A Unifying View on Abstract Domain Design

Gilberto Filé* Roberto Giacobazzi** Francesco Ranzato*

**Dipartimento di Matematica Pura ed Applicata, Università di Padova*
Via Belzoni 7, 35131 Padova, Italy
`{gilberto,franz}@hilbert.math.unipd.it`

***Dipartimento di Informatica, Università di Pisa*
Corso Italia 40, 56125 Pisa, Italy
`giaco@di.unipi.it`

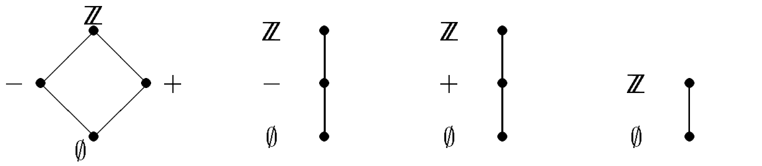
Introduction. The concept of *abstract interpretation* has been introduced by Patrick and Radhia Cousot in [4, 5], in order to formalize static program analyses. Within this framework, our goal is to offer a unifying view on operators for enhancing and simplifying abstract domains. Enhancing and simplifying operators are viewed, respectively, as *domain refinements* and *inverses* of domain refinements. This new unifying viewpoint makes both the understanding and the design of operators on abstract domains much simpler. Enhancing operators increase the expressiveness of an abstract domain: they comprise the Cousot and Cousot *reduced product*, *disjunctive completion* and *reduced cardinal power* ([5]), the Nielson *tensor product* ([9]), the *open product* and the *pattern completion* by Cortesi *et al.* ([3]), and the *functional dependencies* by Giacobazzi and Ranzato ([7]). Simplifying operators are used to reduce complex abstract domains into simpler ones with respect to the efficiency of the corresponding analysis as well as with respect to the proof of their correctness. Simplifying operators comprise the *complementation* of Cortesi *et al.* ([2]) and the Giacobazzi and Ranzato *least disjunctive basis* ([8]).

Domain Refinements and their Inversion. Program analysis is defined in abstract interpretation as non-standard program evaluation. This non-standard semantics is obtained from the standard one by substituting the actual domain of computation (called *concrete*) and its basic operations with, respectively, an *abstract domain* and corresponding *abstract operations*. Both the concrete and the abstract domain are required to be complete lattices, where the ordering relations describe the relative precision of the denotations – the top elements representing no information. An abstraction of a concrete domain C can be viewed as an *upper closure operator* on C, i.e., a function $\rho : C \to C$ which associates with each $x \in C$, an object $\rho(x)$ approximating x (i.e. $x \leq \rho(x)$), and which is both monotonic and idempotent (cf. [5]). Therefore, any domain A which is isomorphic to the image $\rho(C)$ by an upper closure operator ρ on C, can be considered a proper abstraction of C. Once a concrete domain C has been fixed, all its possible abstractions can be compared with each other with respect to their precision of representation. This order corresponds in the most natural way to the usual functional pointwise order between closures (denoted by $\sqsubseteq$), which makes the set $uco(C)$ of all closure operators on C (or equivalently the set of all abstract interpretations of C) a complete lattice.

We introduce the notion of domain refinement as a general scheme in order to formalize enhancing operators on abstract domains. A *domain refinement* is a mapping $\Re : uco(C) \to uco(C)$ such that for any abstract domain A: $\Re(A)$ contains more information than A (i.e., $\Re(A) \sqsubseteq A$), $\Re$ monotonically depends on the information contained in its argument, namely it is monotonic, and a last very reasonable requirement is that $\Re$ upgrades all at once, namely $\Re$ is idempotent. This clearly defines *refinements as lower closure operators* on $uco(C)$.

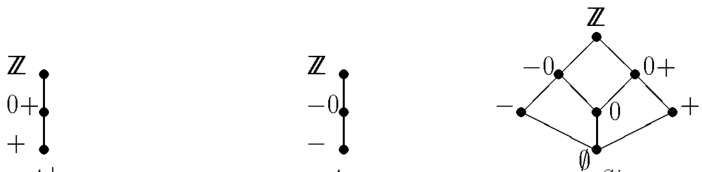
A natural question that arises in this setting is whether it is possible to define the *inverse* of a domain refinement. For a given refinement $\Re$, the inverse $\Re^{-1}$ (if it exists) is a function mapping any domain A into the (unique) most abstract domain $\Re^{-1}(A)$ such that $\Re(\Re^{-1}(A)) = \Re(A)$. Whenever this happens, we say that $\Re^{-1}(A)$ is the *optimal basis* for the domain A and refinement $\Re$. The intuition is that the refined domain $\Re(A)$ can be systematically reconstructed by applying the refinement $\Re$ to the more abstract, and therefore simpler, domain $\Re^{-1}(A)$. This domain $\Re^{-1}(A)$ is in fact the most abstract domain for which this condition holds.

Not all domain refinements are invertible. A simple and meaningful example is provided by the following notion of *completion by complements*: a refinement $\Re_\neg$ which, under certain hypotheses, upgrades a given abstract domain by adding denotations for the lattice-theoretic complements of its elements. If $Sign^{\pm}$, A_1 and A_2 are the domains for sign analysis of integer variables depicted below, with their obvious meanings as upper closures on the subsets of integers $\wp(\mathbb{Z})$, ordered by inclusion, then $\Re_\neg(A_1) = \Re_\neg(A_2) = Sign^{\pm}$, but the common abstraction of A_1 and A_2, denoted $A_1 \sqcup A_2$, is $\{\mathbb{Z}, \emptyset\}$ and $\Re_\neg(\{\mathbb{Z}, \emptyset\}) = \{\mathbb{Z}, \emptyset\} \neq Sign^{\pm}$. Thus, $\Re_\neg^{-1}$ does not exist.



$Sign^{\pm}$ A_1 A_2 $\Re_\neg(A_1 \sqcup A_2)$

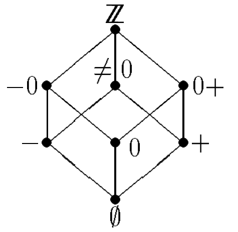
Two Examples. The most widely known domain refinement is *reduced product* ([5]). The reduced product of two abstract domains A and B is the domain obtained by combining the informations in A and B via the concrete *g.l.b.* of their elements. Viewing abstractions as uco's, the reduced product is simply the *g.l.b.* in the complete lattice $uco(C)$, denoted $A \sqcap B$. Reduced product can be viewed as a refinement: given a domain $A \in uco(C)$, $\lambda X.X \sqcap A$ is trivially a lower closure operator on $uco(C)$. If A^+, A^- and *Sign* are the domains depicted below, then *Sign* is the reduced product of A^+ and A^-, i.e. $Sign = A^+ \sqcap A^-$. Note that the denotations 0 and $\emptyset$ are both derived from A^+ and A^- by combining, through set intersection, 0+ and 0−, and, + and −, respectively.



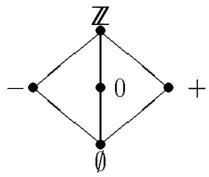
A^+ A^- *Sign*

The practical impact of reduced product has been experimentally shown by Codish *et al.* in [1] for the analysis of logic programs. Reduced product can be inverted. In fact, it corresponds precisely to the operation of *complementation* introduced in [2]. Given any two domains A and B in $uco(C)$, with $A \sqsubseteq B$ (i.e., B more abstract than A), under non restrictive hypotheses, complementation of B in A gives as result the most abstract domain $A \sim B$, whose reduced product with B is exactly A, i.e. $B \sqcap (A \sim B) = A$. It is simple to observe that the existence of the complement implies the existence of the inverse of the reduced product refinement. If we consider the refinement $\lambda X.X \sqcap A$, where $A \in uco(C)$, then the optimal basis for any domain $B \in uco(C)$ is exactly $(B \sqcap A) \sim A$. As a simple example, consider the domains *Sign*, A^+ and A^- introduced above. A^+ and A^- can be "subtracted" from *Sign* by complementation, obtaining $\mathit{Sign} \sim A^+ = A^-$ and $\mathit{Sign} \sim A^- = A^+$. $\langle A^+, A^- \rangle$ is therefore a decomposition for *Sign* (i.e., $\mathit{Sign} = A^+ \sqcap A^-$). Using complementation it is possible to decompose any domain just like we have done for *Sign*. Such decompositions can both improve the space requirement for representing abstract domains (compare $\langle A^+, A^- \rangle$ with *Sign*), and provide a *divide et impera* approach to prove properties of complex abstract domains by exploiting properties of their factors.

A further example of invertible domain refinement is *disjunctive completion* $\Re_\vee$ ([5]). An abstract domain is here upgraded by including denotations for concrete disjunctions of its values. This is achieved by suitable powerset completions. It turns out that $\Re_\vee$ is a refinement. Giacobazzi and Ranzato proved in [8] that, under non restrictive hypotheses on the concrete domain, $\Re_\vee^{-1}$ exists, and for a domain A, $\Re_\vee^{-1}(A)$ depends on the set of the join-irreducible elements of $\Re_\vee(A)$. The disjunctive completion $\Re_\vee(\mathit{Sign})$ (with respect to $\wp(\mathbb{Z})$) and its optimal basis for $\Re_\vee$ are depicted below, the optimal basis being a proper abstraction of *Sign*.



$\Re_\vee(\mathit{Sign})$



$\Re_\vee^{-1}(\mathit{Sign})$

Thus, both *Sign* and its optimal basis lead to the same disjunctive domain for sign-analysis, but the latter is a less expensive domain. It turns out that $0+ = \{0, +\}$, $-0 = \{0, -\}$ and $\neq 0 = \{+, -\}$ can all be systematically reconstructed by disjunctions from elements of the optimal basis.

Research Directions. Many operators for domain design have been defined in the literature. It could be interesting to formalize them as refinements and to find their optimal bases, in particular, for the operators of tensor product, open product, functional dependencies and pattern completion.

Refinements could be a basis for an environment for developing expressive abstract domains. In such an environment, optimal bases represent the least effort that a domain designer has to provide in order to obtain a fully refined domain. Such a situation is considered by Cortesi *et al.* in [3] for the case of pattern completion in logic program analysis. Using their environment, designers need only to contribute about 20% of the whole implementation. A case study of the cost savings when domains are automatically upgraded from their optimal bases is one important direction for future research.

Domain refinements and their inverses lead to an *algebra* of domains for program analysis. A natural question that may arise in this setting is whether it is possible to define abstract domains as solutions of equations on this algebra. This would provide an equational-like presentation of abstract domains, similarly to what is well known in standard domain theory, e.g. for denotational semantics. This algebra may also provide systematic methods to design semantics for programming languages. Cousot and Cousot proved in [6] that it is possible to relate different semantic definitions, at different levels of abstraction, by abstract interpretation. The algebra of basic operators for domain refinement and their inverses would provide here new methods to derive semantic definitions.

References

[1] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. *ACM Trans. on Program. Lang. and Systems*, 17(1):28–44, 1995.

[2] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. In A. Mycroft, ed., *Proc. of the 2nd Static Analysis Symp.*, Lecture Notes in Comp. Sc. 983, pp. 100–117. Springer-Verlag, 1995.

[3] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Proc. of the 21st ACM Symp. on Principles of Programming Languages*, pp. 227–239. ACM Press, 1994.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages*, pp. 238–252. ACM Press, 1977.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of the 6th ACM Symp. on Principles of Programming Languages*, pp. 269–282. ACM Press, 1979.

[6] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. of the 19th ACM Symp. on Principles of Programming Languages*, pp. 83–94. ACM Press, 1992.

[7] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. Lloyd, ed., *Proc. of the 1995 Int. Logic Programming Symp.*, pp. 321–335. The MIT Press, 1995.

[8] R. Giacobazzi and F. Ranzato. Compositional optimization of disjunctive abstract interpretations. In H.R. Nielson, ed., *Proc. of the 6th European Symp. on Programming.* Lecture Notes in Comp. Sc., Springer-Verlag, 1996.

[9] F. Nielson. Tensor products generalize the relational data flow analysis method. In M. Arató *et al.*, eds., *Proc. of the 4th Hungarian Computer Science Conf.*, pp. 211–225, 1985.