<$ arr.length.

We now define a specific reduction operator that refines the information tracked by the relational domain with Intervals. In particular, if Intervals track that x $\mapsto [a..b]$, y $\mapsto [c..d]$ and we have that $b \neq +\infty \wedge c \neq -\infty$, then in the relational domain we introduce the constraint x $<$ y $+ k$ where $k = b - c + 1$. Thanks to this reduction operator, we infer that, in the then branch, ($\{$arr.length $\mapsto [1..1], l \mapsto [-\infty..0]\}, l <$ arr.length $+0$). Thank to this reduction, when we join the two abstract states after the if statement we obtain that ($\{$arr.length $\mapsto [1..+\infty], l \mapsto [-\infty..+\infty]\}, \{l <$ arr.length $+1\}$). This leads to infer (when we analyze arr[i] = 0) that ($\{$arr.length $\mapsto [1..+\infty], l \mapsto [-\infty..+\infty], i \mapsto [0..+\infty],\}, \{l <$ arr.length $+1, i < l+0\}$), and the information tracked by the relational domain proves that i $<$ arr.length.

3.3   Reduced cardinal power

We slightly modify the previous example. In particular, we create an array of one element if l ≤ 2, and we initialize all the elements in the array from the third to the $(l-1)$-th element.

```
if(l <= 2)
    arr = new Int[1];
else
    arr = new Int[l];
for(i = 3; i < l; i++)
    arr[i] = 0;
```

As in the previous example, the main challenge is to prove the second part of the property (that is, i $<$ arr.length) when executing arr[i] = 0.

First of all, we show that the reduced product of Intervals and our relational domain is not in position to prove this property. In the then branch, the Interval domain tracks that l $\in [-\infty..2]$ and arr.length $\in [1..1]$. This information yields the strict lower bound relationship l $<$ arr.length $+2$ through the reduction operator we previously introduced. The abstract state associated to the then branch is ($\{l \to [-\infty..2],$ arr.length $\to [1..1]\}, \{l <$ arr.length $+2\}$), while in the else branch we obtain ($\{l \to [3..+\infty],$ arr.length $\to [3..+\infty]\}, \{$arr.length $< l+1, l <$ arr.length $+1\}$). When we compute the join between these two states, we obtain ($\{l \to [-\infty..+\infty],$ arr.length $\to [1..+\infty]\}, \{l <$ arr.length $+2\}$). In fact, the join of the constraints l $<$ arr.length $+2$ and l $<$ arr.length $+1$ results in l $<$ arr.length $+2$. Finally, inside the for loop we know (from the Interval domain and its widening operator) that i $\to [3..+\infty]$. Moreover, the loop guard implies that, when we perform arr[i] = 0, i $<$ l holds. From i $<$ l and l $<$ arr.length $+2$, we obtain that i $<$ arr.length $+1$, that is weaker than the property of interest i $<$ arr.length.

Now consider the reduced cardinal power of Intervals on l as exponent and our relational domain as base. The then branch is associated to the abstract state $[-\infty..2] \Rightarrow \{l <$ arr.length $+2\}$, and the else branch to $[3..+\infty] \Rightarrow \{$arr.length $< l+1, l <$ arr.length $+1\}$. The join between these two states simply creates a new abstract state which contains both informations, that is, $[-\infty..2] \Rightarrow \{l <$ arr.length $+2\}$ and $[3..+\infty] \Rightarrow \{$arr.length $< l+1, l <$ arr.length $+1\}$. When we enter the while loop, we have to consider the two cases separately:

- in the first case, l $= [-\infty..2]$. Then, the loop guard i $<$ l is surely evaluated to false: the loop is not executed, so we do not need to verify the property about array accesses. Therefore, when i $<$ l

holds, we have that $[-\infty..2] \Rightarrow \perp$. This way, when we analyze $\mathsf{arr}[\mathsf{i}] = 0$, we can discard this case;

- in the second case, we have that $\mathsf{l} = [3..+\infty]$ and $\mathsf{i} < \mathsf{l}$. Then, from the abstract state obtained after the if statements and assuming the loop guard, we know that $[3..+\infty] \Rightarrow \{\mathsf{arr.length} < \mathsf{l}+1, \mathsf{l} < \mathsf{arr.length}+1, \mathsf{i} < \mathsf{l}\}$. By combining $\mathsf{l} < \mathsf{arr.length}+1$ and $\mathsf{i} < \mathsf{l}$, we obtain $\mathsf{i} < \mathsf{l} \leq \mathsf{arr.length} \Rightarrow \mathsf{i} < \mathsf{arr.length}$, which is exactly the property we wanted to prove.

4 Conclusion

In this survey, we presented various product operators in the abstract interpretation theory. For each product, we formalized its main components, we discussed its complexity and the implementation efforts required to implement it. We pointed out that, while the complexity of the Cartesian product does not cause any practical problem, the reduced product may affect the performances of the analysis if the reduction operator is too precise. In addition, the cardinal power leads to a lattice whose height is exponential w.r.t. the heights of the combined domains, and therefore this product requires the user to carefully and manually choose how to combine the two domains. Finally we presented some examples that underline the expressiveness and the limit of the different products, showing in which scenarios a product is satisfactory or when we have to choose a more complex product.

Acknowledgments

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2003): *A Static Analyzer for Large Safety-Critical Software*. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, ACM, New York, NY, USA, pp. 196–207, doi:10.1145/781131.781153.

[2] Agostino Cortesi, Baudouin Le Charlier & Pascal Van Hentenryck (2000): *Combinations of abstract domains for logic programming: open product and generic pattern construction*. Science of Computer Programming 38(1-3), pp. 27–71, doi:10.1016/S0167-6423(99)00045-3.

[3] Agostino Cortesi & Matteo Zanioli (2011): *Widening and narrowing operators for abstract interpretation*. Computer Languages, Systems & Structures 37(1), pp. 24–42, doi:10.1016/j.cl.2010.09.001.

[4] P. Cousot: *MIT course 16.399: Abstract Interpretation*. Available at `http://web.mit.edu/16.399/www/`.

[5] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, ACM, New York, NY, USA, pp. 238–252, doi:10.1145/512950.512973.

[6] P. Cousot & R. Cousot (1979): *Systematic design of program analysis frameworks*. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, ACM, New York, NY, USA, pp. 269–282, doi:10.1145/567752.567778.

[7] P. Cousot & R. Cousot (1992): *Abstract Interpretation and Application to Logic Programs*. Journal of Logic Programming 13(2-3), pp. 103–179, doi:10.1016/0743-1066(92)90030-7.

[8] P. Cousot & N. Halbwachs (1978): *Automatic discovery of linear restraints among variables of a program.* In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, ACM, New York, NY, USA, pp. 84–96, doi:10.1145/512760.512770.

[9] Patrick Cousot & Radhia Cousot (1994): *Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), invited paper.* In: *Proceedings of the 1994 International Conference on Computer Languages*, IEEE Computer Society Press, Los Alamitos, California, Toulouse, France, pp. 95–112.

[10] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival (2005): *The ASTREÉ Analyzer.* In: *Proceedings of the 14th European conference on Programming Languages and Systems*, ESOP'05, Springer-Verlag, Berlin, Heidelberg, pp. 21–30, doi:10.1007/978-3-540-31987-0_3.

[11] Patrick Cousot, Radhia Cousot & Francesco Logozzo (2011): *A parametric segmentation functor for fully automatic and scalable array content analysis.* In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, New York, NY, USA, pp. 105–118, doi:10.1145/1926385.1926399.

[12] Gilberto Filé, Roberto Giacobazzi & Francesco Ranzato (1996): *A Unifying View of Abstract Domain Design.* *ACM Computing Surveys (CSUR)* 28(2), pp. 333–336, doi:10.1145/234528.234742.

[13] Gilberto Filé & Francesco Ranzato (1999): *The Powerset Operator on Abstract Interpretations.* Theoretical Computer Science 222(1-2), pp. 77–111, doi:10.1016/S0304-3975(98)00007-3.

[14] Roberto Giacobazzi & Francesco Ranzato (1997): *Refining and Compressing Abstract Domains.* In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, ICALP '97, Springer-Verlag, London, UK, UK, pp. 771–781, doi:10.1007/3-540-63165-8_230.

[15] Roberto Giacobazzi & Francesco Ranzato (1998): *Optimal Domains for Disjunctive Abstract Intepretation.* Science of Computer Programming - Special issue on the 6th European symposium on programming 32(1-3), pp. 177–210, doi:10.1016/S0167-6423(97)00034-8.

[16] Roberto Giacobazzi & Francesco Ranzato (1999): *The Reduced Relative Power Operation on Abstract Domains.* Theoretical Computer Science 216(1-2), pp. 159–211, doi:10.1016/S0304-3975(98)00194-7.

[17] Philippe Granger (1992): *Improving the Results of Static Analyses Programs by Local Decreasing Iteration.* In: *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS, Springer-Verlag, London, UK, UK, pp. 68–79, doi:10.1007/3-540-56287-7_95.

[18] Thomas P. Jensen (1997): *Disjunctive Program Analysis for Algebraic Data Types.* ACM Transactions on Programming Languages and Systems (TOPLAS) 19(5), pp. 751–803, doi:10.1145/265943.265966.

[19] F. Logozzo & M. Fähndrich (2008): *Pentagons: A weakly relational domain for the efficient validation of array accesses.* In: *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, ACM, New York, NY, USA, pp. 184–188, doi:10.1145/1363686.1363736.

[20] L. Mauborgne & X. Rival (2005): *Trace Partitioning in Abstract Interpretation Based Static Analyzers.* In: *Proceedings of the 14th European conference on Programming Languages and Systems*, ESOP'05, Springer-Verlag, Berlin, Heidelberg, pp. 5–20, doi:10.1007/978-3-540-31987-0_2.