Numerical static analysis with Soot

Gianluca Amato   Simone Di Nardo Di Maio   Francesca Scozzari
Università di Chieti-Pescara - Italy
{gamato, simone.dinardo, fscozzari}@unich.it

Abstract

Numerical static analysis computes an approximation of all the possible values that a numeric variable may assume, in any execution of the program. Many numerical static analyses have been proposed exploiting the theory of abstract interpretation, which is a general framework for designing provably correct program analysis. The two main problems in analyzing numerical properties are: choosing the right level of abstraction (the abstract domain) and developing an efficient iteration strategy which computes the analysis result guaranteeing termination and soundness.

In this paper, we report on our prototype implementation of a Java bytecode static analyzer for numerical properties. It has been developed exploiting `Soot` bytecode abstractions, existing libraries for numerical abstract domains, and the iteration strategies commonly used in the abstract interpretation community. We show pros and cons of using `Soot`, and discuss the main differences between our analyzer and the `Soot` static analysis framework.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis

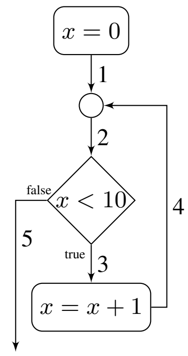
General Terms Design, implementation, static analysis.

Keywords Abstract interpretation, bytecode, numerical domains.

1. Introduction

Static analysis determines, at compile-time, properties about the run-time behavior of programs, in order to verify, debug and optimize the code. Abstract interpretation [14, 15] is a general theory for defining static analyses starting from the property of interest (the so-called *abstract domain*), and for formally proving their correctness.

The basic idea of abstract interpretation is that a static analysis can be derived from the (concrete) semantics of a program. We assume that the semantics of a program P can be computed as the least fixed point of a semantic function $f : C \to C$, where C is a concrete domain. In general, the semantic function is given compositionally starting from a set of basic operators which depends on the kind of programming language under analysis. Typical operators for imperative programs include assignment, test and merge, which are used in the semantics for treating the basic statements of variable assignments, conditionals and loops.



$$
\begin{aligned}
y_1 &= [0, 0]\\
y_2 &= y_1 \vee y_4\\
y_3 &= y_2 \wedge [-\infty, 9]\\
y_4 &= y_3 + [1, 1]\\
y_5 &= y_2 \wedge [10, \infty]
\end{aligned}
$$

Figure 1. A control flow graph and the corresponding system of equations in the interval abstract domain.

In the abstract interpretation theory, a static analysis is viewed as an abstract semantics, which can be directly obtained from the concrete one by substituting the concrete domain C with an abstract domain A, representing the program properties we want to analyze, and all the concrete operators with corresponding abstract operators.

In practice, at the end of this formalization process, we get a system of equations not dissimilar from any other data-flow analysis. Each variable in the equations corresponds to a program point, and each equation describes the effect of an instruction or a basic block on the program properties. The least fixpoint of this equation system is a safe approximation of the concrete semantics of the program.

In this work we are mainly interested in numerical properties. A numerical property on a program variable x gives an (over) approximation of the possible values that the variable x may assume in a specific program point, during any execution of the program. Numerical properties are typically expressed by means of geometrical shapes. For instance, in the interval abstract domain [13], each abstract object maps a program variable to a (possibly unbounded) interval, such as $x \in [3, 7]$. Consider, for instance, the simple program:

```
x = 0
while (x<10)
    x = x + 1
```

whose corresponding control flow graph and system of equations are depicted in Figure 1. Each variable y_i in the system corresponds to the edge in the graph labeled by i, which in turn is a relevant program point.

The abstract semantics of the example program is simply obtained as the least solution of the corresponding system of equations solved over the abstract domain of intervals, where the operators $\vee$ and $\wedge$ represent convex hull and intersection of intervals, while $+$ is the (pointwise) sum.

Analyses over the abstract domain of intervals are quite simple, efficient, but not very precise. Other numerical domains, encoding more expressive properties, are the polyhedra [16], octagon [21] and parallelotope [2] domains. Of course, more expressive abstract domains lead to higher complexity from the computational point of view.

In the polyhedra domain, each abstract object is a polyhedron described by a system of linear equations $l \leq Ax \leq u$, where x is the vector of program variables, A is the coefficient matrix and l, u are vectors. A parallelotope is a polyhedron whose matrix A is invertible. The octagon domain is very similar to the polyhedra domain but the coefficient matrix A is fixed; the linear equations are of the form $\pm x_1 \pm x_2 \leq c$, where x_1 and x_2 are program variables and c is a constant.

All numerical abstract domains should implement a common set of operations such as assignment, convex hull (which corresponds to the merge operation in `Soot`), intersection, projection, and so on. Usually the equations are solved iteratively, through successive approximations to determine a fixed point. Since most numerical abstract domains contain infinite ascending chains, we need to introduce an approximation to be able to compute a fixed point.

The most common technique is to use a *widening* [13]. It is an operator that allows to predict the fixed point by analyzing the sequence of approximations calculated in previous iterations of the analysis. It guarantees termination of the analysis, but may introduce a loss of precision.

In order to partially recover precision, it is almost mandatory to perform a two-phases analysis: an ascending phase, using widening, which computes a rough over-approximation, and a descending phase, where widening is replaced by a new operator called *narrowing*, which refines the previous result.

By following the abstract interpretation theory and exploiting `Soot` [23], we are implementing analyses of numerical properties of Java bytecode inside our analyzer `Jandom`.

2. `Jandom`

`Jandom` is an abstract interpretation based static analyzer written in Scala, derived from our former project `Random` [3, 6], which implemented template parallelotopes [1, 5, 7, 8]. At the moment, `Jandom` supports intra-procedural static analysis of numerical properties for a simple imperative language with a C-like syntax. It has preliminary support for symbolic transition systems of the kind used in the FASTer[1] model checker [10] and for Java bytecode. `Jandom` is freely available online[2] on GitHub. In the long run, `Jandom` aims at becoming a general framework for the abstract interpretation community to ease the implementation of new analysis strategies and the test of new abstract domains.

The support for the analysis of Java bytecode is at a preliminary stage. We support a small set of instructions which allow to analyze only very simple methods. Supporting the full bytecode would not be difficult, but time consuming. Therefore, we prefer to explore different implementation models and technologies before committing to a definitive solution.

For manipulating the Java bytecode, we have first tried `ASM`[3], which is a fast library with pretty good documentation. We also evaluated the use of `BCEL`[4], but we preferred `ASM` since it seems to be better maintained.

[1] `http://tapas.labri.fr/trac/wiki/FASTer/`

[2] `https://github.com/jandom-devel/Jandom/`

[3] `http://asm.ow2.org/`

[4] `http://commons.apache.org/proper/commons-bcel/`

In our experience, `Soot` is not very common in the abstract interpretation community, so we came to it only later. However, `Soot` has a much broader scope than `ASM`, and it has been extensively used for the static analysis of Java. Therefore, we expected that using `Soot` would give us important benefits and accelerate development.

This paper is a report on our experience with the bytecode analyzer in `Jandom`, and the use of `Soot` in its development. We will try to explain what parts of `Soot` we used, the ones we did not use, the ones we plan to use in the future. It is important to note that our aim is to use `Soot` as a library for implementing numerical static analyses. We do not consider here the problem of integrating these analyses in the `Soot` framework itself. The interested reader can find the code described here in the branch `soap2013` of the GitHub repository of `Jandom`.

2.1 Architecture of `Jandom`

We have designed `Jandom` having in mind a strongly layered structure, as depicted in Figure 2. Some of these layers are not so cleanly separated in the real code as they are in this description. Nonetheless, this is the model to which we are aiming.

2.2 Numerical abstract domains

In the lower layer we find the numerical abstract domains, which encode properties of numerical variables. Although there are a few numerical domains natively implemented in `Jandom`, most of them are part of the Parma Polyhedra Library (`PPL` for short) [9]. Another well known library for numerical domains is `APRON` [17], which we plan to integrate in the future. To accommodate the use of native, `PPL` and `APRON` based domains, we have designed a suitable common interface, called `NumericalProperty`. All the native domains directly implement this interface, while `PPL` domains are appropriately wrapped. This interface allowed us to develop a truly parametric analyzer, which can be easily plugged with new, numerical abstract domains.

We believe that the ability to exploit existing libraries is fundamental when designing a new analyzer, so that we carefully studied the problem of best wrapping `PPL` abstract domains into our common interface, which was not an easy task. In fact, while all the `PPL` domains have almost identical method signatures, they do not implement any common Java interface, and directly descend from the `Object` class. This is the heritage of the fact that `PPL` is developed in C++, where templates may be used to achieve generic programming. The Java bindings came later and, unfortunately, did not try to recover some of the flexibility of templates through inheritance. This makes it difficult to write a generic wrapper for all the `PPL` domains. At the moment, we use three kind of wrappers:

- ad-hoc wrappers for the most common numerical domains;
- a generic wrapper based on reflection;
- a generic wrapper based on Scala macros.

Ad-hoc wrappers are the simplest one, but different wrappers are required for different domains. The reflection based wrapper is quite convenient, but suffers from the performance penalty of using reflection. On the contrary, the wrapper based on macros has the same speed of the ad-hoc wrappers, but it is generic for all the domains: the Scala compiler generates a different binary for each numerical domain at compile time, similarly to C++ templates.

Although the macro based wrapper seems to be the best choice, its use is annoying in the developing process. Due to the limits in the Scala compiler and build tools, classes using macros should belong to different projects than classes defining macros. For this reason, we use the reflection based wrappers during the daily development.

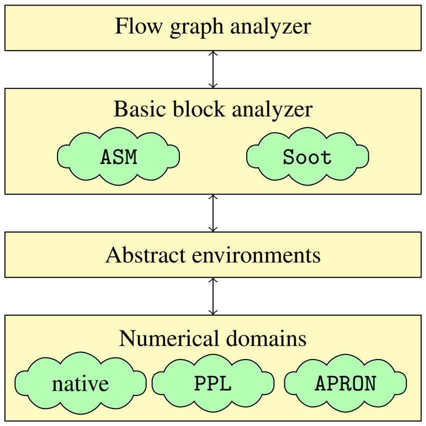


Figure 2. Layered architecture of `Jandom`

Another possibility we are exploring is to use `Soot` or `ASM` abilities of manipulating bytecode to generate ad-hoc wrappers at runtime. This would bring all the advantages of macro based wrappers, together with a reduction of the size of compiled code and a simplification in the management of the project. On the other side, the implementation of this dynamic wrapper is more difficult, especially because the correspondence between source code and bytecode is not so easy in Scala as it is in Java.

Finally, this layer also contains some domain combinators, i.e., methods to get more precise domains from the basic ones. Thanks to the powerful type system of Scala, the numerical domain API is completely type safe.

It is worth noting that, at the moment, numerical domains do not take into consideration overflow and underflow of machine integers and floats. There are standard methods to handle them [20], which we plan to integrate in the near future.

2.3 Abstract environments

Numerical domains only handle numerical variables and their relationships, but the program state in the JVM is much more complex: for instance, there are objects in the heap and references to objects. In addition, in the *Baf* representation we also have a stack to take into consideration. The *Abstract environments* layer allows to implement (different) abstractions of the full program state. It is parametric w.r.t. a numerical domain, which is used for the abstraction of numerical variables in the stack, frame and heap.

We have implemented two abstract environments: one is used for the analysis with *Jimple*, the other one for the analyses with *Baf* and `ASM`. At the moment, they both ignore the heap and everything which is not numeric. In the future, we plan to introduce new domain hierarchies for the analysis of heap and objects, and make them additional parameters of the abstract program environment.

As an example, consider a generic abstract state for *Baf*, as implemented in the class `JVMEnvDynFrame`. It is a triple $\langle f, s, p\rangle$ where p is a numerical property (such as $v_0 + v_1 = 1$), f is an array which maps frame positions to variables in p, and s is a stack of integers which has the same goal as f but for the stack. For example, the state $\langle [0, -1], \langle 1, -1, -1\rangle, v_0 + v_1 = 1\rangle$ means that:

- the frame has two positions. The first position corresponds to the variable v_0 in p, while the value -1 means that the second position is unused or contains a non-numerical value;
- the stack has currently three elements. The top element corresponds to the variable v_1 in p, the other two elements contain non numerical values;
- the first frame position and the top position of the stack are subject to the condition $v_0 + v_1 = 1$.

This choice of tracking frame and stack variables separately from the numerical property p has many advantages: the dimension of p (the number of variables in the associated space) varies dynamically and is generally much lower than the total number of variables in the heap and stack. Since most numerical domains have cubic (or worst, even super exponential) complexity on the dimension of p, we want to keep this value as low as possible.

An alternative choice is to only use p (without the f and s components) and a fixed mapping from numeric dimensions to positions in the stack and frame. For example, the same property above could be represented as $v_0 + v_4 = 1$, where v_0 and v_1 are the variables associated to the frame and $v_2, \dots, v_4$ those associated to the stack. With this solution, implemented in the `JVMEnvFixedFrame` class, p has dimension $n + m$ where n is the number of locals and m is the size of the stack. This solution has the advantage to be simpler than the previous one, but we expect it to be slower for methods with many non-numerical variables.

The abstract environment for *Jimple* is much simpler since there is no stack involved and the correspondence between the local variables and the variables in the numerical properties is fixed for the entire method.

2.4 Basic block analyzer

Our definition of basic block is somewhat different from the standard one: it is a maximal sequence of instructions such that none but the first one may be target of a jump. Therefore, both single instructions and standard basic blocks are instances of our definition, but we do not force the creation of new blocks after every jump instruction. Bigger blocks allow to reduce the number of intermediate program states we need to record to accomplish the analysis.

It turns out that a basic block may have many outgoing edges: one fall-through edge, which is followed when the execution reach the end of the block, and many edges corresponding to the targets of jumps. The result of the analysis of a block is a sequence of pairs $\langle$*outgoing edge*, *abstract env*$\rangle$.

The basic block analyzer is strictly connected to the (abstract) language we want to analyze. Therefore, we have different basic block analyzers for `ASM`, *Baf* and *Jimple*.

2.5 Flow graph analyzer

The flow graph analyzer builds a full intra-procedural analysis from a directed graph of basic blocks. At the moment, it implements a worklist based strategy similar to the one provided by `ForwardBranchedFlowAnalysis`, but it directly supports ascending and descending phases and some advanced widenings.

A crucial point in abstract interpretation based analysis is the ability to determine an admissible set of widening points, i.e., a set of program points where widening should be used instead of merge to ensure the termination of the analysis. Fundamentally, a set of widening points is admissible if every cycle in the control flow graph passes for at least a widening point. Determining a good set of widening points is easy with `Soot`. We can use the `(Slow)PseudoTopologicalOrderer`, and take as widening points those program points which are the target of some retreating edges.

Figure 3 shows the result of the *Baf* analyzer for the flow graph in Figure 1.

3. `Jandom` and `Soot`

Now that we have outlined the architecture of `Jandom`, we go into more depth on the relationship between `Jandom` and `Soot`, we

```
static void loop()
    {
        word i0;

/* Frame: <-1> Stack: <> Property: [  ] */
        push 0;
        store.i i0;

      label0:
/* Frame: <0> Stack: <> Property: [ 0 <= v0 <= 10 ] */
        load.i i0;
        push 10;
        ifcmpge.i label1;

        inc.i i0 1;
        goto label0;

      label1:
/* Frame: <0> Stack: <> Property: [ v0 = 10 ] */
        return;
    }
```

Figure 3. Result of the *Baf* analysis for the flow graph in Figure 1.

discuss some of our implementation choices and we outline plans for future work related to Soot.

3.1 The Soot analysis framework

Before implementing our *Flow graph analyzer*, we evaluated whether to use the Soot analysis framework directly, but we decided against it. The main reason is that we are using Soot as a library for a generic abstract interpretation based analyzer, which we would like to use to test different iteration strategies for several target languages, such as imperative, object-oriented and transition systems. With this in mind, and given the amount of work already done in Jandom to accomplish this goal, we concluded that using the Soot framework was not going to give us any real benefit. We discuss here in detail why this is the case.

First of all, the only viable base class for our analyzer is ForwardBranchedFlowAnalysis, since we need to keep separate numerical properties at different branches of a conditional instruction. This class implements a highly optimized but straightforward worklist based analysis. However, an abstract interpretation based static analysis often requires more complex iteration strategies to achieve adequate precision. At least, a couple of phases are needed: an ascending phase where an over approximation of the required solution is built using widening, and a descending phase where the result of the first phase is improved. These phases could have been implemented with two different ForwardBranchedFlowAnalysis in cascade, but other more complex iteration strategies (with multiple interleaving phases) would have not been possible without overriding the doAnalysis method, which is tantamount to rewrite the analyzer from scratch. This is the case, for example, of the basic recursive [11] strategy or the more advanced localized [4] iteration strategies.

We believe that the analysis algorithm in Soot could be generalized in order to support different iteration strategies. The best approach would be to design a generic framework to solve fixpoint equations. In the abstract interpretation community, the best known tool for this task is the *Fixpoint* library[5], which is written in OCaml but could be ported to Java.

Since we do not use the Soot analysis framework, we have no reason to use the FlowSet interface either. This is too limited to be used for numerical properties, which need a lot of primitive operations such as assignment of linear expressions, projection over a subset of variables, and intersection with an half-plane. In Jandom we use the NumericalProperty abstract class as the base for all the numerical properties. We could have made NumericalProperty descend from FlowSet, but at the moment this is not possible since we have implemented numerical properties as immutable objects.

[5] http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/fixpoint/

3.2 Basic blocks

The Block class in Soot is able to represent large basic blocks. However, in order to really generate blocks larger than the standard ones, we had to provide our subclasses of BlockGraph. Although BlockGraph has a method computeLeader which should be used for such a purpose, overriding it was not enough, since the buildBlocks method assumes that every jump instruction is the tail of a block. Therefore, we had to override both methods. We believe that Soot could be retrofitted with our new implementation of buildBlocks.

Some operations in Jandom may be simplified by the assumption that when a node (either a Unit or a Block) has more than one successor, the first one is the fall-through node, if it exists. Although this seems to hold in the current implementation of Soot, it is not documented anywhere. We think that this is a useful property that should be made explicit.

Note that representing larger blocks is completely optional and just for the sake of optimization, since the upper layer of Jandom also works with block graphs built by the standard Soot libraries.

3.3 *Baf* vs *Jimple* vs *Grimp*

We have investigated both *Baf* and *Jimple* intermediate representations. The common expectation is that *Jimple* is easier to analyze, since it is more high-level and has fewer statements. However, we are not really sure that this makes a big difference for the kind of analysis we are interested in.

Most of the complexity of the bytecode w.r.t. the 3-address code used in *Jimple* is due to the big number of arithmetic and conditional instructions. The standard analyses in Soot do not care about arithmetic properties. Hence, abstracting all these instructions into an AssignStmt unit may actually simplify the code. In our case we need to explicitly handle arithmetic instructions. Using *Jimple* instead of *Baf* just means that we need to carefully inspect the right hand side of assignments. However, expressions (i.e., objects of class Value) have a more complex structure than bytecode.

Another reason for the big number of instructions in the bytecode is that some of them have several variants. For example, the *aload* instruction has variants aload_0, aload_1, aload_2, aload_3 and aload. However *Baf* (and also ASM) abstracts away from these differences. For example, all the load instructions above are collapsed in the single LoadInst instruction.

On the other side, *Jimple* has still some advantages over *Baf*, even for our analyses: it abstracts away from the frame and stack of the JVM, so we only need to deal with variables (objects implementing Local).

The *Grimp* intermediate representation is not generally used for static analysis. It is similar to *Jimple* but expressions are not linearized and may be quite complex. However, for the analysis of numerical properties, this representation may help in improving precision. It turns out that, in many cases, we may analyze the effect of a complex assignment with greater precision if it is considered all at once.

Consider for example the assignment `z = z + x + y`. Given the precondition $z = w \wedge x + y = 0$, with the octagon abstract domain the analyzer infers that, after the assignment, $z = w \wedge x + y = 0$ still holds. But if the assignment is decomposed as `z = z + x` and `z = z + y`, after `z = z + x` any information regarding z is lost, since octagon cannot represent the correct invariant which is $w = z - x$.

In conclusion, we think that performing analyses on *Grimp* may be more precise than on *Jimple*, and not terribly more difficult. As observed by one of the referees, for some numerical domains *Grimp* could also improve performance, since less abstract operations are performed, in particular for domains implemented in APRON or PPL, due to the overhead of calling native methods.

Although we have not yet tried *Grimp* in Jandom, we plan to transform the *Jimple* analyzer into a *Grimp* analyzer, while keeping the *Baf* analyzer.

3.4 The tag system

The result of our analyzer is a map, which we call annotation, from program points (i.e. `Units`) to abstract environments. At the moment, the map is implemented through a Scala `HashMap`. This implies the need to access hash maps and compute hash functions each time we read or write annotations (and we do it continuously in the analysis engine).

A better solution would be to link annotations directly to the corresponding program point. To this aim, the Soot tag system might be used. However, it requires a linear search for the tag name each time we access a tag, so it is not going to improve performance very much. It would be convenient to modify the tag system adding the possibility to access tags in an indexed manner, using an `ArrayList` as a backend instead of a `List`. This should allow constant access time to an annotation if we know its index.

3.5 Planned use of Soot in the future

Other parts of Soot we plan to explore in the future are:

- *Dava*: although *Dava* is presented as a decompiler for Java, we think that performing a structured analysis on the AST of a Java program may sometimes be more convenient than analyzing the unstructured bytecode. See [18] for a discussion on the benefits of analyzing bytecode vs program source code. However, to the best of our knowledge it is not possible to generate a *Dava* AST directly from the Java source code, and this limits its usefulness for us. We could use one of the many available Java parsers, but it would be great to have this integrated in Soot.
- Eclipse plugin: using the Soot tag system, we plan to interface Jandom with the Eclipse plugin.

3.6 Documentation

Not everything in the experience with Soot was pleasant. One aspect which can be improved is documentation. One of the main drawbacks of the current documentation is the lack of a real user manual. There are many tutorials available online, but nothing with a thorough treatment. On the contrary, ASM has a very detailed guide [12] which allows the reader to become very competent with the API quite easily. It is also true that Soot is much more complex and powerful than ASM, and therefore it is more difficult to describe.

Also, the Javadoc could be improved. For example, consider the class `PseudoTopologicalOrderer`. There is no reference to what a pseudo topological order is. It turns out that the algorithm essentially computes a depth-first visit of a graph, and reports the order of visit. It is the same algorithm suggested by Burdoncle in [11] to efficiently find a *weak topological order*. Although this can be inferred by the source code, the user cannot be sure this is the intended behavior, and not an artifact of the current implementation that may change in the future.

4. Related work

Although Soot is not commonly used for the analysis of numerical properties, [22] describes an analysis to remove bound checks in Java which is particularly relevant to our aims. Bound check elimination is based on an intra-procedural numerical analysis called *variable constraint analysis* (VCA) coupled with auxiliary inter-procedural analysis to improve precision. VCA is not dissimilar from a classic abstract interpretation based analysis. *Variables constraint graphs*, which are used to represent linear constraints among variables, are an alternative representation of an abstract domain known in the abstract interpretation literature as *difference bound matrices* [19], a precursor of the Octagon domain.

It seems that the authors of [22] had to solve some problems similar to the ones we found in Jandom. For example, they do not use the standard Soot analysis engine, since they want more control on the order in which semantic equations are solved. Although their aim is different from ours and their analysis is optimized for a particular purpose, some ideas might be implemented in Jandom since appear to be of general usefulness, for example the strategy to distinguish between loop body and loop exit.

5. Conclusions

Have we any benefit to using Soot in Jandom instead of a simple bytecode library such as ASM? The immediate answer to this question is: yes, but not as much as we would. The point is that Soot is a complex framework and to get all the benefits we should embrace it completely. This is at the moment not possible since Jandom also supports ASM and languages other than Java bytecode. However, we should evaluate whether ditching the other targets (or compiling them to bytecode) and making Jandom a pure Java bytecode analyzer.

We also expect Soot to be much more useful once we implement inter-procedural analysis in Jandom. For example, the ability to browse all the classes of the `Scene` and to compute call graphs will be of great help.

Nonetheless, there are some improvements to Soot which could greatly help in spreading the use of this library in the abstract interpretation community. The most important one is probably, as we said before, integrating in Soot a more sophisticated data-flow equation solver such as *Fixpoint*.

References

[1] G. Amato and F. Scozzari. Observational completeness on abstract interpretation. *Fundamenta Informaticae*, 106(2–4):149–173, 2011. doi: 10.3233/FI-2011-381.

[2] G. Amato and F. Scozzari. The abstract domain of parallelotopes. In J. Midtgaard and M. Might, editors, *Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012*, volume 287 of *Electronic Notes in Theoretical Computer Science*, pages 17–28. Elsevier, 2012. doi: 10.1016/j.entcs.2012.09.003.

[3] G. Amato and F. Scozzari. Random: R-based analyzer for numerical domains. In N. Bjrner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 375–382. Springer, 2012. doi: 10.1007/978-3-642-28717-6_29.

[4] G. Amato and F. Scozzari. Localizing widening and narrowing. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2013.

[5] G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626–4671, 2009. doi: 10.1016/j.tcs.2009.07.038.

[6] G. Amato, M. Parton, and F. Scozzari. A tool which mines partial execution traces to improve static analysis. In H. Barringer and *et al.*, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 475–479. Springer, 2010. doi: 10.1007/978-3-642-16612-9_37.

[7] G. Amato, M. Parton, and F. Scozzari. Deriving numerical abstract domains via principal component analysis. In R. Cousot and M. Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2010. doi: 10.1007/978-3-642-15769-1_9.

[8] G. Amato, M. Parton, and F. Scozzari. Discovering invariants via simple component analysis. *Journal of Symbolic Computation*, 47 (12):1533–1560, 2012. doi: 10.1016/j.jsc.2011.12.052.

[9] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. doi: 10.1016/j.scico.2007.08.001.

[10] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008. doi: 10.1007/s10009-008-0064-3.

[11] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993. doi: 10.1007/BFb0039704.

[12] E. Bruneton. *ASM 4.0 – A Java bytecode engineering library*, 2011. URL `http://download.forge.objectweb.org/asm/asm4-guide.pdf`. Last accessed 2013/05/18.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977. doi: 10.1145/512950.512973.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM Press, 1979. doi: 10.1145/567752.567778.

[16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–97. ACM Press, 1978. doi: 10.1145/512760.512770.

[17] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009. doi: 10.1007/978-3-642-02658-4_52.

[18] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In L. J. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008. doi: 10.1007/978-3-540-78791-4_14.

[19] A. Miné. A new numerical abstract domain based on difference-bound matrices. In O. Danvy and A. Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001. doi: 10.1007/3-540-44978-7_10.

[20] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004. doi: 10.1007/978-3-540-24725-8_2.

[21] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. doi: 10.1007/s10990-006-8609-1.

[22] F. Qian, L. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In R. N. Horspool, editor, *Compiler Construction*, pages 325–341. Springer, 2002. doi: 10.1007/3-540-45937-5_23.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.