Domain Theory in Abstract Interpretation

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

> *"My ambition [...] is to develop an understanding of the mathematical ideals of programming languages and to combine them with other principles of common sense as correctives of exaggeration, allowing the individual reader to draw as moderate conclusions as he will" [41]*
> — Christopher Strachey, March 1973.

According to a recent and widely recognized definition: *"Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems"* [10]. Abstract interpretation was originally developed by Patrick and Radhia Cousot [13] as a unifying framework for specifying and then validating static program analyses. In recent years this theory has increasingly gained popularity as a general methodology for describing and formalizing approximate computations in many different areas of computer science, like for instance in theorem proving [37], model checking [5], verification of distributed memory systems [31], type inference [12], constraint solving [4], query optimization [32], and comparative semantics [6, 17, 11, 23]. The success of abstract interpretation stems from its simple, but nevertheless rigorously defined, underlying idea that the specification of the behavior of a system, e.g. a program, at different levels of abstraction, is an approximation of its formal semantics. We consider here the classical case of programming languages and their semantics approximation, although the discussion that follows is also relevant in all the above fields, and in every area where abstract interpretation is applicable.

One of the most fundamental facts of abstract interpretation is that most properties of the abstract semantics, like precision, completeness, and compositionality, which may involve complex operators, fixpoints etc., all depend upon the notion of *abstraction*, which is precisely and uniquely specified by the chosen domain of properties [14]. Central in the construction of an abstract interpretation is therefore the notion of *domain*. A domain of properties is a set of mathematical objects which represent the properties of interest about a dynamic system, and are related with each other with respect to their relative degree of precision. Therefore any formal method to compare or transform abstract interpretations is inherently based on corresponding methods to compare and transform abstract domains. The goal of this survey is to promote a theory and a systematic methodology to design domains for abstract interpretation. This is in contrast to the existing folklore that wants domains (e.g. in program analysis) to be guessed in a naive and in most cases ad hoc way, out of the blue.

The foundation of a theory of domains for abstract interpretation was fixed in 1979 by Cousot and Cousot [14]. We exploit these ideas, and go beyond towards an advanced algebraic framework where domains, and therefore semantics, at different levels of abstraction, can be composed, refined, compressed, and decomposed according to the need. Recent works have shown that this is possible. In particular it is always possible to systematically construct domains for abstract interpretation from some basic domain of properties by refining, simplifying and compressing it, in such a way that eventually the resulting domain contains the desired information. Well known domains in the field of programs analysis and comparative semantics have been reconstructed in this way. This both provides a better comprehension of existing domains and gives new hints on how these constructions can be generalized for other applications. The methods used to achieve this task closely resemble the ones used in standard *domain theory* (see [1] for an excellent introduction). This justifies the above ambitious, and somehow provocative, title, and the structure of the present survey. Although standard domain theory origins from a different problem than abstract interpretation, viz. the problem of giving models for spaces where computable functions can be defined [39], and leads to different notions of domains than abstract interpretation does, we believe that there is a strong analogy between these two theories, which still deserves further investigation. We follow the structure of Abramsky and Jung's chapter on domain theory in [1] to introduce step by step the structure and the properties of domains in abstract interpretation. We believe that this will help in drawing possible links between these two theories.

Domains individually

Abstract interpretation provides formal methods to approximate *semantics*. A (concrete) semantics for a programming language can be specified by a concrete domain C of objects and a semantic function $[\![\cdot]\!] : \mathit{Program} \to C$, associating with each (well-formed) program $P \in \mathit{Program}$ its semantics $[\![P]\!] \in C$. According to the classical framework [13], an abstract interpretation is defined as a nonstandard (approximated) program semantics, which is obtained from the standard (or concrete) one by substituting the actual (concrete) domain C of computation and its (concrete) semantics $[\![\cdot]\!]$ with, respectively, an *abstract domain* A and a corresponding *abstract semantics* $[\![\cdot]\!]^\sharp$. The notion of approximation is encoded by a suitable partial order on domain's objects. Both C and A are assumed to be complete lattices with respect to this approximation order. The orderings on the concrete and abstract domains describe the relative precision of domain values, somehow in a dual fashion with respect to the standard domains for denotational semantics – the top element representing no information. Therefore, $a \in A$ approximates $c \in C$ if a represents some concrete object $\tilde{c} \in C$ such that $c \leq \tilde{c}$. Essential is therefore to establish a correspondence between abstract and concrete domains. We consider here the more standard *adjoint framework* [13], based on the notion of *Galois connection* [3]: A pair of adjoint functions, respectively the abstraction map $\alpha : C \to A$ and the concretization map $\gamma : A \to C$, are defined such that $\alpha(c) \leq_A a$, or, equivalently $c \leq_C \gamma(a)$, means that a is a correct approximation of c. Clearly in this case $\alpha(c)$ plays the role of the best possible approximation of c in A, i.e. any other approximation of c is less precise than $\alpha(c)$. In this context, given a concrete semantics $\langle C, [\![\cdot]\!]\rangle$ and an abstract domain A, abstract interpretation theory provides systematic methods to design an abstract semantics $[\![\cdot]\!]^\sharp$ such that it correctly approximates $[\![\cdot]\!]$: *For all programs* P, $\alpha([\![P]\!]) \leq_A [\![P]\!]^\sharp$.

Weaker notions than Galois connection have been studied to include domains that do not fit into the above adjoint framework, in particular in the case of program analysis [16, 34]. The lack of a Galois connection in relating abstract and concrete domains is typically due to the possible lack of best approximations for concrete domain objects, e.g. there may exist many possible optimal approximations but not a unique best one. This happens (rarely) when abstract domains are designed to be directly applicable in effective (terminating) computations, e.g. in program analysis. Because of this, abstract interpretation theory has not to be confused with Galois connections: It provides additional and powerful tools like *widening/narrowing operators* for gaining efficiency in the implementation of abstract domains. Although this is true, in particular in practical applications, we believe that Galois connections still represent the essence of any approximation process. This is justified by the following basic observations: (0) abstract interpretation is not static analysis! for a great number of applications of abstract interpretation, in for instance comparative semantics, Galois connections are perfectly adequate and natural; (1) most well known and successfully used abstract domains, also in static analysis, fit into the adjoint framework; (2) Galois connections provide domains with a relevant (mathematical) structure, making available to the designer a great number of results coming from the basic theory of adjoint functions, this is not the case for other frameworks which are often so weak that any deep and formal treatment of abstraction becomes impossible; (3) the meaning of abstract objects is given in the concrete semantics (i.e. in C) which is the only available semantics, and any subset $B \subseteq C$ of the concrete domain can be extended to the least set A such that $B \subseteq A$ and A fits into the adjoint framework.

Therefore, at least theoretically, the lack of the adjoint framework in relating abstract and concrete domains seems to be more a pathological situation than a real need. Note that by point (3) above, the extension of any domain to fit into the adjoint framework may lead to a combinatorial explosion of the original domain. Although this is easily acceptable in all areas where the abstract semantics is not directly used for effective computations, e.g. in comparative semantics, one may object that this becomes unacceptable in, for instance, program analysis. In contrast we believe that the gap that separates the theoretical specification of an abstract domain from its practical use into a program analyzer is mostly a matter of *efficient implementation*, and therefore can be easily handled with suitable widening/narrowing operators. Any domain that fits into the adjoint framework can be rightfully considered as an ideal optimal starting point to develop further levels of abstractions to gain efficiency in implementation, either by further Galois connections (abstractions) or by widening/narrowing operators to enforce/accelerate convergence. Therefore for the purposes of this survey, in the following, we will call abstract domain any domain A which is related by a Galois connection with a given fixed concrete domain of reference C. In particular we are interested in non-redundant domains, i.e. in domains where each point represents some properties of the concrete domain. In this case we assume that α is onto, or equivalently γ is 1-1, and say that (α, C, A, γ) specifies a *Galois insertion*. More formally, A is an isomorphic representation for a complete meet-subsemilattice of C. This is well known [3] to be a necessary and sufficient condition in order that A is not redundant and belongs, together with C, to the above adjoint framework. It is known that any Galois connection may be lifted to a Galois insertion by identifying in an equivalence class those values of the abstract domain with the same concrete meaning. This process is known as *reduction* of the abstract domain [14].

Domains collectively

When viewing domains collectively, it is often more practical to consider them independently from the specific representation for their objects, otherwise for instance it would not be even possible to collect them into a set, being the collection of all abstract domains a proper class. A simplification is possible by observing ([14]) that the process of approximating concrete domain objects is basically the result of two steps: (1) identifying a subset of the concrete domain which represents the properties of interest, and (2) finding an efficient representation for these objects. The first step uniquely determines the meaning of abstract objects and gives us the possibility to abstract over specific representations. This step can be formalized by an approximating operator $\rho : C \to C$ such that: $\rho(c)$ approximates c (i.e. $c \leq \rho(c)$), ρ respects the order of relative precision of objects (ρ is monotone), and you cannot further approximate an already approximated object (ρ is idempotent). This specifies abstract domains as *upper closure operators*: An abstract domain is the image $\rho(C)$ of the concrete domain with respect to a given upper closure operator. Note that Galois insertions and closure operators are perfectly equivalent methods to specify abstract domains: If $\rho \in uco(C)$ and $A \cong \rho(C)$ (with $\iota : \rho(C) \to A$ and $\iota^{-1} : A \to \rho(C)$ being the isomorphism) then $(\iota \circ \rho, C, A, \iota^{-1})$ is a Galois insertion; if (γ, D, C, α) is a Galois insertion then $\rho_A = \gamma \circ \alpha \in uco(C)$ is the closure associated with A such that $\rho_A(C) \cong A$ [14]. We identify with $uco(C)$, the set of all upper closure operators on C, the so-called *lattice of abstract interpretations* of C (cf. [13, Section 7] and [14, Section 8]), i.e. the complete lattice of all possible abstract domains (modulo isomorphic representation of their objects) of the concrete domain C. The top closure $\lambda x.\top$ is the top domain $\{\top\}$, which contains no information about C. The bottom closure $\lambda x.x$ is the bottom domain C which is the most precise approximation of C. All other domains are partially ordered by the ordering on $uco(C)$ which corresponds precisely to the standard order used to compare domains with regard to their precision: A_1 is *more precise* than A_2 (or A_2 is an abstraction of A_1) iff $A_1 \sqsubseteq A_2$ in $uco(C)$, which holds iff $A_2 \subseteq A_1$. The *lub* and *glb* on $uco(C)$ have therefore the following meaning as operators on domains. Suppose $\{A_i\}_{i \in I} \subseteq uco(C)$: (i) $\sqcup_{i \in I} A_i$ is the most concrete among the domains which are abstractions of all the A_i's, i.e. $\sqcup_{i \in I} A_i$ is the *least common abstraction* of all A_i's; (ii) $\sqcap_{i \in I} A_i$ is (isomorphic to) the well-known *reduced product* (basically cartesian product plus reduction) of all the A_i's, or, equivalently, it is the most abstract among the domains (abstracting C) which are more concrete than every A_i.

The lattice of abstract interpretations inherits all the relevant algebraic properties of the lattice of all closure operations [35, 42]. However, this is not a one-way fertilization. Recent researches on the structure of abstract domains have put forward interesting hidden properties of $uco(C)$ which have mathematical interest. In [24], the authors proved that whenever C is a meet-continuous complete lattice (i.e., for any chain $Y \subseteq C$ and $x \in C$, $x \wedge (\vee Y) = \vee_{y \in Y}(x \wedge y)$), $uco(C)$ is pseudocomplemented. Recall that the *pseudocomplement* of x, if it exists, is the (unique) element x^* such that $x \wedge x^* = \bot$ and $\forall y.\ (x \wedge y = \bot) \Rightarrow (y \leq x^*)$. This result has been applied in [8] in order to define a notion of domain complementation in abstract interpretation. Starting from any two domains C and A, where A abstracts C, i.e. $C \sqsubseteq A$, and C is meet-continuous, the *complement* of A in C is defined as the unique most abstract domain $C \sim A$ such that $(C \sim A) \sqcap A = C$. Note that meet-continuity is a sufficiently weak hypothesis to include most abstract domains used in practice. Complementation is important for *abstract domain decomposition*: By complementation, if $C \sqsubseteq A$ then $\langle C \sim A, A\rangle$ is a binary decomposition for C, i.e. $C = A \sqcap (C \sim A)$ [8]. The advantage of domain decomposition is twofold: (1) it provides compact representations for complex domains, and (2) it simplifies verification problems for complex domains, by decomposing them into simpler problems for their factors.

Recursive domain equations

The main idea behind the use of domain theoretical methods in abstract interpretation is to specify abstract domains systematically from the specification of some basic domain of properties of interest. Basically, we are interested in enhancing the above lattice of abstract interpretations with systematic operations that transform domains. The central notion in this theory is that of *abstract domain refinement* and its dual operations of *abstract domain simplification* and *abstract domain compression* [20, 28]. Intuitively a refinement is any operator performing an action of refinement with respect to the standard ordering of precision, e.g. by adding information to domains; while simplificators and compressors perform the dual action of "taking out" information from domains. Again there is a strong link between operators for refinements/simplifications/compression of domains and *closure operators*. Let us see this theory in more detail.

The refinement $\Re(A)$ of a domain A is a new domain which includes more information than A, i.e. $\Re(A) \sqsubseteq A$. Moreover a refinement respects the order of precision among domains—if we have more information then we add more information. Therefore a refinement is a reductive and monotone operator on $uco(C)$. Relevant examples of these operators include Cousot and Cousot's *reduced product* discussed above [14], *disjunctive completion* [14, 18, 15, 22, 33] and *reduced cardinal power* [14]; Nielson's *tensor product* [36]; Giacobazzi and Ranzato's *dependencies*, *dual-Moore-set completion* [25], and *complete extension* [27], the *open product* and *pattern completion* of Cortesi et al. [9], and Giacobazzi and Scozzari's *Heyting completion* [30].

Most well known operations for domain refinement, including some of those listed above, are designed in such a way that they add to input domains certain specific functionalities which are directly inherited from the structure of the concrete domain. This is the case when the structure of the concrete domain is more complex than a simple complete lattice, e.g. it is a (possibly many sorted) algebra $\mathcal{C}$ with some basic semantics operators $\mathcal{C} = \langle C, op_1, \ldots, op_n\rangle$, for instance when domains represent suitable data structures with corresponding operators. In this case a typical pattern for refining a domain is to enhance it with the functionalities specified by the semantic operators op_i. We say that a domain specified by a closure ρ is *complete* for a function $op : C \to C$ when $\rho \circ op = \rho \circ op \circ \rho$. Basically completeness means that there is no additional loss of information by computing op in the abstract domain. This desirable property can be viewed as the common goal of any refinement: Modifying a domain in such a way it becomes complete for certain basic operations. A typical refinement is therefore a mapping $\Re$ which transforms any input domain $A \in uco(C)$ in such a way that $\Re(A)$ becomes complete (or even a subalgebra) for the operations in $\mathcal{C}$. In particular we are interested into the most abstract domain (when it exists unique) which includes A and for which this holds. A recent result in [28, 27] has shown that this is always possible when the basic operations are continuous functions. This means that the new upgraded domains are rich enough to both include A and give a complete interpretation for the basic concrete operators op_i in A, without including redundant information. These ideal refinements are clearly idempotent, i.e. they upgrade domains all at once, and therefore they can be considered as *lower closures* on the lattice of abstract interpretations, i.e. $\Re \in lco(uco(C))$ [28]. Any idempotent refinement can be systematically and constructively designed as solution of simple recursive abstract domain equations. It is sufficient to identify a basic non idempotent refinement operation $\mathcal{O} : uco(C) \to uco(C)$, such that the refinement of a given domain A can be viewed as the most abstract solution, viz. the greatest fixpoint, of the following recursive abstract domain equation $X = A \sqcap \mathcal{O}(X)$. Clearly, the above construction can be easily generalized to tuples of domains and many sorted concrete algebras, e.g. corresponding to abstractions of different data types, by considering systems of abstract domain equations. Domains can therefore be directly derived from the given programming language and a basic domain of properties. A relevant example of this construction in program analysis can be found in [40]. In this work the author proved that the domain Pos, for ground dependency analysis, can be constructively characterized as the most abstract solution of the following simple abstract domain equation $X = G \sqcap (X \to X)$ where G is the basic domain for pure (non relational) groundness and $\to$ is the binary operation of Heyting completion refinement. A similar constructive characterization for the complex domain of integer intervals in [13] has been obtained in [27], by solving a simple domain equation which includes the simple 4-points lattice of sign analysis as the basic domain of properties and integer addition as concrete operation.

Clearly, the construction of domains in the above way may lead to too complex domains for practical applications, like for instance in the case of sign analysis vs. intervals above. As observed in [28] it is possible to define a dual theory of domain simplificators and compressors, which shares with the above theory of refinements precisely the same, but dual, ideas and constructions. The common aspects of simplificators and compressors is that they both reduce precision in domains. These are particularly important operations to reduced the complexity of domains in program analysis [43]. A typical pattern for domain simplificators is the operation that transforms a given domain A into the most concrete (when it exists) among the abstractions of A which is are complete for the operations in $\mathcal{C}$. Likewise refinements, also simplificators and compressors have a constructive definition as solutions of (systems of) recursive domain equations [27].

The main difference between simplificators and compressors can be understood by viewing how they react when composed with the corresponding refinements, when they exist. Assume that an idempotent refinement $\Re$ is given. $\Re$ admits a simplificator $\mathcal{S}$ when $\Re(\mathcal{S}(X)) = \mathcal{S}(X)$ and $\mathcal{S}(\Re(X)) = \Re(X)$. This holds when both $\Re$ and $\mathcal{S}$ transform domains to meet a given common property, like for instance in the above case of completeness. A relevant example of domain refinement which has a corresponding simplificator is in fact the *complete extension* refinement in [27], which, given an input domain A, returns the most abstract domain which includes A and which is complete for some given semantic operator op. In this case the corresponding simplificator, called *complete kernel*, returns the most concrete domain which is contained in A and is complete for op [27]. Compressors instead act like *gzip* on files: If $\Re$ is a given domain refinement, $\mathcal{S}$ is a compressor for $\Re$ if $\Re(X) = \Re(\mathcal{S}(X))$ and $\mathcal{S}(\Re(X)) = \mathcal{S}(X)$, namely when $\mathcal{S}(A)$ is the most abstract domain such that $\Re(X) = \Re(\mathcal{S}(X))$, which holds basically when the whole refined domain $\Re(A)$ can be fully reconstructed by refinement from its *basis* $\mathcal{S}(A)$. Examples of domain compressors are *complementation* in [8, 21] and discussed above, which is the compressor associated with reduced product, and *least disjunctive basis* in [29], which is associated with disjunctive completion refinement. Clearly, not all refinements admit a corresponding simplificator or compressor. Actually, as suggested by the above definitions, it is possible to relate refinements and simplificators/compressors by means of adjunctions [28].

How to cook an abstract domain

We are now in the position to use the above tools for "cooking" an abstract domain for specific applications in abstract interpretation. The following general rules can be helpful as guidelines for this task.

1. Specify a concrete semantics for the considered programming language, with a (possibly many sorted algebra as) concrete domain $\mathcal{C} = \langle C, op_1, \ldots, op_n \rangle$;

2. Identify, as a subset of the lattice of abstract interpretations, some basic semantics properties $\pi \subseteq uco(C)$ that we want to preserve during the abstraction process;
3. Design a suitable domain refinement $\Re_\pi$ which refines domains by adding some functionalities of the concrete algebra $\mathcal{C}$, in such a way that $\Re_\pi(X) = X \ \Rightarrow X \in \pi$;
4. Specify some basic domain A, representing the basic properties of interest (e.g. the basic properties we want to analyze) concerning concrete data objects;
5. Solve the (system) of recursive domain equations $X = A \sqcap \Re_\pi(X)$;
6. Specify a domain simplificator, compressor, or widening operation on the solution of the above domain equation, when efficiency is needed.

Step (1) is common in any abstract interpretation, and corresponds to the design a suitable base (collecting) semantics. Step (2) instead is a meta-level operation: The designer has to identify the common structure of any domain which shares a given semantic property that has to be preserved during the abstraction process. This may include completeness, compositionality, and any combination of semantic properties of interest for the specific application. A taxonomy of basic observable properties of semantics is essential in order to solve this problem [2, 6, 7]. Step (3) is strongly related with step (2) and is based on the theory of domain refinements described above [28]. Step (4) strongly needs the creative contribution of the designer, which has to guess a minimal domain of basic properties of interest for concrete data objects. Compressors may provide here a great help for simplifying and adapting the solutions generated at design time. Steps (5) and (6) are based on the above observations and can be iterated towards the desired final result. Most of these steps, in particular (3), (5) and (6), are systematic and, in most cases, constructive. Therefore, the designer is provided with all the tools for tuning the process of domain construction in accuracy and costs.

Domains and logic

A desirable feature that any theory of domains should have, is the ability to give a clean logical interpretation for the basic operations used for domain manipulation. This is an ambitious task [1], which may help the designer to specify domains in terms of the logical properties that their objects have to satisfy. This provides a better control on the design process of a domain, by giving a clean interpretation on the logical relation between the various components of a domain. For instance reduced product and complementation have both a clean logical interpretation: The first is conjunction of domain's properties, while the second provides some sort of negation. Unfortunately, the logic structure (if any) of the lattice of abstract interpretations, a kind of *Stone duality* result for abstract interpretation, is not known yet. Although this is (today) the major limitation of this side of the theory, still lot of efforts have been spent to give a clean logical meaning to some basic operators for domain transformation, which are used in most domains in, for instance, program analysis and comparative semantics. This is the case for disjunctive completion, which provides domains with some sort of disjunctive information, and Heyting completion, which provides a logical interpretation for Cousot's reduced cardinal power of domains in terms of intuitionistic implications between their objects. Much research still has to be done in this direction, e.g. by generalizing the above approaches towards a clean and uniform logical interpretation for most domain refinements, simplificators, and compressors.

Applications

Although the above theory of domains can be applied in all areas where abstract interpretation is applicable, program analysis still represents the typical target of applications for this theory. Domain operations provide here high level facilities to tune the analysis in accuracy and cost. This is particularly important, for instance, in analysis of reactive systems [19] and in abstract model checking [5], where refinements may play a key role, providing the systematic derivation of optimal abstract (simplified) systems which simulate the original concrete one in such a way that they satisfy as many as possible of its properties. Particular important is also the case of so-called "Web Languages" and Network computing. They extend the traditional areas of applications of program analysis, to programs which are partially defined entities, whose structure can be modified at run time – e.g. by downloading new classes in Java, and suggest new interesting problems like *security*. Due to the partially defined nature of these programs and the continuous evolution of the environment where programs are executed, it is not possible to fix a priori a domain of properties for analyzing them. The idea here is that we might dynamically refine or simplify the domain to obtain more information about the evolving environment where programs are executed.

The application of the above techniques in program analysis, in particular refinements, may considerably increase the global cost of the analysis, perhaps so much that it becomes unacceptable. The idea of domain simplification has to be improved in this sense, and new domain simplificators or compressors has to be specified, with the aid of experimental evaluation. Of particular interest would be in this case a domain simplificator that transforms domains in such a way that the simplified domain belongs to a different (lower) complexity class than the input domain. It is worth noting that the idea of domain simplificators was already included and implemented for this task, together with some basic refinements, into a prototype analyzer generator called System Z [43].

Abstract interpretation can be used to establish relationships between various semantics or systems of types at different levels of abstraction [17, 11, 12]. Comparative semantics and type inference represent two important areas of application of the above theory of domains. This because the same technology presented above for domain construction can be used for building concrete semantics or type systems. Because each semantics in uniquely specified by an abstraction function, it is possible to combine, refine, compress, decompose and simplify semantics as well as any abstract domain. As a consequence, also step (1) and (2), in the above rules for domain design, can be the result of a constructive process similar to that of domains. Results in this direction have been obtained in the systematic design of semantics for logic programs in [6, 23, 26, 25].

Acknowledgments. Most results presented in this survey would not exist without the fundamental contribution of Francesco Ranzato, with whom I developed, since 1995 when we met at LIX–Ecole Polytechnique, most of the results on domain transformation presented here; part of which represent the main body of his Ph.D. thesis [38]. I am also particularly grateful to Gilberto Filè, with whom we developed the "manifesto" of this research in [20] and whose ideas were at the origin of this research; and to Francesca Scozzari for her contribution in the most recent researches in this field, in particular in the logic of domains and in abstract domain equations.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 2–168. Oxford University Press, Oxford, U.K., 1994.

[2] G. Amato and G. Levi. Properties of the lattice of observables in logic programming. In *Proc. of the Italian-Portuguese-Spanish Joint Conf. on Declarative Programming,* (*APPIA-GULP-PRODE '97*), pages 175–187, 1997.

[3] G. Birkhoff. *Lattice Theory.* AMS Colloquium Publication, 3rd edition. AMS, Providence, RI, 1967.

[4] Y. Caseau. Abstract interpretation of constraints on order-sorted domains. In *Proc. of the 1991 Internat. Logic Programming Symp.* (*ILPS '91*), pages 435–452. The MIT Press, Cambridge, Mass., 1991.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[6] M. Comini and G. Levi. An algebraic theory of observables. In M. Bruynooghe, editor, *Proc. of the 1994 Internat. Logic Programming Symp.* (*ILPS '94*), pages 172–186. The MIT Press, Cambridge, Mass., 1994.

[7] M. Comini, G. Levi, and M. C. Meo. Compositionality of *SLD*-derivations and their abstractions. In J. Lloyd, editor, *Proc. of the 1995 Internat. Logic Programming Symp.* (*ILPS '95*). The MIT Press, Cambridge, Mass., 1995.

[8] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[9] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Conference Record of the 21st ACM Symp. on Principles of Programming Languages* (*POPL '94*), pages 227–239. ACM Press, New York, 1994.

[10] P. Cousot. Abstract interpretation. *ACM Comp. Surv.*, 28(2):324–328, 1996.

[11] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation (Invited Paper). In S. Brookes and M. Mislove, editors, *Proc. of the 13th Internat. Symp. on Mathematical Foundations of Programming Semantics* (*MFPS '97*), volume 6 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 1997.

[12] P. Cousot. Types as abstract interpretations (Invited Paper). In *Conference Record of the 24th ACM Symp. on Principles of Programming Languages* (*POPL '97*), pages 316–331. ACM Press, New York, 1997.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.

[14] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.

[15] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Program.*, 13(2-3):103–179, 1992.

[16] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Computation*, 2(4):511–547, 1992.

[17] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM Symp. on Principles of Programming Languages* (*POPL '92*), pages 83–94. ACM Press, New York, 1992.

[18] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) (Invited Paper). In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '94*), pages 95–112. IEEE Computer Society Press, Los Alamitos, Calif., 1994.

[19] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[20] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.

[21] G. Filé and F. Ranzato. Complementation of abstract domains made easy. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming* (*JICSLP '96*), pages 348–362. The MIT Press, Cambridge, Mass., 1996.

[22] G. Filé and F. Ranzato. The powerset operator on abstract interpretations. To appear in *Theor. Comput. Sci*, 1998. Preliminary version in *Proc. ILPS '94*, pages 655–669, The MIT Press, Cambridge, Mass.

[23] R. Giacobazzi. "Optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech and R. Reischuk, editors, *Proc. of the 13th Internat. Symp. on Theoretical Aspects of Computer Science* (*STACS '96*), volume 1046 of *Lecture Notes in Computer Science*, pages 503–514. Springer-Verlag, Berlin, 1996.

[24] R. Giacobazzi, C. Palamidessi, and F. Ranzato. Weak relative pseudo-complements of closure operators. *Algebra Universalis*, 36(3):405–412, 1996.

[25] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. Lloyd, editor, *Proc. of the 1995 Internat. Logic Programming Symp.* (*ILPS '95*), pages 321–335. The MIT Press, Cambridge, Mass., 1995.

[26] R. Giacobazzi and F. Ranzato. Complementing logic program semantics. In M. Hanus and M. Rodríguez Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming* (*ALP '96*), volume 1139 of *Lecture Notes in Computer Science*, pages 238–253. Springer-Verlag, Berlin, 1996.

[27] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proc. of the 6th Internat. Conf. on Algebraic Methodology and Software Technology* (*AMAST '97*), Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[28] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '97*), volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, Berlin, 1997.

[29] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. To appear in *Sci. Comput. Program*, 1998. Preliminary version in *Proc. ESOP '96*, LNCS 1058, pages 141–155, Springer-Verlag, Berlin.

[30] R. Giacobazzi and F. Scozzari. Intuitionistic implication in abstract interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. of the 9th Internat. Symp. on Programming Languages, Implementations, Logics and Programs* (*PLILP'97*), volume 1292 of *Lecture Notes in Computer Science*, pages 175–189. Springer-Verlag, Berlin, 1997.

[31] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. To appear in *Distributed Computing*, 1997. Preliminary version in *Proc. CAV '94*, LNCS 818, Springer-Verlag, Berlin.

[32] R. Helm, K. Marriott, and M. Odersky. Spatial query optimization: From boolean constraints to range queries. *J. Comput. Syst. Sci.*, 51(2):197–210, 1995.

[33] T. P. Jensen. Disjunctive strictness analysis. In *Proc. of the 7th IEEE Symp. on Logic in Computer Science* (*LICS '92*), pages 174–185. IEEE Computer Society Press, Los Alamitos, Calif., 1992.

[34] K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30:103–129, 1993.

[35] J. Morgado. Some results on the closure operators of partially ordered sets. *Portug. Math.*, 19(2):101–139, 1960.

[36] F. Nielson. Tensor products generalize the relational data flow analysis method. In M. Arató, I. Kátai, and L. Varga, editors, *Proc. of the 4th Hungarian Computer Science Conf.*, pages 211–225, 1985.

[37] D. Plaisted. Theorem proving with abstraction. *Artif. Intell.*, 16:47–108, 1981.

[38] F. Ranzato. *Disegno sistematico di domini in interpretazione astratta*. PhD thesis, Univ. di Padova, 1997.

[39] D.S. Scott. Domains for denotational semantics. In M. Nielsen and E.M. Schmidt, editors, *Proc. of the 9th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '82*), volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, Berlin, 1982.

[40] F. Scozzari. Logical optimality of groundness analysis. In P. Van Hentenryck, editor, *Proc. of the 4th Internat. Static Analysis Symp.* (*SAS '97*), volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997. To appear in *Theoretical Computer Science*.

[41] C. Strachey. The varieties of programming language. Technical Report PRG-10, Oxford University Comp. Lab., 1973. Preliminary version in *Proc. Int. Computing Symposium*, Cini Foundation, Venice 1972, pages 222–233.

[42] M. Ward. The closure operators of a lattice. *Ann. Math.*, 43(2):191–196, 1942.

[43] K. Yi and W. L. Harrison. Automatic generation and management of interprocedural program analyses. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (*POPL '93*), pages 246–259. ACM Press, New York, 1993.