Towards 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 body of research in static program analysis can be split up in two major areas: one–probably the larger in terms of publications–is concerned with discovering properties of data structures (shape analysis, pointer analysis); the other addresses the inference of numerical invariants for integer or floating-point algorithms (range analysis, propagation of round-off errors in numerical algorithms). We will call the former "symbolic static analysis" and the latter "numerical static analysis". Both areas were successful in effectively analyzing large applications [19, 8, 12, 2, 6]. However, symbolic and numerical static analysis are commonly regarded as entirely orthogonal problems. For example, a pointer analysis usually abstracts away all numerical values that appear in the program, whereas the floating-point analysis tool ASTREE [2, 6] does not abstract the memory at all.

If one wants to use static analysis to support or achieve verification of real programs, we believe that symbolic and numerical static analysis must be tightly integrated. Consider the two code snippets in Fig. 1. If one wants to check that the assignment operation in the first example is performed within the bounds of the array, one needs a numerical property relating the sizes of the objects pointed to by `p` and `q` and the parameter `n`. The second example constructs a two-dimensional array of semaphores using VxWorks' `semCreate` library function. If one wants to verify concurrency properties of the program, like the absence of deadlocks, one must be able to distinguish between the elements of the `sems` array. In the first case, a static analyzer would have to construct an abstract memory graph labeled with metavariables denoting the size of objects and relate these metavariables with the program variables. The second case is more complex, in the sense that the points-to relation itself has to be parameterized by array indices. These two examples are not artificial: the first one is

```
void equate (int *p, int *q, int n) {
    int i;
    for(i=0; i<n; i++)
       p[i] = q[i];
}
```

```
for(i=0; i<10; i++)
  for(j=0; j<8; j++)
    sems[i][j] = semCreate();
```

Fig. 1. Code samples illustrating the interaction of symbolic and numerical properties

characteristic of the object-oriented programming style used in the flight mission software developed at NASA for the Mars Exploration Program [26, 3]; the second one comes from the controller of a science payload developed at NASA for the International Space Station [25].

Our research work has been mostly concerned with the design of techniques for combining symbolic and numerical static analysis in order to discover the kind of properties described above. We came up with a number of static analysis algorithms [23, 22, 24–26] aimed at various categories of properties and programs. This approach proved to be successful in achieving the large-scale verification of pointer-intensive NASA flight software [26, 3]. The major difficulty in developing those kind of analyses lies in the absence of a general framework for guiding the design. Except for the base idea of blending symbolic and numerical structures together, these analyses vary broadly in terms of the semantic model used, the abstract representation of memory and the resolution algorithms. If we add to this list the fact that these analyses are very complex to implement, one may cast doubts on the viability of this approach for the development of production-level verification tools. This paper proposes a research agenda aimed at making this technology mainstream and easily applicable to a broad spectrum of verification problems. In Sect. 2 we will review the major achievements in the design of mixed symbolic and numerical static analysis tools, and Section 3 will describe the remaining challenges. In Sect. 4 we will sketch the bases of a general abstract interpretation framework for automating the implementation of static analyzers. This framework is the formal foundation for an effort underway at Kestrel Technology to industrialize this static analysis technology. This is discussed in Sect. 5.

2 Achievements

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

$\pi_1 B_1^* \pi_2 B_2^* \dots \pi_n B_n^* \pi_{n+1}$ 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:

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

by using the numerical lattice of affine equalities [13]. 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 [20]. 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 [20]. This allowed us to construct a pointer analysis of similar power for dynamically typed languages like Java [23], as well as a communication analysis for systems of concurrent processes based on the π-calculus [21]. 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 [24] and a considerably simpler communication analysis for the π-calculus [22]. 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 [25]. 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 [25].

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 [26]. 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 [19], 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 Challenges

Anyone reading the literature on mixed symbolic and numerical static analysis would rapidly come to the conclusion that all those analyses are completely ad hoc and difficult to reproduce. If there were not the scalability issues, a general framework like three-valued logic [18, 15] looks more appealing because most of the technicalities are hidden inside a generic engine. Hence, we need a general and practical framework for the integration of symbolic and numerical static analysis. In the following sections, we describe the three major challenges that lie ahead.

3.1 Nonstandard Semantics

A static analysis is defined by an abstract interpretation of the concrete semantics [4, 5]. In order to be able to relate a symbolic information with a numerical one, we need to provide a numerical encoding of the symbolic part. Standard semantic models are almost always inadequate, and this requires the definition of a more appropriate semantics. Examples are the storeless [9, 23] and timestamp [24, 25] semantics of aliasing. This is the most critical design decision since this choice will drive the precision and efficiency of the analysis. The definition of a nonstandard semantics is very similar in its purpose to the choice of instrumentation predicates in three-valued logic [15]. In practice, the definition of a nonstandard semantics–however baroque it may be–is quite straightforward and just takes few lines. The major difficulty is that the whole analysis entirely depends on the choice of this semantics. Finding out after experiments that this semantics is inadequate essentially implies rewriting the whole analysis, which is a tremendous work. Hence, we need to be able to generate most of the analysis implementation from a relatively small specification of the concrete semantics.

3.2 Complexity of the Abstract Domain Construction

The structure of the abstract domains for these analyses is extremely complex, which makes the construction of the semantic transformers very tedious and lengthy. However, the structure of the domains is very modular and the semantic transformers closely mimick those of the concrete semantics. Although the complete formal specification of the analysis is bulky and intricate, its stepwise construction is fairly systematic. If this systematic construction can be mechanized, then the core of the analysis engine can be automatically generated from a description of the abstract domain's structure.

3.3 Techniques for Scalability

Making the analyzer scale to large code bases entails doing some tradeoffs that further complicates the construction of the analyzer. For example, flow-sensitive pointer analysis does not scale up to the hundreds of KLOC, whereas flow-sensitivity is required in order to compute numerical loop invariants. The analyses described in [26, 25] perform a flow-sensitive numerical analysis combined with a shallow pointer analysis first, and then a deep pointer analysis. This means that two different abstract domains have to be constructed, one for each phase of the analysis. To take another example, performances of important numerical abstract interpretations like polyhedra [7] or difference-bound matrices [17] degrade rapidly with the number of variables analyzed, whereas the numerical encoding of symbolic information tends to introduce a lot of variables. The solution consists of partitioning the set of variables in small clusters for which the numerical algorithms perform well. From our experience, the clustering significantly complicates the implementation of the analysis engine. Ideally, these optimizations should be orthogonal to the construction of the abstract interpretation. The fact that they are deeply interwoven with the specification of the analysis makes this separation all the more challenging.

4 Sections, Coverings and Glueing

We propose a unified framework for tackling all the challenges listed in the previous section. Our approach is based on constructions inspired from algebraic topology. The theory of sheaves and fiber bundles [16] in particular, provide techniques for studying global properties of complex topological spaces possessing a regular structure locally. Similarly, the abstract semantic domains used in mixed symbolic and numerical static analysis can be seen as bundles of simple numerical lattices. All semantic operations on these domains can be interpreted as the "patching" of abstract values defined on local lattices. This provides us with a uniform algebraic framework that can be used as the foundation of a generic analysis engine for this class of static analysis. In the rest of this section we will sketch the main lines of our approach, and show how existing mixed symbolic-numerical analyses can be expressed in this framework.

We start with describing the concrete semantic domains of the mixed symbolic-numerical static analyses defined in the literature. In each case the concrete semantic domain is given by a powerset lattice $\mathcal{D} = (\wp(D), \subseteq, \emptyset, \cup, D, \cap)$. In all static analyses considered, D is a set of tuples mixing symbolic values and integers. For storeless alias analyses [10, 23] and the communication analysis of the π-calculus of [21], the aliasing/communication structure is given by an equivalence relation. The analysis of [22] 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 [24], 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 [26] 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 [25] 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.

In each case, an abstract value is a tuple of integers and symbolic values. The first step of the abstract interpretation process consists of building a finite approximation of the set $\mathcal{S}$ of symbolic values, when $\mathcal{S}$ is infinite. This essentially consists of projecting $\mathcal{S}$ onto a finite set B, that we call the *base*. We denote by $\pi : D \to B$ the projection. In the case of the analyses of [9, 10], B is the set of monomial unitary-prefix path expressions Π obtained from the Eilenberg decomposition of the language denoting all possible access paths in the data structures of the program. 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_Π contains the counters associated to each base of the Eilenberg decomposition appearing in Π. The local trivialization ϕ_Π maps an access path π to the tuple of integers denoting the number of times each base of Π is traversed by π.

Now, given such a structure, a tuple τ of integers and symbolic values can be abstracted by a tuple $\bar{b}$ of elements of B, together with a set $V_{\bar{b}}$ of integer variables denoting the integer values in τ plus the variables of the local trivializations associated to each symbolic element of τ. We denote by $\overline{B}$ the set of tuples of elements of B induced by this abstraction. Finally, we associate an abstract numerical lattice $\mathcal{F}(\bar{b})$ with each $\bar{b} \in \overline{B}$, that we call the *fiber* over $\bar{b}$. This abstract numerical lattice provides a computable numerical abstraction of sets of integer valuations over the variables $V_{\bar{b}}$ through a concretization function $\gamma_{\bar{b}} : \mathcal{F}(\bar{b}) \to \wp(\mathbb{N}^{V_{\bar{b}}})$. We call the structure $(D, \overline{B}, \phi, \mathcal{F})$ an *abstract fiber bundle*. A *section* of the fiber bundle is a family $(\nu_{\bar{b}})_{\bar{b}\in\overline{B}}$ of abstract numerical relations, where $\nu_{\bar{b}} \in \mathcal{F}(\bar{b})$ for each $\bar{b}$ in $\overline{B}$. The collection of all sections endowed with the pointwise extension of the abstract numerical lattice operations forms a lattice Σ. This defines an approximation $\gamma : \Sigma \to \mathcal{D}$ of the concrete domain. 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 of final states of a deterministic regular automaton $\mathcal{A}$, π maps an access path in the language recognized by $\mathcal{A}$ to the paccepting final state, and ϕ_q maps an access path q to the tuple of integers denoting the number of times each transition of the automaton is traversed by the access path, then we obtain the abstract domain of [23, 21, 22]. The abstract domains of the other static analyses [24, 26, 25] 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 [20] 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 [23, 21]. They do not give any insight into the internal structure of the domain itself, which is the purpose served by the abstract fiber bundle.

All intermediate data structures that are created and manipulated during the analysis are actually tuples of integer symbolic values. Therefore, the abstract fiber bundle can be used as a universal object factory in the implementation of the analysis engine. This brings us one step closer to a generic analysis framework. Now, given two sets of numerical variables U and V, one can define a morphism $f : U \to V$ as an injective linear transformation between the $\mathbb{Q}$-vector spaces spanned by U and V. Given a numerical abstract domain $\mathcal{N}$, such a morphism defines a projection map $\mathcal{N}f$ from the domain $\mathcal{N}V$, defined over the set of variables V, into the abstract domain $\mathcal{N}U$, defined over the set of variables U. Hence, an abstract numerical domain can be viewed as a contravariant functor. Now, given a family of morphisms $f_i : U_i \to U$ and elements $\nu_i \in \mathcal{N}U_i$, one can define the *glueing* $G\langle(\nu_i)_{i\in I}\rangle$ as the conjunction of all numerical constraints defined by the ν_i transposed into the domain $\mathcal{N}U$ via the morphisms f_i. This structure is reminiscent of that of sheaves frequently used in algebraic topology. This simple algebraic structure is sufficient to express all the operations used in the construction of a mixed symbolic-numerical static analysis. For example, a transitive closure step in the model of [9] is expressed as: "given an alias pair (Π_1, Π_2) with the abstract numerical relation ν_1, and (Π_2, Π_3) with the abstract numerical relation ν_2, construct the alias pair (Π_1, Π_3) together with the composition of ν_1 and ν_2". In our framework, this amounts to glueing ν_1 and ν_3 over the abstract domain $\mathcal{N}(V_{\Pi_1} \cup V_{\Pi_2} \cup V_{\Pi_3})$ and projecting back the result into $\mathcal{N}(V_{\Pi_1} \cup V_{\Pi_3})$.

This sheaf-like structure also allows us to express the optimization techniques used for ensuring scalability. The clustering of variables amounts to using a *covering*, i.e. a family of inclusion morphisms $f_i : U_i \to U$ for representing an abstract numerical relation over U. Since all operations are expressed using the projection morphisms and the glueing operation, using a covering of U instead of U becomes a transparent operation. Similarly, the distinction between shallow and deep pointer analyses vanishes, since they now share the same representation and are manipulated using the same operations. This means that this framework allows us to synthesize an abstract interpretation on top of a generic analysis engine.

5 Automated Generation of Static Analyzers

The implementation of this framework is underway, based on the formal specification environment SpecWare [14]. Our first objective is to be able to reconstruct existing analyses using this framework. In particular, we aim at achieving the same level of scalability. This is probably the main characteristic of our approach: unifying the construction of semantic transformers and the definition of optimizations within a single formal framework. This comes in sharp contrast with three-valued logic for example, where there is no handle for controlling the scalability. Our experience with C Global Surveyor [26] showed that there is no universal strategy for achieving scalability. This is based on a "try and fix" process, driven by empirical data and dependent on the family of applications considered. Therefore, we believe that there is no point in trying to build a sophisticated "push button" tool that will work well on a broad spectrum of applications. It is more important to allow the developers to customize the analysis rapidly and find the best blend of semantic approximation and optimization techniques. In our opinion, this quick turnaround is the key to a successful industrialization of the technology and its widespread use.

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, June 7–14 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. 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.
13. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
14. Kestrel. *Specware System and documentation*, 2003. http://www.specware.org/.
15. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.
16. S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.
17. 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.
18. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis using 3-valued logic. In *Proceedings of Symposium on Principles of Programming Languages*, 1999.
19. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Computational Complexity*, pages 136–150, 1996.
20. 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.
21. 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.
22. 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.

23. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.
24. 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.
25. 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.
26. 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.