Towards the Integration of Symbolic and Numerical Static Analysis

Arnaud Venet

Kestrel Technology
4984 El Camino Real #230
Los Altos, CA 94022
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 [16,6,11,2,4]. 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 [2,4] does not abstract 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 [23,3]; the second one comes from the controller of a science payload developed at NASA for the International Space Station [22].

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 [20,19,21,22,23] 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 [23,3]. 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 have little commonalities. Since their architecture is quite complex and is tailored towards a specific class of applications, 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. Section 3 gives an informal description of the technical challenges of designing such analyses. 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.

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 [8,9] 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 [10]. 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 [12]. 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 [17]. 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 [17]. This allowed us to construct a pointer analysis of similar power for dynamically typed languages like Java [20], as well as a communication analysis for systems of concurrent processes based on the π-calculus [18]. 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 [21] and a considerably simpler communication analysis for the π-calculus [19]. 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 [22]. 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 [22].

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 [23]. 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 [16], 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 Technical Challenges

Anyone reading the literature on mixed symbolic and numerical static analysis will likely be struck by the conceptual complexity of the algorithms. However sophisticated it may be, a pointer analysis relies on few simple concepts that can be clearly stated (unification-based/inclusion-based, flow-sensitive/flow-insensitive, store-based/storeless, etc.). Similarly, a numerical static analysis framework is acutely described by the family of geometric shapes used to approximate point clouds: linear affine spaces (linear equalities), higher-dimensional rectangles (intervals), convex polyhedra (linear inequalities), etc. In both cases, the technical description essentially consists of carrying out the formalization of these basic concepts in details. In a mixed symbolic and numerical analysis, the complexity stems from the association between symbolic values (object addres, channel name, etc.) and numerical components (index in an array, loop iteration counter, index in a list, etc.). This association seems completely arbitrary and is not supported by any general underlying concept. One could naturally question the need of building such complex analyses and suggest instead making pointer and numerical analyses cooperate. Such an approach has become quite popular in the theorem proving community [7]. In some sense, this is exactly what a mixed symbolic-numerical static analysis does. However, there is no *canonical* way of combining the symbolic and numerical components so that the resulting analysis scales well and gives precise results. The rest of this section will be devoted to discussing these points.

As a basis for our discussion we consider the pointer analysis of C with a byte-based representation of memory blocks [22]. For the sake of clarity, we ignore dynamic memory allocation. In such a model, a pointer is given by a symbolic address a and an offset o in bytes from the beginning of the block referenced by a. This low-level representation greatly simplifies the analysis of union types and casts. A program configuration is made of an environment E and a heap H. The environment maps scalar variables i to integer values and pointers p to pairs of address and offset as follows:

$$E = \langle i_1 \mapsto n_1, \ldots, i_k \mapsto n_k, p_1 \mapsto (a_1, o_1), \ldots, p_m \mapsto (a_m, o_m)\rangle$$

The heap H is a graph (V, E), such that the set of vertices V contains addresses and an edge (a_s, o_s, a_t, o_t) of E denotes the existence of a pointer in memory block a_s at offset o_s referencing the memory cell in a_t located at offset o_t. For simplicity, we ignore scalar data stored in the heap. The role of a pointer analysis consists of abstracting sets of configurations (E, H).

First, we need to recall briefly how numerical abstract interpretation works. Given a finite set of integer-valued variables $U = \{v_1, \ldots, v_n\}$, a numerical abstract domain $\mathcal{N}U$ provides a computable approximation of point clouds in $\wp(\mathbb{Z}^U)$. The precision of the approximation is determined by the class of numerical relationships between variables of U that can be expressed in the abstract numerical domain. The domain of intervals is one of the least expressive domains, since no relationship between variables can be expressed. The domain of convex polyhedra [5] can describe arbitray systems of linear inequalities over variables of U, Karr's domain [12] can describe systems of linear equalities whereas the domain of difference-bound matrices [15] can only express inequalities of the form $x - y \leq c$. Expressiveness comes with a price, and the maximum number of variables that can be handled by an abstract numerical domain in practice ranges from about a dozen for convex polyhedra, to a few tens for difference-bound matrices, to tens of thousands for intervals.

In the case of pointer analysis, it appears in many practical situations that the approximation of environments and heaps can be carried out independenlty without incurring a significant loss of accuracy. For example, in embedded applications the pointer structure in global memory is typically set up during the initialization phase and remains stable at mission time [3]. One can use a graph-based abstraction such that an edge (a_s, o_s, a_t, o_t) is abstracted by a triple (a_s, a_t, ν), where $\nu \in \mathcal{N}\{o_s, o_t\}$ describes numerical relationships between the source and target offsets. This clear separation between environment and heap abstractions enables the use of modified versions of existing pointer algorithms [22], so that the computation of numerical invariants can be carried out independently from that of the points-to graph [22], or both can be interleaved [23]. At this level, the numerical and symbolic algorithms do really work in cooperation. Experiments on aerospace code have shown that simple choices for the abstract domain yield good results in practice [23,22].

As for environments, there is a very natural abstraction. Given an environment structure with k scalar variables and m pointer variables as described above, we have to abstract a set of $k + m$ integer variables (scalar variables plus offsets) and m symbolic variables. Assuming that the set A of addresses is finite, we are left with approximating an element of $\wp(A^m \times \mathbb{Z}^{m+k})$. This is isomorphically equivalent to approximating a mapping of $A^m \to \wp(\mathbb{Z}^{m+k})$. If we denote by N the set of numerical variables $\{i_1, \ldots, i_k, o_1, \ldots, o_m\}$, a natural abstract domain for the analysis is $A^m \to \mathcal{N}N$. This can be regarded as a canonical abstraction of a set of environments. However, it poses two major scalability issues:

– since A can be large, the mapping from A^m may cause a combinatorial explosion,
– the size of N may preclude the use of expressive numerical abstract domains.

This is a major problem, since we do need epxressive numerical abstract domains in order to infer e.g., that `i` $\leq$ `n` in the first example of Fig. 1. The problem is actually much deeper than achieving good precision, since computing numerical relationships between variables is precisely what enables us to keep the construction of the abstract heap as a separate phase [23,22].

A solution to this problem consists of breaking down the "big" abstract domain $A^m \to \mathcal{N}N$ into a collection of smaller, more manageable domains. This essentially amounts to grouping variables into small clusters and applying the same reasoning for each cluster individually. For example, one can group variables in clusters of the form $C_j = \{a_j, o_j, i_1, \ldots, i_k\}$. Each cluster C_j yields an abstract domain $A \to \mathcal{N}\{o_j, i_1, \ldots, i_k\}$. With this collection of abstract domains the analysis can infer relationships between each pointer offset and all the integer variables. These relationships are parameterized by the address of the memory block that may be referenced by the pointer. However, relationships between pointers that are simultaneously manipulated within a loop (like in the statement `*p++ = *q++`) are lost. This is an example of clustering using a static criterion. Clustering can also be performed dynamically during the analysis as done in C Global Surveyor [23]. Using variable clusters is what makes the design of the analysis complex and intricate, mostly because a semantic operation on a variable does not only affect this variable but also all the clusters in which it appears. The choice of a particular clustering sets a certain tradeoff between precision and efficiency, and depends on the characteristics of the program or family of programs considered. This implies that clustering has to be empirically validated on the target applications.

In conclusion, combining symbolic and numerical analyses allows us to achieve the high level of precision required to perform automatic verification of pointer-intensive programs. The architecture of these static analyzers enables the integration of numerical and symbolic algorithms that work in cooperation. However, in order to achieve scalability we must introduce a layer of complexity in the structure of the analyzer. One may question the relevance of this approach to program verification if a complex analyzer has to be constructed for each application. The situation is aggravated by the absence of any methodology for designing these analyzers. We will present perspectives for addressing these issues in the next section.

4 Perspectives

4.1 Towards a General Framework

The clustering technique exposed in the previous section basically consists of *covering* the abstract domain $A^m \to \mathcal{N}N$ using smaller domains $A^n \to \mathcal{N}M$ where $n \leq m$ and $M \subseteq N$. The semantic operations on this collection of abstract domains are carried out by "patching" elements belonging to overlapping clusters, without ever using the larger domain $A^m \to \mathcal{N}N$. This computational scheme bears a striking analogy with the techniques provided by the theory of fiber bundles and sheaves [14] for studying global properties of complex topological spaces that possess a regular structure locally. Actually, there is more than a simple analogy, and we are currently investigating the transposition of these topological constructs into the theory of Abstract Interpretation. We have already come up with a framework that is able to express existing mixed symbolic-numerical pointer analyses. The major benefit of the framework is that it provides a systematic construction of the semantic operations from an arbitrary clustering of variables. This means that the complexity of designing such analyses can be encapsulated in a small set of semantic operators that are used to systematically derive the analysis algorithms from a simple specification of the concrete semantics.

4.2 Automated Generation of Static Analyzers

The implementation of this framework is underway, based on the formal specification environment SpecWare [13]. Our first objective is to be able to reimplement existing analyses with much lesser effort. 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 (variable clustering) 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 [23] 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. In our opinion, this quick turnaround is the key to a successful industrialization of the technology and its widespread use.

4.3 Spectrum of Applications

If pointer or communication analyses were the only applications of a mixed symbolic-numerical static analysis framework, however important they are, there would not be much interest in pursuing long-term research in this direction. We believe that this class of static analysis has a broader range of applications for it basically permits to attach numerical invariants to discrete structures. Numerical computations form the basis of the control structure for the vast majority of applications, and numerical properties naturally appear in resource analysis (cpu, memory, network) and the statement of some security properties (number of times a cryptographic object is used, number of channels open simultaneously, etc.). In order to analyze real code the analysis must be able to attach numerical properties to objects of the program, for example:

– the number of bytes transmitted through a channel to the channel descriptor,
– the number of times cryptographic data are used to the corresponding objects in memory.

The properties to analyze should be expressed in the semantics of the program and interpreted by the static analyzer generator with little intervention of the user. It means that the property to be analyzed should be part of the specification that forms the input of the static analyzer generator. This opens the way to the automated generation of custom static analyzers that verify properties defined by users who are not experts in the field, like software developers.

References

1. Andersen, L.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), June 7–14, 2003, pp. 196–207. ACM Press, New York (2003)
3. Brat, G., Venet, A.: Precise and scalable static program analysis of NASA flight software. In: Proceedings of the 2005 IEEE Aerospace Conference (2005)
4. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the Fifth Conference on Principles of Programming Languages, ACM Press, New York (1978)
6. Das, M.: Unification-based pointer analysis with directional assignments. ACM SIGPLAN Notices 35(5), 35–46 (2000)
7. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N.: Integrating verification components. In: Verified Software: Theories, Tools, Experiments, Zrich, Switzerland (October, 2005)
8. Deutsch, A.: 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, pp. 2–13. IEEE Computer Society Press, Los Alamitos (1992)
9. Deutsch, A.: Interprocedural alias analysis for pointers: beyond k-limiting. In: ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, ACM Press, New York (1994)

10. Eilenberg, S.: Automata, Languages and Machines, vol. A. Academic Press, London (1974)
11. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 254–263 (2001)
12. Karr, M.: Affine relationships among variables of a program. Acta Informatica, 133–151 (1976)
13. Kestrel,: Specware System and documentation (2003), `http://www.specware.org/`
14. Mac Lane, S., Moerdijk, I.: Sheaves in Geometry and Logic. Springer, Heidelberg (1992)
15. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
16. Steensgaard, B.: Points-to analysis by type inference of programs with structures and unions. In: Computational Complexity, pp. 136–150 (1996)
17. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 266–382. Springer, Heidelberg (1996)
18. Venet, A.: Abstract interpretation of the π-calculus. In: Dam, M. (ed.) LOMAPS-WS 1996. LNCS, vol. 1192, pp. 51–75. Springer, Heidelberg (1997)
19. Venet, A.: Automatic determination of communication topologies in mobile systems. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 152–167. Springer, Heidelberg (1998)
20. Venet, A.: Automatic analysis of pointer aliasing for untyped programs. Science of Computer Programming 35(2), 223–248 (1999)
21. Venet, A.: Nonuniform alias analysis of recursive data structures and arrays. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 36–51. Springer, Heidelberg (2002)
22. Venet, A.: A scalable nonuniform pointer analysis for embedded programs. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 149–164. Springer, Heidelberg (2004)
23. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: Proceedings of the International Conference on Programming Language Design and Implementation, pp. 231–242 (2004)

A Discussion on Arnaud Venet's Presentation

Willem-Paul de Roever

I can try to recall my algebraic topology course. What I recall from this is that you have to get change of groups to get homology theory. So, this is what homology, as I knew, is about: a classification of topological spaces. So, the mathematical analysis technique is that of groups, where you compute homology groups. What is the analogy of a homology group here?

Arnaud Venet

The use of the term "homology" stems from the striking analogy between the algebraic structures underlying this class of static analysis and those appearing in algebraic topology in the context of homology theory, or more exactly cohomology. At this point this is nothing more than an analogy, but the connection between these two fields are intriguing enough to motivate a deeper investigation. In standard algebraic topology, the cohomology groups define an equivalence relation on cochains that define the same covering. In our case, the analogue of a cochain is the definition of a semantic transformer as the gluing of numerical relations over several sets of symbolic variables that overlap. We are working on pushing this analogy further toward a more formal connection between both fields.