The Julia Static Analyzer for Java

Fausto Spoto(✉)

Dipartimento di Informatica, Università di Verona, and Julia Srl, Verona, Italy
fausto.spoto@univr.it

Abstract. The Julia static analyzer applies abstract interpretation to the analysis and verification of Java bytecode. It is the result of 13 years of engineering effort based on theoretical research on denotational and constraint-based static analysis through abstract interpretation. Julia is a library for static analysis, over which many checkers have been built, that verify the absence of a large set of typical errors of software: among them are null-pointer accesses, non-termination, wrong synchronization and injection threats to security. This article recaps the history of Julia, describes the technology under the hood of the tool, reports lessons learned from the market, current limitations and future work.

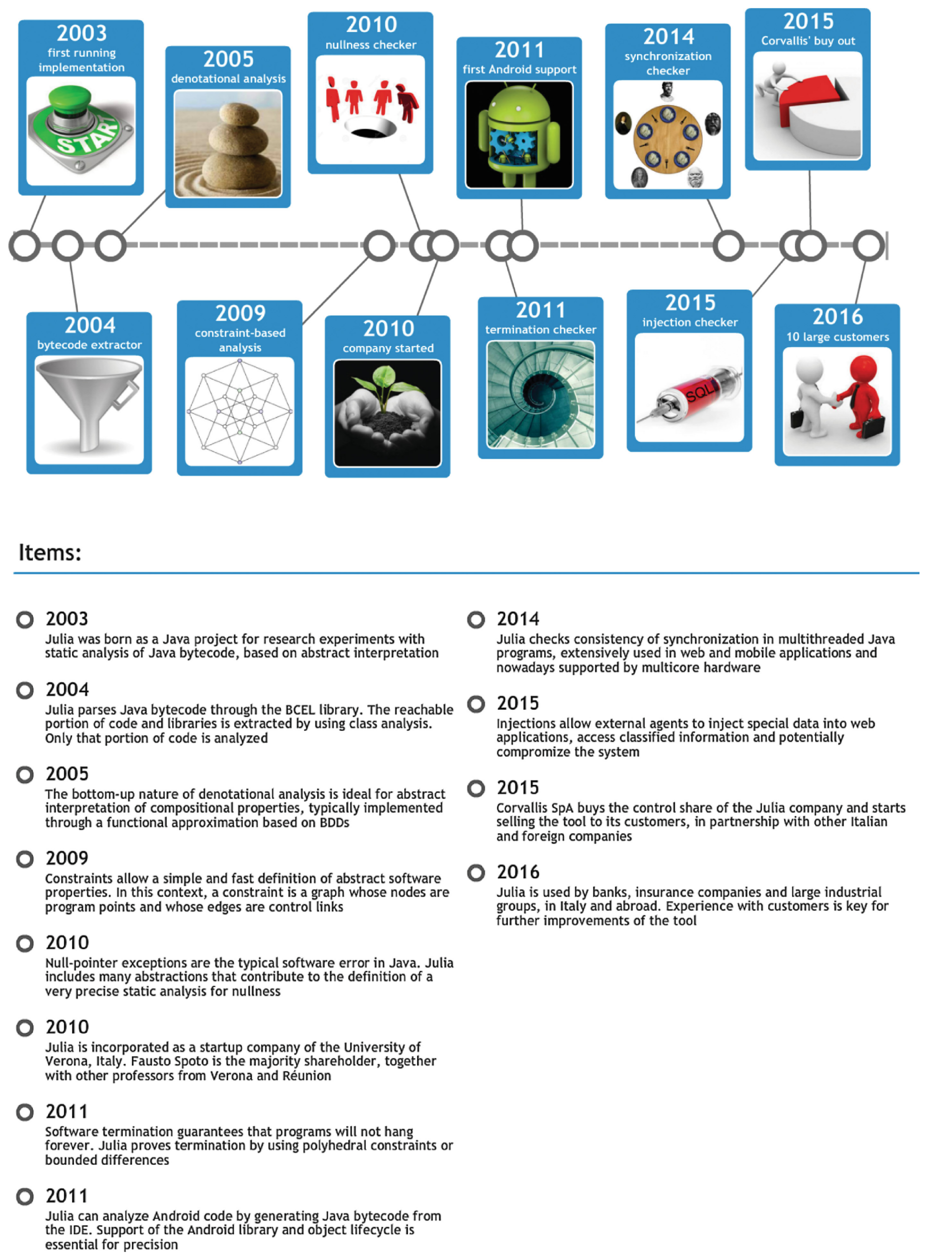
1 Introduction

The Julia analyzer applies static analysis to Java bytecode, based on abstract interpretation [7]. Its internal technology has been published already [9,10,21, 22,24,25,28–33] and the rest is a major, at times painful engineering effort, aimed at making theory match the complex reality of modern software, as such of virtually no direct interest to the scientific community. Hence, the goal of this invited article is twofold: give a brief overview of the technology, meant as a reference to other, more detailed articles; and report experience gathered from the transformation of the tool into a company that survives on the market, something that research scientists typically overlook but, at the end, is the sole justification for the existence of a research community that does not want to remain confined inside pure theoretical speculation.

Julia was born in 2003 to perform experiments that could complement and support the formalizations presented in scientific articles. Figure 1 shows the timeline and main milestones of the development of the tool. The main guideline was the development of a *sound* static analyzer. That is, Julia was meant to find all errors, without any false negatives. For this reason, abstract interpretation was the preferred theoretical basis. In practice, this means that the main issue has been the fight against false positives, whose reduction proved essential once the tool was incorporated into a company and customers started using it. Currently, the Julia analyzer is unique on the market in applying sound non-trivial static analyses to large Java programs, such as nullness, termination, synchronization and injection attacks (see Sects. 3, 4, 5 and 6). In particular, it is the only sound analyzer able to find SQL-injections and cross-site scripting attacks [20] in Java.

The Julia company was incorporated in 2010, after 7 years of software development, as a university startup, together with Roberto Giacobazzi from Verona

© Springer-Verlag GmbH Germany 2016
X. Rival (Ed.): SAS 2016, LNCS 9837, pp. 39–57, 2016.
DOI: 10.1007/978-3-662-53413-7_3



Items:

- 2003
  Julia was born as a Java project for research experiments with static analysis of Java bytecode, based on abstract interpretation
- 2004
  Julia parses Java bytecode through the BCEL library. The reachable portion of code and libraries is extracted by using class analysis. Only that portion of code is analyzed
- 2005
  The bottom-up nature of denotational analysis is ideal for abstract interpretation of compositional properties, typically implemented through a functional approximation based on BDDs
- 2009
  Constraints allow a simple and fast definition of abstract software properties. In this context, a constraint is a graph whose nodes are program points and whose edges are control links
- 2010
  Null-pointer exceptions are the typical software error in Java. Julia includes many abstractions that contribute to the definition of a very precise static analysis for nullness
- 2010
  Julia is incorporated as a startup company of the University of Verona, Italy. Fausto Spoto is the majority shareholder, together with other professors from Verona and Réunion
- 2011
  Software termination guarantees that programs will not hang forever. Julia proves termination by using polyhedral constraints or bounded differences
- 2011
  Julia can analyze Android code by generating Java bytecode from the IDE. Support of the Android library and object lifecycle is essential for precision
- 2014
  Julia checks consistency of synchronization in multithreaded Java programs, extensively used in web and mobile applications and nowadays supported by multicore hardware
- 2015
  Injections allow external agents to inject special data into web applications, access classified information and potentially compromize the system
- 2015
  Corvallis SpA buys the control share of the Julia company and starts selling the tool to its customers, in partnership with other Italian and foreign companies
- 2016
  Julia is used by banks, insurance companies and large industrial groups, in Italy and abroad. Experience with customers is key for further improvements of the tool

Fig. 1. Timeline of the development of the Julia static analyzer and company (courtesy of ReadWriteThink: www.readwritethink.org).

and Fred Mesnard and Étienne Payet from Réunion. Nevertheless, the tool needed further technological evolution to become strong enough for the market. From 2015, Julia Srl is part of the larger Italian group Corvallis and enjoys the support of its commercial network. The tool is currently used by banks and insurance companies as well as by large industrial groups, in Italy and abroad.

This article is organized as follows. Section 2 shows the technology implemented by the Julia library. Sections 3, 4, 5 and 6 show the most significant examples of analyses built over that library. Section 7 describes problems and solutions for the analysis of real software, which is never pure Java. Section 8 reports our experience with the engineering of a static analyzer. Section 9 reports problems faced when a university startup hits the market and a sound static analyzer must become a tool used on an everyday basis by non-specialized personnel. Section 10 concludes.

Throughout this article, examples are made over two simple Java classes. Figure 2 shows a synchronized vector, implemented through an array, that collects comparable, nullable objects and sort them through the `bubblesort()` method. Method `toString()` returns an HTML itemized list of the strings. Figure 3 is a Java servlet, *i.e.*, an entry point for a web service. It expects four parameters and collects their values inside a `Collector` of strings. It sorts those values and writes their HTML list as output (typically shown on a browser's window).

2 A Library for Static Analysis

Julia is a parallel library for static analysis of Java bytecode based on abstract interpretation, over which specific analyses can be built, called *checkers*. As of today, Julia has been used to build 57 checkers, that generate a total of 193 distinct classes of warnings, and a code obfuscator. The construction of the representation of a program in terms of basic blocks (Sect. 2.1) is done in parallel, as well as type inference. Checkers run in parallel. However, each single analysis is sequential, since its parallel version proved to be actually slower. The computation of a static analysis is asynchronous and yields a *future* (a token that allows blocking access to the result of a running computation [14]), hence it is still possible to launch many static analyses in parallel with a performance gain, as long as they do not interact.

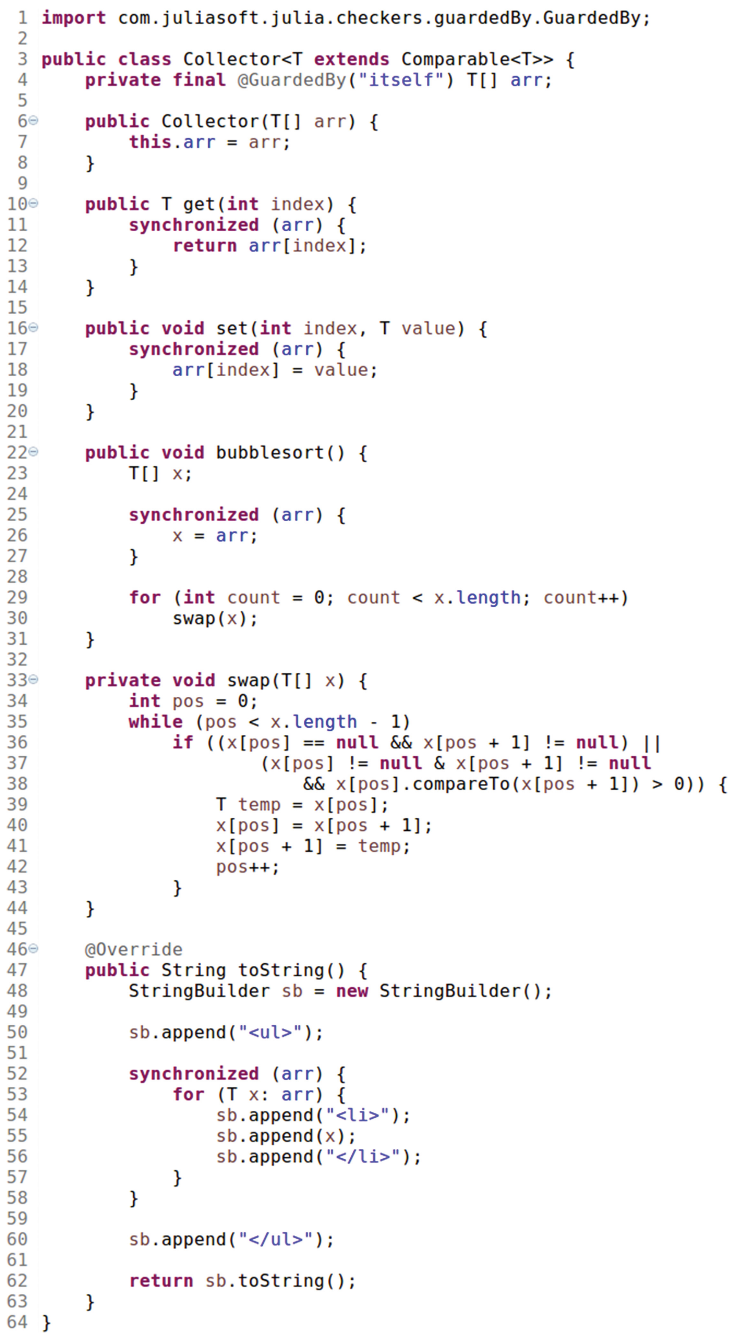
We describe below the support provided by the Julia library and then the four scientifically more appealing checkers that have been built over that library.

2.1 Representation of Java Bytecode

The library provides a representation of Java bytecode which is:

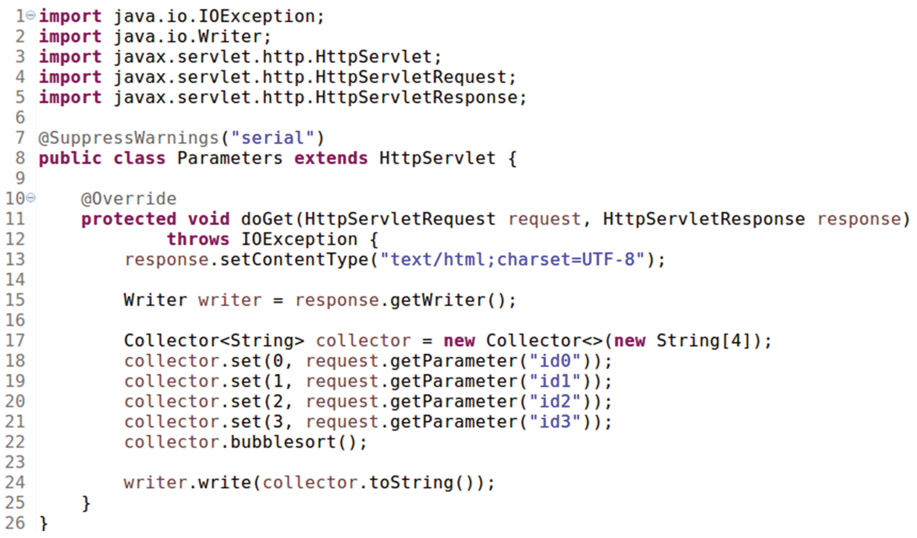
ready for abstract interpretation: all bytecodes[1] are state transformers, including those modelling exceptional paths; the code is a graph of basic blocks;

[1] In this article, *bytecode* refers both to the low-level language resulting from the compilation of Java and to each single instruction of that language. This is standard terminology, although possibly confusing.



```java
import com.juliasoft.julia.checkers.guardedBy.GuardedBy;

public class Collector<T extends Comparable<T>> {
    private final @GuardedBy("itself") T[] arr;

    public Collector(T[] arr) {
        this.arr = arr;
    }

    public T get(int index) {
        synchronized (arr) {
            return arr[index];
        }
    }

    public void set(int index, T value) {
        synchronized (arr) {
            arr[index] = value;
        }
    }

    public void bubblesort() {
        T[] x;

        synchronized (arr) {
            x = arr;
        }

        for (int count = 0; count < x.length; count++)
            swap(x);
    }

    private void swap(T[] x) {
        int pos = 0;
        while (pos < x.length - 1)
            if ((x[pos] == null && x[pos + 1] != null) ||
                    (x[pos] != null & x[pos + 1] != null
                        && x[pos].compareTo(x[pos + 1]) > 0)) {
                T temp = x[pos];
                x[pos] = x[pos + 1];
                x[pos + 1] = temp;
                pos++;
            }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append("<ul>");

        synchronized (arr) {
            for (T x: arr) {
                sb.append("<li>");
                sb.append(x);
                sb.append("</li>");
            }
        }

        sb.append("</ul>");

        return sb.toString();
    }
}
```

Fig. 2. A collector of comparable values.



```java
import java.io.IOException;
import java.io.Writer;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class Parameters extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
        response.setContentType("text/html;charset=UTF-8");

        Writer writer = response.getWriter();

        Collector<String> collector = new Collector<>(new String[4]);
        collector.set(0, request.getParameter("id0"));
        collector.set(1, request.getParameter("id1"));
        collector.set(2, request.getParameter("id2"));
        collector.set(3, request.getParameter("id3"));
        collector.bubblesort();

        writer.write(collector.toString());
    }
}
```

Fig. 3. A servlet that reads four parameters, sorts them and writes them as a list.

fully typed and resolved: bytecodes have explicit type information available about their operands, the stack elements and locals. Instructions that reference fields or methods are resolved, *i.e.*, the exact implementation(s) of the field or methods that are accessed/called are explicitly provided;

analysis-agnostic: it makes no assumption about the checker that will be applied, hence it does as few instrumentation as possible; in particular, it does not transform stack elements into locals not translate the code into three-address form.

For instance, Fig. 4 shows the representation of the bytecode of `bubblesort()` from Fig. 2. It shows exceptional paths, that are implicit in the source code and correspond to situations where a runtime exception is raised and thrown back to the caller. More complex exceptional paths are generated for explicit exception handlers. Figure 4 shows that field accesses and method calls are decorated with their resolved target(s), under square brackets. For instance, the access to the field signature `arr` is resolved into an access to field `Collector.arr` of type `Comparable[]`. The call to the method signature `swap()` is resolved into a call to `Collector.swap(Comparable[])`. In more complex situations, resolution is less trivial. For instance, Fig. 5 shows that the call to signature `compareTo()` at line 38 in Fig. 2 is resolved into a call to `String.compareTo(Object)`, since the program stores only strings inside a `Collector`. Field resolution is static in Java bytecode, while method resolution is dynamic and Julia applies a specific algorithm [23], known as *class analysis*. For simplicity, Fig. 4 does not show static types for each stack element and local variable at each bytecode. However, Julia infers them and makes them available in this representation.

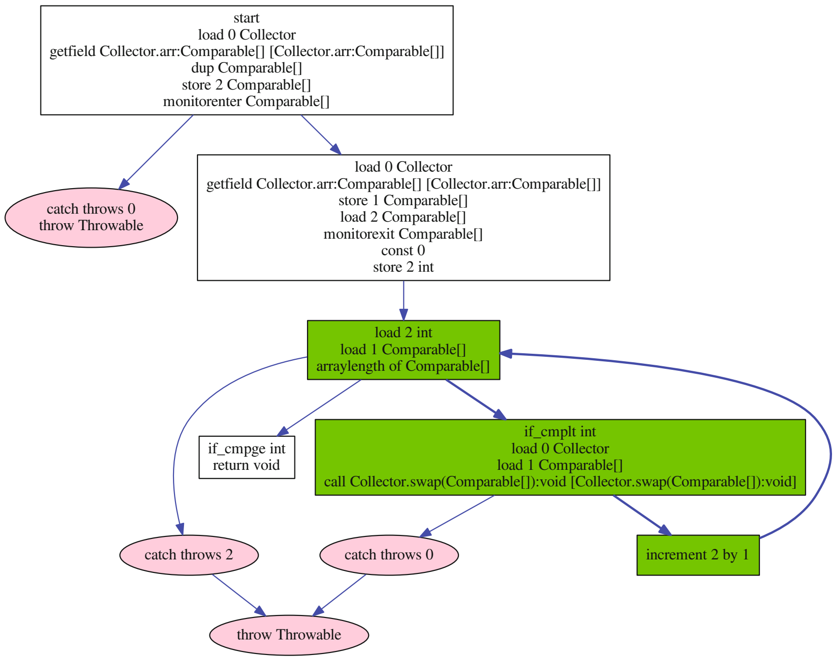


Fig. 4. The representation that Julia builds for method `bubblesort()` in Fig. 2. Ellipses are blocks implementing exceptional paths. Filled rectangles are the compilation of the `for` loop.

The Julia library builds over `BCEL` [13], for parsing the bytecode, whose later version makes Julia able to parse the latest instructions used for the compilation of lambda expressions and default methods in Java 8. An important issue is the portion of code that should be parsed, represented in memory and analyzed. Parsing the full program and libraries might be possible but programs use only a small portion of the libraries (including the huge standard library), whose full analysis would be prohibitive in time and space. Since Java bytecode is an object-oriented language where method calls are dynamic, determining the boundaries of the code to analyze is non-trivial. Julia solves this problem with class analysis [23], whose implementation has been the subject of extreme optimization, to let it scale to very large codebases, that possibly instantiate many classes and arrays. This implementation is called *bytecode extractor* and is a delicate component of Julia that takes into account many situations where objects are created by reflection (for instance, injected fields in Spring [18]). It is important to know the entry points of the application from where the extraction starts. Julia assumes by default that the `main()` method and callback redefinitions (such as `equals()` and `hashCode()`) are entry points. But it can be instructed to consider all public methods as entry points, or methods explicitly annotated as `@EntryPoint`.

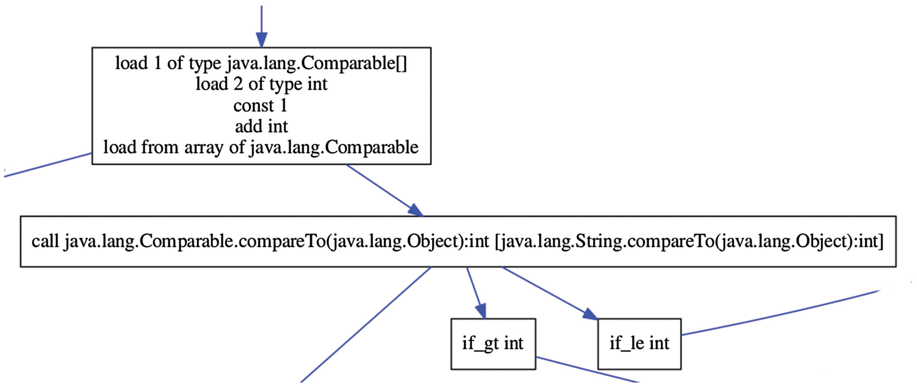

load 1 of type java.lang.Comparable[]
load 2 of type int
const 1
add int
load from array of java.lang.Comparable
call java.lang.Comparable.compareTo(java.lang.Object):int [java.lang.String.compareTo(java.lang.Object):int]
if_gt int
if_le int

Fig. 5. A portion of the representation that Julia builds for method `swap()` in Fig. 2.

2.2 Denotational Analyses

Support for denotational analyses has been the first provided by Julia, by applying to imperative object-oriented programs previous results for logic programs. Denotational semantics is a way of formalizing the semantics of programs [34], based on the inductive, bottom-up definition of the functional behavior of blocks of code. The definition is direct for the smaller components of the code and is inductive for larger components, made up of smaller ones. For loops and recursion, a fixpoint computation saturates all possible execution paths. By *semantics of programs*, in the context of Julia one must always understand *abstract semantics*, defined in a standard way, through abstract interpretation [7], so that it can be computed in finite time.

Julia is completely agnostic about this abstraction. In general, for Julia an abstract domain is a set of elements $A = \{a_1, \ldots, a_n\}$, that can be used to soundly approximate the behavior of each bytecode instruction, together with operations for least upper bound ($\sqcup$) and sequential composition ($\circ$) of abstract domain elements. Consider for instance Fig. 4. A denotational analyzer starts by replacing each non-`call` bytecode with the best abstraction of its input/output behavior (from pre-state to post-state) provided by A (Fig. 6). Then the analyzer merges abstractions sequentially, through the $\circ$ operator of A (Fig. 7). This process is known as *abstract compilation* [17]. The fixpoint computation starts at this point: Julia keeps a map ι from each block in the program to the current approximation computed so far for the block (an *interpretation*). Each `call` bytecode gets replaced with the $\sqcup$ of the current approximations $\iota(b_1) \ldots \iota(b_k)$ of its k dynamic targets (modulo variable renaming) and sequentially merged ($\circ$) with the approximation inside its block (as $a31$ in Fig. 7). Then $\iota(b)$ is updated with the approximation inside b sequentially composed ($\circ$) with the least upper bound ($\sqcup$) of the current approximations of its f followers: $\iota(b_1) \sqcup \ldots \sqcup \iota(b_f)$. This process is iterated until fixpoint. At the end, $\iota(b)$ is the abstraction of all execution traces starting at each given block b.

A nice feature of denotational analysis is that of being completely flow and context sensitive, fully interprocedural and able to model exceptional paths. In practice, this depends on the abstract domain A. In particular, its elements should represent functional behaviors of code (from pre-state to post-state) in order to exploit the full power of denotational analysis. In practice, this is achievable for Boolean properties of program variables (being `null`, being tainted, being cyclical, and so on), since Boolean functions can be used as representation for functional behaviors and efficiently implemented through binary decision diagrams (BDDs) [5]. Julia has highly optimized support for building abstract domains that use BDDs, It reduces the number of Boolean variables by abstracting the variables of interest only (for nullness, only those of reference type).

The implementation of denotational analysis is difficult for non-Boolean properties of variables (for instance, the set of runtime classes of the objects bound to a variable or the set of their creation points). Moreover, it provides only functional approximations of the code, rather than approximations of the state just before a given instruction. The latter problem is solved at the price of a preliminary transformation [24], that Julia implements.

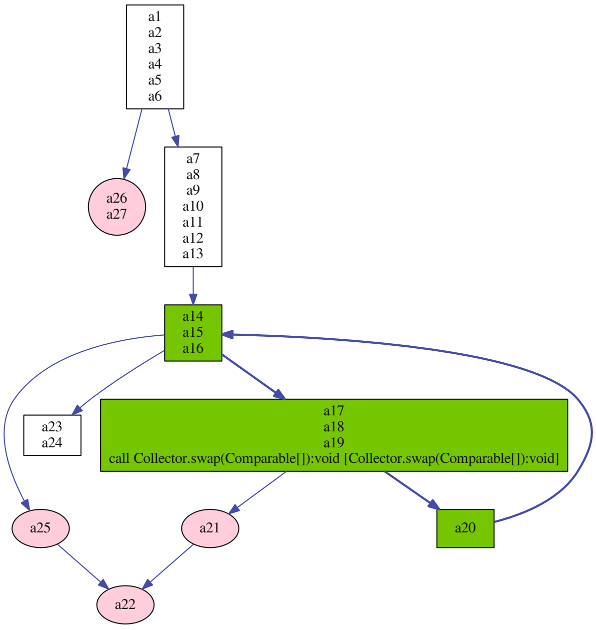

a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a26
a27
a14
a15
a16
a23
a24
a17
a18
a19
call Collector.swap(Comparable[]):void [Collector.swap(Comparable[]):void]
a25
a21
a20
a22

Fig. 6. The abstraction of the non-`call` bytecodes of method `bubblesort()` in Fig. 2.

2.3 Constraint-Based Analyses

A constraint-based analysis translates a Java bytecode program into a set-constraint, whose nodes are (for instance) blocks of code and whose edges

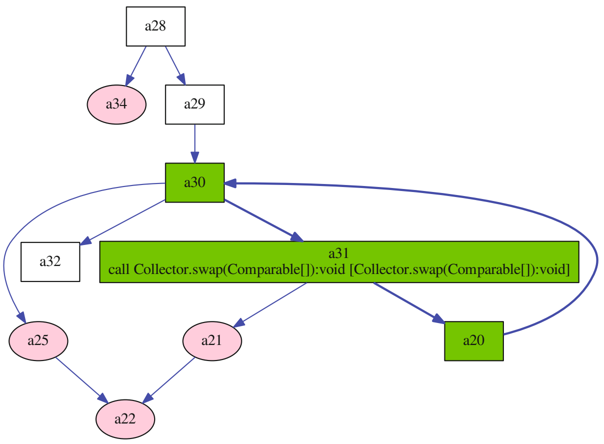

a28
a34
a29
a30
a32
a31
call Collector.swap(Comparable[]):void [Collector.swap(Comparable[]):void]
a25
a21
a20
a22

Fig. 7. The sequential merge of the abstraction of method `bubblesort()` in Fig. 2.

propagate abstract information between nodes. The approximation at a node is the least upper bound of the abstract information entering the node from each in-edge. Abstract information is propagated along the edges until fixpoint. The constraint for Fig. 4 might be as in Fig. 8, where each node contains the fixpoint abstract information at that node and arrows represent propagation of abstract information. Method calls have been flattened by linking each call place to the first node of the target(s) of the call. Each `return` instruction inside a callee is linked back to the non-exceptional followers of the `calls` in the callers. Each `throw` bytecode inside a callee is linked back to the exceptional followers of the `calls` in the callers. There might exist a direct link from a `call` instruction to its subsequent instructions, such as between the node approximated with $a6$ and that approximated with $a9$ in Fig. 8. This edge can model side-effects induced by the execution of the method, if they can occur.

The nice feature of constraint-based analysis is that it can be easily applied to every kind of abstract domain, also for non-Boolean properties. For instance, abstract information might be the set of already initialized classes; or undirected pairs of variables that share; or directed pairs of reachable variables. Moreover, there is large freedom about the granularity of the constraint: nodes might stand for approximations at blocks of code; but also for approximations at each single instruction (by splitting the blocks into their component instructions); or even for approximations of each single local variable or stack element at each program point. The latter case allows one for instance to compute the set of creation points for each variable; or the set of runtime classes for each variable; or the set of uninitialized fields for each variable. Finally, edges can propagate abstract information by applying any propagation rule, also one that might transform that information during the flow. The limitation of constraint-based analysis is that it is inherently context-insensitive, since method calls have been flattened: there is a merge-over-all-paths leading to each given method.

The Julia library includes highly optimized algorithms for building constraints and computing their fixpoint, with different levels of granularity. These algorithms strive to keep the constraint as small as possible, by automatically collapsing nodes with identical approximation. Abstract information at each node is kept in a bitset, for compaction. Propagation is particularly optimized for additive propagation rules.

2.4 Predefined Static Analyses

The Julia library builds on its infrastructure for denotational and constraint-based analyses and provides a set of general-purpose predefined analyses, that can be useful for building many checkers. In particular, the library provides:

- a possible class analysis for variables [23,32] (constraint-based, used for the extraction algorithm) and its concretization as creation-points analysis;
- a definite aliasing analysis between variables and expressions [21] (constraint-based);
- a possible sharing analysis between pairs of variables [29] (constraint-based);
- a possible reachability analysis between pairs of variables [22] (constraint-based);
- a numerical analysis for variables, known as *path-length* [33] (denotational);
- a possible cyclicity analysis for variables [28] (denotational);
- a possible nullness analysis for variables [30] (denotational);
- a definite non-`null` analysis for expressions [30] (constraint-based);
- a definite initialization analysis for fields [31] (constraint-based);
- a definite locked expressions analysis [10] (constraint-based);
- a possible information flow analysis for variables [9] (denotational).

3 Nullness Checker

A typical error consists in accessing a field or calling a method over the `null` value (a *dereference* of `null`). This error stops the execution of the program with a `NullPointerException`. Julia can prove that a program will never raise such exception, but for a restricted set of program points where it issues a warning. The Nullness checker of Julia uses a combination of more static analyses. A first analysis approximates the Boolean property of being `null` for program (local) variables, by using BDDs that express constraints on all possible nullness behaviors of a piece of code [30]. However, object-oriented programs store values in fields and not just local variables. This makes things much more difficult, since a field holds `null` by default: hence Julia proves also that a field is always initialized before being read [31]. Moreover, expressions might be locally non-`null` because of some previous non-nullness check. Julia can prove it, provided the expression does not change its value between check and dereference [30].

Consider for instance the code in Figs. 2 and 3. The Nullness checker of Julia issues *only* the following warning:

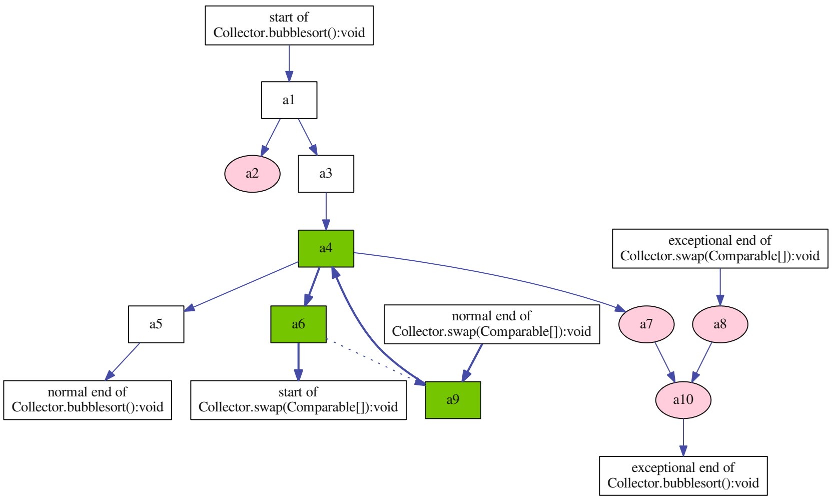

start of Collector.bubblesort():void
a1
a2
a3
a4
exceptional end of Collector.swap(Comparable[]):void
a5
a6
normal end of Collector.swap(Comparable[]):void
a7
a8
normal end of Collector.bubblesort():void
start of Collector.swap(Comparable[]):void
a9
a10
exceptional end of Collector.bubblesort():void

Fig. 8. A constraint generated for method `bubblesort()` in Fig. 2.

```
Collector.java:38: [Nullness: FormalInnerNullWarning]
  are the elements inside formal parameter "x" of "swap" non-null?
```

First of all, this proves that the program will *never* raise a `NullPointer Exception` elsewhere. Let us check that warning then. At line 38 of Fig. 2 the value of `x[pos]` can actually be `null`, since the array `x` is allowed to hold `null` elements (parameters of a servlet are `null` when missing) and the test `x[pos] != null` at line 37 is not enough to protect the subsequent dereference at line 38. The reason is that the programmer at line 37 has incorrectly used the eager `&` operator, instead of the lazy `&&`. Hence the dereference at line 38 will be eagerly executed also when `x[pos]` holds `null`.

In order to appreciate the power of this checker, consider that, after fixing the bug, by replacing `&` with `&&` at line 37, the Nullness checker does not issue any warning anymore, which proves that the program will *never* raise a `NullPointerException`. This is achieved through the expression non-`null` analysis [30], that now proves expression `x[pos]` to be non-`null` at line 38.

4 Termination Checker

Most programs are expected to terminate. A non-terminating program will never provide an answer to a query or will hang a device. The Termination checker of Julia proves that loops terminate, by translating Java bytecode into logic programs over linear constraints [2,33], whose termination is subsequently proved by using traditional techniques for logic programs [4,6,19,26]. Since imperative programs allow shared data structures and destructive updates, the inference of linear constraints needs the support of sharing and cyclicity analysis [28, 29]. Julia implements linear constraints with bounded differences [8]. For small programs, Julia can use the more precise and more expensive polyhedra [3] or a mixed implementation of bounded differences and polyhedra.

Consider for instance the code in Figs. 2 and 3. The Termination checker of Julia issues *only* the following warning:

```
Collector.java:35: [Termination: PossibleDivergenceWarning]
  are you sure that Collector.swap always terminates?
```

Hence it proves that all other loops in the program terminate. The reason is that variable `pos` is incremented at line 42, but that line is only executed inside the body of the `if` at line 36. If the statement `pos++` is moved outside the body of the `if`, then the code is correct and Julia proves termination of this loop as well. This result is achieved through the use of a preliminary definite aliasing analysis [21] that proves that, during the execution of the loop, the strictly increasing value of `pos` is always compared against the value of the expression `x.length-1` and that value does not change, not even by side-effect.

5 Synchronization Checker

Concurrency is more and more used in modern software. It supports performance, by running algorithms in parallel on multicore hardware, but also responsivity, by running long tasks on threads distinct from the user interface thread (typical scenario in Android programming [15]). But concurrency is difficult and programmers tend to write incorrect concurrent code. Code annotations have been proposed [14] as a way of specifying synchronization policies about how concurrent threads can access shared data to avoid *data races* (concurrent access to the same data). The semantics of such annotations has been recently formalized [11] and tools exist nowadays that check and infer them [10]. Julia has a GuarbedBy checker that is able to prove the correctness of annotations already provided by the programmer, but can also infer sound annotations for fields and method parameters [10], that were not explicitly written in code.

Consider for instance Fig. 2 and 3. Field `arr` at line 4 of Fig. 2 is annotated as `@GuardedBy("itself")`. This means that a thread can dereference the *value* of field `arr` only at program points where that thread locks the value itself [11]. For instance, this is what happens at lines 12, 18 and 53. Nevertheless, the GuardedBy checker of Julia issues the following warning:

```
Collector.java: [GuardedBy: UnguardedFieldWarning]
  is "itself" locked when accessing field "arr"?
```

Indeed, the value of `arr` is copied into variable `x` at line 26 and later dereferenced at line 29 and inside method `swap()`, without holding the lock on that value. The programmer might have expected to optimize the code by reducing the span of the critical section to line 26 only. But this is incorrect since it protects, inside the critical section, only the *name* `arr` (which is irrelevant) rather than the *value* of `arr` (which is semantically important). If the synchronization starting at line

25 is extended until the end of line 30, Julia does not issue any warning anymore and infers the `@GuardedBy("itself")` annotation for field `arr`.

6 Injection Checker

Injection attacks are possibly the most dangerous security bugs of computer programs [20], since they allow users of a web service to provide special input arguments to the system, that let one access sensitive data or compromize the system. The most famous attacks are SQL-injection and cross-site scripting (XSS). Julia has an Injection checker that applies taintedness analysis through an abstract domain for information flow, made of Boolean formulas and implemented as BDDs [9].

Consider for instance the code in Figs. 2 and 3. The Injection checker of Julia issues only the following warning:

```
Parameters.java:24: [Injection: XSSInjectionWarning]
  possible XSS-injection through the 0th actual parameter of write
```

Lines 18–21 in Fig. 3 actually store some parameters of the request inside the collector and such parameters can be freely provided by an attacker. They are then sorted at line 22 and written to the output of the servlet as an HTML list, at line 24. Hence, if the attacker provides parameters that contain special HTML of Javascript tags, they will be rendered by the browser and hence open the door to any possible attack.

The power of Julia's analysis is related to a new notion of taintedness for data structures, that does not classify fields as tainted or untainted on the basis of their name, bur considers instead an object as tainted if tainted data can be *reached* from it [9]. For instance, if another `Collector` would be created in the code in Fig. 3, populated with strings not coming from the parameters of the request, and written to the output of the servlet, then Julia would not issue a warning for that second `write()`. Both collectors would be implemented through field `arr` (Fig. 2) but only one would actually reach tainted data. This object-sensitivity justifies the precision of the Injection checker of Julia, compared to other tools [9].

7 Frameworks: Java Does Not Exist

Julia is a static analyzer for Java bytecode. But real programs nowadays are very rarely pure Java code. Instead, they are multilanguage software based on a large variety of *frameworks*. For instance, Java often integrates with Java Server Pages (JSPs), another programming language that is used to program the views of web applications. If a static analyzer does not understand JSPs, then it will miss part of the code, non-nullness information, information flows and injection attacks. A framework is, instead, a library that simplifies the implementation of frequently used operations and very often modifies the semantics of the language by introducing new behaviors through reflection. The typical example, in the banking sector, is Spring [18], that provides declarative configuration of software applications through XML or annotation-based specification of *beans*; as well as simplified implementations of web services and of the model-view-controller design pattern for web applications; it also provides aspect-oriented programming. Spring applications are often built on top of a persistence framework, typically Hibernate [16].

If a static analyzer does not understand the multilanguage nature of software or does not understand the semantical modifications that frameworks induce in Java, then it will issue many false alarms and also miss actual bugs. That is, it will become imprecise and unsound. Julia manages JSPs by compiling them into Java code through the Jasper compiler [12]. Julia embeds partial support for Spring, in particular for bean instantiation and their injection as dependencies. Nevertheless, this is work in progress and the evolution of the frameworks tends to be faster than our ability to deal with their new features.

A notable framework is Android. Knowledge about the Android standard library is essential to avoid many false alarms. For instance, Julia has been instructed to reason about the way views are constructed in an Android activity from their XML specification, in order to reduce the number of false alarms about wrong classcasts and `null`-pointer dereferences [25]. A big problem is that Julia analyzes Java bytecode, but Android applications are packaged in Dalvik bytecode format. Currently, it is possible to export the Java bytecode from the integrated development environment or convert Dalvik into Java bytecode, with the Dex2Jar tool [1].

8 Engineering Problems

A static analyzer for a real programming language is very complex software. It implements involved semantical definitions; it is highly optimized in order to scale to the analysis of large programs; it uses parallel algorithms, whose implementation requires experience with multithreading; it must be clear, maintainable and expandable, which requires object-orientation and design pattern skills. The natural question is then who can ever write such software. Julia has been developed in 13 years, largely by a single person, who is a university researcher and a passionate programmer. It is possibly the case that a very professional development team could have be used instead, but a deep involvement of researchers in the development effort seems unavoidable. In particular, the complexity and duration of the effort seems to us incompatible with spotty development by students or PhD's, which would result in fragmentation, suboptimal implementation and final failure. However, researchers are usually reluctant to spending so much time with activities that do not translate into published papers. This is understandable, since their production is typically measured in terms of publications only. It seems to us necessary to find alternative ways of measuring the production of researchers, by including their direct involvement in technology transfer, at the right weight.

After a static analyzer has been developed by researchers, it must eventually be entrusted to professional programmers, who will continue with the less critical development of the tool. This is a delicate moment and requires time. Knowledge can pass from researchers to programmers but this is a long term investment with no immediate monetary benefit. As with any complex software, the alternative is to lose experience and finally memory about the code, which would mean utter disaster. The best solution is to open-source the code of the analyzer, if this is compatible with the business model of the company.

Maintenance of a static analyzer is another difficult engineering task. In our experience, a small *improvement* of the analyzer might change its precision or computational cost very much, or introduce bugs. Hence it becomes important to find ways for reducing the risk of regression. Since Julia is written in Java, we have of course applied Julia to Julia itself from time to time, found errors and debugged the analyzer. But this is not possible for functional bugs. Testcases are more helpful here, but there is little experience with testing of static analyzers. We have developed hundreds of higher level tests, that run the static analysis of a given program and compare the result with what was obtained in the past. Such tests must be updated from time to time, since Julia performs more and more checks and becomes more and more precise. Of course, this process cannot be done by hand. Instead, Julia has the ability to translate, automatically, the result of a static analysis into a JUnit test class, that can later be used to check for non-regression. Hence testcases can be updated automatically. For instance, the following JUnit testcase is automatically generated from the analysis of the code in Fig. 2 and 3 with the checkers Nullness, Termination, GuardedBy and Injection. It re-runs the analyses, asserts the existence of the expected warnings and the non-existence of other warnings:

```
@Test
public void test() throws WrongApplicationException, ClassNotFoundException {
  Program program = analyse();
  ClassMembersBuilder classMembersBuilder = new ClassMembersBuilder(program);

  assertWarning(new UnguardedFieldWarning("Collector.java", "arr", "itself"));
  assertWarning(new PossibleDivergenceWarning
    (classMembersBuilder.getMethod("Collector", "swap", "void", "Comparable[]")));
  assertWarning(new FormalInnerNullWarning("Collector.java", 38, "Collector", "swap", 0));
  assertWarning(new XSSInjectionWarning("Parameters.java", 24,
    classMembersBuilder.getMethodReference("java.io.Writer", "write", "void", "String"), 0));
  assertNoMoreWarnings();
}
```

9 From Research to Market

Pushing new technology from laboratory to market is definitely exciting, but it is also source of deception. In the case of Julia, the key point of *soundness* has been the hardest to communicate to the market. Customers often already use a static analysis tool, which is typically highly unsound, and do not see soundness as a reason to change. Instead, they tend to highlight non-scientific aspects, such as the ability to integrate the tool into their organization, its ease of use, the quality of the graphics of the reports, the classification of the warnings inside well-known grids, the ability to work on specific software technology, which is typically multilanguage and based on frameworks. Very few customers seem impressed by soundness. Instead, customers tend to support other tools that do *everything*, without questioning how good such tools actually are at doing this *everything*. This situation might change in the future, but decades of misinformation will not be easily forgotten. From this point of view, the dichotomy between scientific publication and industrial communication must be eventually resolved, but this requires good will on both sides.

Hitting the market is also an opportunity to discover how *real* software looks like and hence what a static analyzer should be able to analyze. For instance, very often programmers do not initialize objects inside their constructors, but leave them uninitialized and later call *setter methods*. This is sometime the consequence of the use of frameworks, such as Java Beans or Hibernate, or it s just a programming pattern. In any case, it hinders static analysis, since uninitialized fields hold their default value in Java (`null` or 0), which induces the analyzer to issue hundreds of warnings, since objects can be used before being fully initialized. Similarly, programmers never check for nullness of the parameters of public methods, which leads to warnings if such parameters are dereferenced. We stress the fact that all these warnings are real bugs, in principle. But programmers will never see it that way, since they assume that methods are called in a specific order and with specific non-`null` arguments. Of course, this is not documented, not even in comments. Julia reacts to this situation with analyses of different strictness: for instance, together with the Nullness checker, Julia has a BasicNullness checker that is optimistic *w.r.t.* the initialization of fields and the nullness of the parameters passed to public methods.

Another problem faced with *real* software is that this is written against all programming recommendations. We found methods of tens of thousands of lines, with hundreds of variables. This means that abstract domains based on BDDs or linear inequalities typically explode during the analysis. We applied worst-case assumptions, triggered when computations become too complex, although this results in more false alarms. But naive triggers are sensitive to the order of computation of the abstract analysis, *i.e.*, they might fire non-deterministically, which is unacceptable in an industrial context. Hence we had to find more robust triggers for the worst-case assumption, that do not depend on the order of computation of the analysis.

10 Conclusion

The Julia static analyzer is the result of 13 years of research and engineering. It is the proof that research from academia can move to the market and provide a solution to an actual market need. It is exciting to see theoretical results about static analysis, BDDs, polyhedra and fixpoint computation applied to solve real problems of large banks, insurance companies, automobile industries and simple freelance programmers around the world.

This article presents a synthetic view of the history of Julia and of its underlying technology and strengths. It acknowledges the problems faced once the tool is used to analyze actual software, written in peculiar ways and using reflection, also through several frameworks, and the expectations that customers have about such technology. These aspects are the real issues that still jeopardize the success of sound static analysis outside the academic world.

The development of Julia continues. There are many open problems that need an answer. First of all, concurrency is an opportunity not yet completely exploited, that can improve the efficiency of the tool by using more parallel algorithms. It is also a problem, since the analysis of concurrent programs is difficult and current techniques are often unsound in that context. Also the presentation of the warnings needs improving. Instead of a flat list, possibly organized into static priority classes, it might be possible to rank warnings *w.r.t.* their *features*, by using machine learning [27]. Finally, the applicability of Julia will be expanded. The translation from CIL bytecode into Java bytecode is already implemented and should allow, in the near future, to analyze safe CIL code with Julia, such as that derived from the compilation of C#. The application to other programming languages is possible in principle, but seems more difficult. Large parts of the Julia library might be recycled for other languages, in particular the fixpoint algorithms, but the specific analyses need to be built again from scratch.

References

1. Dex2Jar. https://sourceforge.net/projects/dex2jar
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
4. Bagnara, R., Mesnard, F., Pescetti, A., Zaffanella, E.: A new look at the automatic synthesis of linear ranking functions. Inf. Comput. **215**, 47–67 (2012)
5. Bryant, R.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
6. Codish, M., Lagoon, V., Stuckey, P.J.: Testing for termination with monotonicity constraints. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 326–340. Springer, Heidelberg (2005)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
8. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)

9. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in Java. In: Davis, M., et al. (eds.) LPAR-20 2015. LNCS, vol. 9450, pp. 130–145. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_10
10. Ernst, M.D., Lovato, A., Macedonio, D., Spoto, F., Thaine, J.: Locking discipline inference and checking. In: Proceedings of Software Engineering (ICSE 2016), Austin, TX, USA, pp. 1133–1144. ACM (2016)
11. Ernst, M.D., Macedonio, D., Merro, M., Spoto, F.: Semantics for locking specifications. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 355–372. Springer, Heidelberg (2016). doi:10.1007/978-3-319-40648-0_27
12. The Apache Software Foundation. Jasper 2 JSP Engine How To. https://tomcat.apache.org/tomcat-8.0-doc/jasper-howto.html
13. The Apache Software Foundation. Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel. 24 June 2016
14. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison Wesley, Reading (2006)
15. Göransson, A.: Efficient Android Threading. O'Reilly Media, Sebastopol (2014)
16. Red Hat. Hibernate. Everything Data. http://hibernate.org
17. Hermenegildo, M., Warren, D.S., Debray, S.K.: Global flow analysis as a practical compilation tool. J. Logic Program. **13**(4), 349–366 (1992)
18. Pivotal Software Inc. Spring Framework. https://projects.spring.io/spring-framework
19. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proceedings of Principles of Programming Languages (POPL 2001), pp. 81–92. ACM (2001)
20. MITRE/SANS. Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25. September 2011
21. Nikolić, Ð., Spoto, F.: Definite expression aliasing analysis for Java bytecode. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 74–89. Springer, Heidelberg (2012)
22. Nikolić, Ð., Spoto, F.: Reachability analysis of program variables. ACM Trans. Program. Lang. Syst. (TOPLAS) **35**(4), 14 (2013)
23. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1991). ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM, November 1991
24. Payet, É., Spoto, F.: Magic-sets transformation for the analysis of Java bytecode. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 452–467. Springer, Heidelberg (2007)
25. Payet, É., Spoto, F.: Static analysis of android programs. Inf. Softw. Technol. **54**(11), 1192–1201 (2012)
26. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
27. Raychev, V., Bielik, P., Vechev, M.T., Krause, A.: Learning programs from noisy data. In: Proceedings of Principles of Programming Languages (POPL 2016), St. Petersburg, FL, USA, pp. 761–774. ACM (2016)
28. Rossignoli, S., Spoto, F.: Detecting non-cyclicity by abstract compilation into boolean functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 95–110. Springer, Heidelberg (2006)

29. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
30. Spoto, F.: Precise null-pointer analysis. Softw. Syst. Model. **10**(2), 219–252 (2011)
31. Spoto, F., Ernst, M.D.: Inference of field initialization. In: Proceedings of Software Engineering (ICSE 2011), Waikiki, Honolulu, USA, pp. 231–240. ACM (2011)
32. Spoto, F., Jensen, T.P.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. (TOPLAS) **25**(5), 578–630 (2003)
33. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst. (TOPLAS) **32**(3), 1–70 (2010)
34. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)