Toward the Integration of Symbolic and Numerical Static Analysis

Arnaud Venet

Kestrel Technology
3260 Hillview Avenue
Palo Alto, CA 94304
arnaud@kestreltechnology.com

1 Introduction

Verifying properties of large real-world programs requires vast quantities of information on aspects such as procedural contexts, loop invariants or pointer aliasing. It is unimaginable to have all these properties provided to a verification tool by annotations from the user. Static analysis will clearly play a key role in the design of future verification engines by automatically discovering the bulk of this information. The program properties that can be computed by a static analyzer fall in two main categories: numerical properties, like the range of scalar variables, and symbolic properties, like pointer aliasing. There is a striking dichotomy between numerical and symbolic static analysis techniques in terms of both precision and efficiency. On one hand, the analysis of numerical properties of programs has been intensively investigated and a broad spectrum of abstractions has been developed based upon results from linear algebra [17], arithmetic [15], linear programming [7] or graph theory [22]. These algorithms can discover complex program invariants, however their scalability is severely limited. Techniques that partition the variables of the program in small-size packets that are tractable by these algorithms have been recently applied to the verification of large safety-critical software with success [2, 33]. On the other hand, research on symbolic static analysis has mainly focused on achieving scalability through its most prominent application: pointer analysis. A decade long of efforts have raised pointer analysis to the point where million line programs can be analyzed in reasonable time [25, 8, 12, 26, 16, 34, 35]. However, there has been little gain in expressiveness compared to Andersen's original pointer analysis [1], and adding precision causes tremendous scalability issues. For instance, there is no known flow-sensitive pointer analysis that can handle large programs.

We believe that the integration of numerical and symbolic static analysis is one of the key challenges lying ahead for achieving the verification of large-scale programs. Existing automatic verifiers able to analyze large programs make simplifying assumptions which limit the degree of interactions between symbolic and numerical properties. In the case of ASTRÉE [2, 6], the programs considered manipulate simple data structures and pointer aliasing is trivial. C Global Surveyor [33, 3] can discover array-sensitive pointer aliasing relationships, but the value of scalar fields inside aggregate structures is abstracted away. Verifying more general classes of programs requires analyzing a broad variety of data structures including arrays, lists, trees, message queues, sockets, whose shape and contents may be strongly correlated to the value of scalar variables. For example, consider a function that takes a slice of an array of floating-point numbers and stores it in a list. The static analysis should be able to infer that each element of the list is equal to the corresponding one in the array, thus relating the position of an element in the list with the position of the corresponding one in the array. Although there is probably no general-purpose solution to that problem, there is a need to design a general algebraic framework that allows us to systematically construct integrated symbolic and numerical static analyses. The body of research on shape analysis [23] and three-valued logic [18] provides a general framework that has been primarily developed for verifying nontrivial properties of dynamic data structures [24], and which has been endowed more recently with numerical abstractions [14, 13]. However, the graph-based abstraction of memory cannot distinguish between elements of chained or aggregate data structures, and despite efforts for optimizing the computations [20], the algorithms do not scale beyond a few thousand lines of code.

Bridging the gap between symbolic and numerical static analysis without sacrificing scalability seems an inextricable problem. We propose an approach that radically departs from standard static analysis design methodologies and which can be summarized as follows: "use known scalable symbolic analysis algorithms and recast any additional required symbolic information into a numerical property". The benefits of this approach are twofold: first, we can use the large number of existing numerical abstractions to encode a broad spectrum of symbolic properties; second, correlating symbolic properties of the program with numerical ones becomes straightforward, since both are expressed in the same setting. Most of our research work consisted of applying this approach to the analysis of real-world programs, and Section 2 gives an overview of the results obtained in this domain. At first glance, the various static analyses developed under this approach [9, 10, 27, 29, 31–33] bear little resemblance one with another, apart from the fact they embed an abstract numerical lattice within a symbolic structure. In Sect. 3 we show that the core mechanism of all these static analyses can be expressed as an instance of a general algebraic structure, that we call *abstract fiber bundle* in reason of its intriguing connection with Algebraic Topology. In Sect. 4 we discuss the relevance of this approach for the design of large-scale verification systems.

2 Mixed Symbolic and Numerical Static Analysis

The first occurrence in the literature of a static analysis that mixes symbolic and numerical approximations is an alias analysis for strongly typed languages [9, 10] that is able to discover properties such as "two lists of arbitray length share their elements pairwise". In that model, pointer aliasing is represented by an equivalence relation over access paths into data structures. The abstraction is based on a finite partitioning of the set of access paths by monomial unitary-prefix path expressions, which are given by the Eilenberg decomposition of a rational language [11]. Monomial unitary-prefix path expressions have the form $\pi_1 B_1^* \pi_2 B_2^* \ldots \pi_n B_n^* \pi_{n+1}$ where the π_i are sequences of data selectors and the B_i are rational languages, called the bases of the decomposition. The key idea consists of assigning a counter variable to each base and use standard numerical lattices to set constraints between these counters. For example, two lists `x` and `y` that share their elements pairwise can be described as follows:

$$\texttt{x.(tl)}^i\texttt{.hd} \equiv \texttt{y.(tl)}^j\texttt{.hd} \iff i = j$$

by using the numerical lattice of affine equalities [17]. The pointer aliasing relation is thus completely abstracted by a finite number of numerical relations. We have designed an abstraction of relations over free monoids inspired by this model that did not require any type annotation and did not incur the possible exponential cost of the Eilenberg decomposition [27]. The main idea was to use a regular automaton as the base symbolic structure and assign a numerical counter to each transition of the automaton. The automaton describes the access paths within data structures and is constructed jointly with the aliasing relation. Since the aliasing relation is based on this structure, changing the automaton requires to modify the representation of the aliasing relation accordingly. This operation was carried out by endowing the abstract domain with the structure of a cofibered domain [27]. This allowed us to construct a pointer analysis of similar power for dynamically typed languages like Java [30], as well as a communication analysis for systems of concurrent processes based on the π-calculus [28]. However, this numerical model has two important drawbacks: the operations on aliasing relations are costly and arrays cannot be represented precisely.

In order to lift these limitations we built a new numerical model based on a different interpretation of the semantics of memory allocation. Each object allocated in memory is assigned a timestamp, which is a numerical abstraction of the execution trace that led to the object creation. The memory is represented by a graph whose vertices are labels of allocation statements together with a timestamp, and whose edges represent the points-to relation. Arrays can naturally be integrated into this scheme by simply adding a numerical index to edges. This new model allowed us to build a flow-sensitive pointer analysis for Java-like languages [31] and a considerably simpler communication analysis for the π-calculus [29]. It also allowed us to tackle the analysis of multithreaded programs. Flow-sensitive analyses are impractical in the presence of threads due to the combinatorial blowup of interleaving. We have developed a pointer analysis for the C language that lies between flow-sensitive and flow-insensitive analyses [32]. An inexpensive flow-sensitive analysis is first run on each function in order to build flow-insensitive points-to equations that incorporate all local loop invariants. Then, these equations are solved using a constraint resolution algorithm. This analysis can be seen as an homeomorphic extension of Andersen's analysis scheme [1] in which inclusion constraints are annotated by numerical invariants. The constraint resolution algorithm is similar to Andersen's except that numerical operations are performed at each elementary step. The analysis scales well and has been successfully applied to the control software of a science payload for the International Space Station [32].

These encouraging results motivated us to apply these techniques to the large mission-critical programs developed at NASA for the Mars Exploration Program. We have developed a static array-bound checker for NASA flight software, called C Global Surveyor, which is based on a numerical abstraction of the heap [33]. The focus of this tool was not so much on memory allocation, which is scarcely used in mission-critical software, but on pointer arithmetic. In the family of programs considered, data are organized in large structures and manipulated by transmitting their address to generic functions. We designed a model in which all data are referenced using a byte-based offset within the memory block where they belong. The abstract heap is a points-to graph labeled with numerical intervals representing offset ranges. This graph is iteratively refined by narrowing intervals and pruning edges. The process is bootstrapped by using the memory graph produced by Steensgaard's analysis [25], and subsequent phases essentially consist of arithmetic manipulations on the labels of the graph. We have applied this static checker to codes ranging from 140 KLOC to 550 KLOC (the flight software of the current mission Mars Exploration Rovers). On average, 80% of all array accesses could be decided by the verifier, with the analysis speed peaking at 100 KLOC/hour [3]. The only limiting factor was the enormous amount of artifacts produced by the analyzer, which forced us to use an external storage management that degraded the performances.

3 Abstract Fiber Bundles

At first glance, the static analyses described in the previous section may look like a disparate collection of ad hoc algorithms complex to design and implement. If it were so, there would be no point in promoting a general approach to integrating numerical and symbolic static analyses. In this section, we would like to show that the core structure of *all* static analyses above mentioned [10, 30, 29, 31–33] are instances of a generic algebraic structure that precisely defines the interaction between the symbolic and numerical aspects of the analysis, and also provides a systematic decomposition of the complex semantic operations involved, making the implementation modular.

We use Abstract Interpretation [4, 5] to define our analysis framework. Our concrete semantic domain is given by a powerset lattice $\mathcal{D} = (\wp(D), \subseteq, \emptyset, \cup, D, \cap)$. In each case we give a simplified construction of $\mathcal{D}$, without loss of generality though, in order to emphasize the common structure of these analyses. In all static analyses considered, D is a set of tuples mixing symbolic values and integers. For storeless alias analyses [10, 30] and the communication analysis of the π-calculus of [28], the aliasing/communication structure is given by an equivalence relation. The analysis of [29] represents the communication structure by a binary relation which is not an equivalence relation. In all those cases, $\mathcal{D}$ is the set of all binary relations on strings, which denote either access paths into data structures or sequences of process interactions. In the alias analysis of [31], the main structure is a points-to relation between ojects, which may be structured records or arrays. These objects are identified by timestamps, which are sequences of tuples of integers. In that case, D contains all triples $\langle t_1, f, t_2 \rangle$, where t_1, t_2 are timestamps and f is either a field selector or an integer (in the case of a points-to relation involving an array). The representation of the points-to graph in C Global Surveyor [33] is given by a set of tuples $\langle v_1, i_1, v_2, i_2 \rangle$ where v_1, v_2 are program variables and i_1, i_2 are offsets expressed in bytes and denoting positions within v_1, v_2. Although it is described differently in the paper, the pointer analysis of [32] is based on an abstraction of inclusion constraints enriched with integer values, which denote timestamps and array indices, like in $\mathcal{X}_i \supseteq *(\mathcal{Y}_j + o)$, where i, j, o are integers. An enriched inclusion constraint of that form can be encoded as a tuple of symbolic variables and integers.

The first step of the abstraction consists of projecting D onto a set B, that we call the *base*, and which contains only symbolic elements. We denote by $\pi : D \to B$ the projection. In the case of the analyses of [9, 10], B is the set of all pairs $\langle \Pi_1, \Pi_2 \rangle$, where Π_1, Π_2 are monomial unitary-prefix path expressions from the Eilenberg decomposition of the language denoting all possible access paths. We will use this instantiation of the framework as our running example. Now, we assign a set V_b of integer-valued variables to each $b \in B$ together with a family of mappings $\phi_b : \pi^{-1}(b) \to \mathbb{N}^{V_b}$, that we call *local trivializations*. The variables of V_b represent the numerical information associated to a symbolic element of the structure. In our example, $V_{\langle \Pi_1, \Pi_2 \rangle}$ contains the counters associated to each base of the Eilenberg decomposition appearing in Π_1 or Π_2. The local trivialization $\phi_{\langle \Pi_1, \Pi_2 \rangle}$ maps a pair $\langle \pi_1, \pi_2 \rangle$ of access paths to the tuple of integers denoting the number of times each base of Π_1 and Π_2 is traversed by π_1 and π_2. Finally, we associate an abstract numerical lattice $\mathcal{F}(b)$ with each $b \in B$, that we call the *fiber* over b. This abstract numerical lattice provides a computable numerical abstraction of sets of integer valuations over the variables V_b through a concretization function $\gamma_b : \mathcal{F}(b) \to \wp(\mathbb{N}^{V_b})$. We call the structure $(D, B, \phi, \mathcal{F})$ an *abstract fiber bundle*. A *section* of the fiber bundle is a family $(\nu_b)_{b \in B}$ of abstract numerical relations, where $\nu_b \in \mathcal{F}(b)$ for each b in B. The collection of all sections endowed with the pointwise extension of the abstract numerical lattice operations forms a lattice $\mathcal{S}$. This defines an approximation $\gamma : \mathcal{S} \to \mathcal{D}$ of the concrete domain as follows:

$$\gamma((\nu_b)_{b \in B}) = \{x \in D \mid \phi_{\pi(x)}(x) \in \gamma_{\pi(x)}(\nu_{\pi(x)})\}$$

In our example, the lattice of sections obtained is isomorphic to the lattice of monomial unitary-prefix relations of [9, 10].

Now, if B is the set of pairs $\langle q_1, q_2 \rangle$ of final states of a deterministic regular automaton $\mathcal{A}$, π maps a pair of access paths in the language recognized by $\mathcal{A}$ to the pair of accepting final states, and $\phi_{\langle q_1, q_2 \rangle}$ maps a pair of access paths to the tuple of integers denoting the number of times each transition of the automaton is traversed by the access paths, then we obtain the abstract domain of [30, 28, 29]. We leave to the reader the task of showing that the abstract domains of the remaining static analyses [31, 33, 32] can be constructed along the same lines. The main idea is that the base contains all symbolic information, whereas the fibers carry all numerical information. Although they bear a similar name, the *abstract cofibered domains* of [27] are orthogonal to this construction. Their main purpose is to formalize the notion of an *adaptive lattice*, where the abstract domain changes during the execution of the static analysis, and to provide a systematic way of constructing widening operators. For example, cofibered domains can be used on top of an abstrat fiber bundle when the base B of the bundle is computed during the analysis, as it is the case in [30, 28]. They do not give any insight into the internal structure of the domain itself, which is the purpose served by the abstract fiber bundle.

The main difficulty in the design of a mixed symbolic and numerical static analysis is the construction of the semantic transformers. The semantic definitions are clogged with intricate manipulations on the variable sets V_b and the abstract numerical relations. We will now show that all these semantic transformations can be carried out in a common algebraic framework using a reduced set of basic operations. We first need a few definitions. Any numerical approximation defined in the literature [17, 7, 15, 21, 22] associates to any set of variables V an abstract numerical lattice $\mathcal{N}(V)$ over V. Given a one-to-one mapping $\iota : U \to V$ between sets of variables, each numerical approximation comes equipped with two functions: a *projection* function $\underline{\mathcal{N}}\iota : \mathcal{N}(V) \to \mathcal{N}(U)$, which eliminates the variables which are not in the image of ι from an abstract numerical relation and renames the others, and an *extension* function $\overline{\mathcal{N}}\iota : \mathcal{N}(U) \to \mathcal{N}(V)$, which lifts an abstract numerical relation defined over U onto V and renames the variables accordingly. It is not difficult to see that for every abstract numerical lattice, the pair $(\underline{\mathcal{N}}\iota, \overline{\mathcal{N}}\iota)$ forms a Galois connection. A numerical approximation $\mathcal{N}$ can be endowed with the structure of a contravariant functor from the category of finite sets and one-to-one mappings to the category of lattices and Galois connections.

Now let $\mathcal{N}$ be a numerical approximation. We assume that we have a language of constraints $\mathcal{C}$ that can be reflected in $\mathcal{N}$, that is, if S is a system of constraints of $\mathcal{C}$ over the variables V, there is an element $[\![S]\!]$ of $\mathcal{N}(V)$ that over-approximates the set of solutions of S in $\mathbb{N}^V$. In practice, the language of affine equality constraints is sufficient and it can be exactly reflected in most relational numerical approximations. We call *local section* a finite family $(U_i, \nu_i)_{i \in I}$ where, for all i in I, U_i is a finite set of variables and $\nu_i \in \mathcal{N}(U_i)$, together with a family $(S_{i,j})_{(i,j) \in I \times I}$ of systems of constraints of $\mathcal{C}$, where $S_{i,j}$ is defined over the variables of the disjoint union $U_i \oplus U_j$. We call the family $(U_i, \nu_i)_{i \in I}$ alone *a local covering*. Moreover, a local section must satisfy the *compatibility condition*, which is defined as follows. If $(i, j) \in I \times I$, we denote by $\iota_i : U_i \to U_i \oplus U_j$ and $\iota_j : U_j \to U_i \oplus U_j$ the canonical inclusion mappings. The compatibility condition says that:

$$\forall (i, j) \in I \times I : \overline{\mathcal{N}}\iota_i(\nu_i) \sqcap \overline{\mathcal{N}}\iota_j(\nu_j) \sqsubseteq [\![S_{i,j}]\!]$$

In practice, most $S_{i,j}$ are empty, that is $[\![S_{i,j}]\!] = \top$. Now, given a local covering $(U_i, \nu_i)_{i \in I}$ that does not satisfy the compatibility condition with respect to a family of constraints $(S_{i,j})_{(i,j) \in I \times I}$, we are interested in finding the greatest local covering $(U_i, \mu_i)_{i\in I}$ for the pointwise extension of the abstract numerical lattice ordering, that satisfies the compatibility condition and that refines $(U_i, \mu_i)_{i\in I}$, i.e. $\forall i \in I : \mu_i \sqsubseteq \nu_i$. All semantic operations of the static analyses under consideration can be stated as instances of this problem. For example, the alias analysis of [30] requires to perform the transitive closure of an abstract aliasing relation. Recall that the base of the fiber bundle is given by pairs $\langle q, q'\rangle$ of final states of a regular automaton. Each transition of the automaton is assigned a numerical counter and the numerical approximation expresses relations between the values of these counters within an alias pair. The basic step of the transitive closure consists of taking two elements $\nu_{\langle q_1,q_2\rangle}$ and $\nu_{\langle q_2,q_3\rangle}$ of a section and compute a new element $\mu_{\langle q_1,q_3\rangle}$ which consists of equating the transition counters associated to the component q_2 in the first two alias pairs, and then project the numerical relation onto $V_{\langle q_1,q_3\rangle}$, renaming the counters accordingly. This actually amounts to finding the greatest local section refining the local covering $\big((V_{\langle q_1,q_2\rangle}, \mu_{\langle q_1,q_2\rangle}), (V_{\langle q_2,q_3\rangle}, \mu_{\langle q_2,q_3\rangle}), (V_{\langle q_1,q_3\rangle}, \top)\big)$ satisfying the compatibility condition under the systems of constraints $(S_i)_{1\leq i\leq 3}$, that express the equality of transition counters between two alias pairs having the state q_i in common.

The greatest refinement of a local covering $(U_i, \nu_i)_{i\in I}$ satisfying the constraints $(S_{i,j})_{(i,j)\in I\times I}$ always exists and can be constructed as follows. For all i, j in I, we denote by $\varepsilon_i : U_i \to \bigoplus_{i\in I} U_i$ and $\sigma_{i,j} : U_i \oplus U_j \to \bigoplus_{i\in I} U_i$ the canonical inclusion mappings. We define

$$\chi = \prod_{(i,j)\in I\times I} \overline{\mathcal{N}}\sigma_{i,j}(\llbracket S_{i,j}\rrbracket)$$

as the conjunction of all compatibility constraints. We define the *gluing function* $G : \prod_{i\in I} \mathcal{N}(U_i) \to \mathcal{N}(\bigoplus_{i\in I} U_i)$ as follows:

$$G((\nu_i)_{i\in I}) = \left(\prod_{i\in I} \overline{\mathcal{N}}\varepsilon_i(\nu_i)\right) \sqcap \chi$$

We define the projection function $P : \mathcal{N}(\bigoplus_{i\in I} U_i) \to \prod_{i\in I} \mathcal{N}(U_i)$ as

$$P(\mu) = (\underline{\mathcal{N}}\varepsilon_i(\mu))_{i\in I}$$

Let $\downarrow \chi = \{\mu \mid \mu \sqsubseteq \chi\}$ be the ideal generated by χ, and $P\mid_{\downarrow\chi}$ the restriction of P to $\downarrow \chi$. We can show using the properties of the functor $\mathcal{N}$, that the pair $(P\mid_{\downarrow\chi}, G)$ defines a Galois connection, and that for every local covering $C = (U_i, \nu_i)_{i\in I}$, the greatest local covering refining C is given by $P \circ G\,((\nu_i)_{i\in I})$. We call the triple $(\mathcal{N}, P, G)$ *the structure sheaf* of the abstract fiber bundle. This means that all semantic transformations can be carried out using only the structure sheaf, the language of constraints $\mathcal{C}$ and the lattice operations.

The reader must have noticed that we have borrowed the names of the structures we have introduced to unify the different models of mixed symbolic and numerical static analyses from Algebraic Topology. The analogy between our model and standard topological constructs is extremely intriguing. The curious reader can check that there is a one-to-one correspondence between each point in the definition of a sheaf and a fiber bundle that we gave and the standard ones [19]. The main difference is that Abstract Interpretation deals with approximation, hence we do not have "exact" constructs as in the topological case. We get lattices instead of groups and Galois connections instead of isomorphisms. We believe that there is more than an analogy here, and that techniques from Algebraic Topology could be used with profit for designing novel abstract interpretation frameworks.

4 Applications to Software Verification

There are two major hurdles when it comes to designing a software verification system based on static analysis. The first one lies in the difficulty of keeping a precise hence complex abstract interpretation framework flexible. Specializing a static analyzer for a family of programs or enriching the abstract domains in order to analyze certain classes of algorithms more precisely are not tasks that can be done once and for all at design time. They require to conduct a lot of experiments on real applications [2, 33]. Therefore, the architecture of the static analyzer must be as modular as possible in order to support this activity. We think that the model we propose is well adapted to incremental refinement. Since all the complexity of the analysis has been shifted into the numerical model, increasing the precision essentially means changing the structure sheaf of the abstract fiber bundle, which is completely transparent as long as the semantic transformers are expressed using sheaf operations.

The second difficulty that arises when using a static analyzer for verification is the interpretation of the results. Whenever the analyzer either detects an error or is unable to conclude, elements of information must be provided to the user. The properties calculated by the static analyzer can be cryptic and very large to print out, especially for complex abstract domains. We think that our model provides a way of taming this problem by using a symbolic abstraction that can be traced back to the program easily. Inclusion-based pointer analysis is a good example [1] of an analysis whose results can be directly manipulated by the user [16]. Recovering readable program properties from the numerical information is an area that remains to be explored.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.
3. G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.
5. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, 2005.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.
8. M. Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.
9. A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.
10. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM Press, 1994.
11. S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.
12. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, 1998.
13. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.
14. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.
15. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.
16. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.
17. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
18. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.
19. S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.
20. R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Static Analysis Symposium*, pages 196–212, 2002.
21. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the 2nd Symposium PADO'2001*, volume LNCS 2053, pages 155–172, 2001.
22. A. Miné. The octagon abstract domain. In *AST 2001 at WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

23. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.
24. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis using 3-valued logic. In *Proceedings of Symposium on Principles of Programming Languages*, 1999.
25. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Computational Complexity*, pages 136–150, 1996.
26. Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.
27. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 266–382. Springer Verlag, 1996.
28. A. Venet. Abstract interpretation of the π-calculus. In *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer, 1997.
29. A. Venet. Automatic determination of communication topologies in mobile systems. In *5th International Symposium on Static Analysis, SAS '98*, volume 1503 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 1998.
30. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.
31. A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis SAS'02*, volume 2477 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2002.
32. A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Proceedings of the International Static Analysis Symposium, SAS 04*, volume 3148 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2004.
33. A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 231–242, 2004.
34. J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, September 2002.
35. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.