ScalaSCA: A Static Bug Finder for Scala*

Jean André Gauthier
M.S.c. Computer Science, EPFL
jean.gauthier@epfl.ch

Viktor Kuncak†
EPFL - Lab for Automated Reasoning and Analysis (LARA)
viktor.kuncak@epfl.ch

Iulian Dragos‡
Typesafe Lausanne
iulian.dragos@typesafe.com

ABSTRACT

The aim of static code analyses is to detect programming or logic errors before execution, in order to avoid time-consuming code inspection, and most importantly to prevent errors that occur only on a small number of execution paths and might not be discovered during debugging. Unfortunately, a wide range of those static analyses are computation-intensive, and cannot be included by default as a compiler phase without disrupting the programmer's workflow. In this paper we present *ScalaSCA*, a both fast and extensible static code analysis engine that is implemented as a plugin for the Scala compiler. We discuss some of the bug finding strategies we have implemented and lastly, we measure their performance overhead compared to a standard execution of the Scala compiler.

Categories and Subject Descriptors

F.3.2.5 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Verification, Measurement

Keywords

Static Analysis, Scala, Testing

1. INTRODUCTION

In an ideal world, software engineers would be fully reliable and commit no errors at all while developing their software. Since this is obviously not the case, the best they can do is to try to limit the occurrence of such bugs and to standardise the processes of error reporting. This is critical for minimising the amount of time a given bug is left unpatched, and leads to a greater satisfaction of the private or corporate end-users.

As shown in Figure 1, the later the stage at which a coding (or design) error is detected, the higher the cost associated to its resolution. Hence, it would make make sense to introduce additional checks at the first machine-level stage, i.e. construction, in order to prevent at least some of the bugs that would usually be detected at later stages of software development. As an example, "one in a million" errors that only occur on very few execution paths.

There exist several ways to tackle those issues, each of which has its specific advantages and disadvantages:

- Manual code review: as developers are notoriously poor at discovering their own bugs, this often implies that two or several developers mutually review their code. But such reviews are often time-consuming (and thus costly), and due to the sheer sizes and complexities of the codebases, very few errors are actually discovered which leads to a poor cost to bug discovery ratio.
- Dynamic code analysis: as a first step towards code analysis automation, these analysers[1] rely on test cases or interactive input during the execution of the target program. Typically, debuggers fall into this category of analysers. While these tools are particularly useful when resolving memory management issues or any dynamic problems involving e.g. reflection, they do not guarantee a complete code coverage for their analysis.
- Static code analysis: on the other side of the spectrum, static code analysers do not execute the target code at all. This has the consequence that these tools tend to be less suited to the task of solving concurrency or memory issues.[2] Static code analysers are nevertheless an invaluable tool, inasmuch that they usually have a complete code coverage.

Ideally, a static code analyser would produce a correctness proof about the code in question. Unfortunately, this is currently infeasible due to the complexity of this task. A more promising approach are correctness proofs about a reduced number of properties, usually specified as assertions. The task of the static analyser is then reduced to proving that the assertion's predicate holds on all execution paths, where relatively good results can be obtained. [3] A last category of analysers does not try to provide proofs at all but rather flags probable bugs, which is the approach chosen for *ScalaSCA*.

*Submitted in partial fulfilment of the requirements for CS-498 - Project in Computer Science II, at École Polytechnique Fédérale de Lausanne

†Professor in charge of the project

‡Supervisor of the project

[1]Cf. [10] for the presentation of Valgrind, a collection of dynamic code analysis tools for the C language.

[2]It should however be noted that this field is an active area of research and there indeed exist techniques that allow a static analysis for potential concurrency issues, as presented in [12] for example

[3]The Leon verification system for Scala is an example of such a tool. [1]

Time Introduced	Time Detected				
	Re-quire-ments	Archi-tecture	Con-struc-tion	System Test	Post-Re-lease
Requirements	1	3	5-10	10	10-100
Architecture	—	1	10	15	25-100
Construction	—	—	1	10	10-25

Figure 1: Cost reduction through bug detection at construction time [9]

However, it is important to remember that in practice, code analysis tools are measured against two metrics only, when being integrated in software development cycles:

- False positives: If the analyser produces too many unnecessary warnings, developers tend to start ignoring them, and this effectively nullifies the purpose of the tool.
- Execution time: If on the other hand the analyser takes too long to run, it is likely to be taken out of the iterative software development cycle and run too rarely to be helpful in preventing common coding errors.

Thus, a static analyser has to strike a balance between performance and accuracy in order to be used in the most effective manner, i.e. during each codebase compilation. This explains why we deliberately oriented the development of *ScalaSCA*'s towards unsound code analyses, in an attempt to improve performance at the expense of making the tool slightly less accurate.

2. SCALASCA INTERNALS

ScalaSCA has been implemented in order to provide a standardised framework for bug finding strategies. This allows an easier sharing of those strategies across the community, and simplifies the task of programming personalised strategies, if needed.

So-called "rules" are the core component of *ScalaSCA*, as they represent code snippets that perform a specific code analysis. In order to unify the referencing of each one of the rules and allow an easier lookup in bug trackers, a unique identifier has been attributed to each rule, much in the style of *Findbugs* [**?**]. *ScalaSCA* differentiates between two types of rules, i.e. rules that can be applied directly on abstract syntax trees (inheriting from the **ASTRule** class) and general rules (which inherit from **StandardRule** class).

Not all categories of rules can be expressed easily

2.1 Personalised rules

ScalaSCA allows the inclusion of independently developed rules by passing the `-P:scalasca:c:<config-file-path>` to *scalac* when using *ScalaSCA*.

3. EXAMPLE RULES

Broadly speaking, we have implemented three types of rules:

- Syntactic Rules: these rules work exclusively on ASTs, and do not rely on any intermediate representation. GEN_BLOCK_CONST_PROP
- Control-flow rules:
- Data-flow rules:
- Code metrics:

3.1 AST Rules

3.2 The Rule Engine

4. IMPLEMENTATION

ScalaSCA has been made available free of charge at `https://github.com/jean-andre-gauthier/scalasca`, under the 3-clause BSD license. The current implementation is entirely written in Scala, and does not make use of any additional libraries other than the standard Scala library and the Scala compiler library.

As *ScalaSCA* has been developed using the *sbt* interactive build tool, it provides out-of-the-box integration with any existing *sbt* project by simple addition of the following line to any *sbt* project configuration file:

```
addCompilerPlugin("lara.epfl"%%"scalasca
    "%"0.1")
```

This will automatically run the `DefaultRule` from *ScalaSCA*. In order to change this behaviour, one or several personalised rules can be set up, compiled to *jar* files and their paths stored line by line in a configuration file. This rules can then be accessed by *ScalaSCA* by specifying the `-P:scalasca:c:<config-file-path>` option.

As an example, in an *sbt* configuration file, this option would be specified as follows:

```
scalacOptions <<= (scalacOptions,
    scalaSource in Compile) { (options,
    base) =>
     options :+ ("-P:scalasca:c:" +
         configFilePath)
}
```

For additional details, a sample *ScalaSCA* plugin project has been made available online, at `https://github.com/jean-andre-gauthier/scalasca-plugin`.

5. PERFORMANCE

6. RELATED WORK

Research in static code analysis has been conduced for most programming languages, one of the earliest examples being the *lint* checker for the *C* programming language [6]. *lint* inspired many code analysis tools, such as *splint* [2] for example, whose philosophy is to avoid false positives at all costs. Among others, *Goanna* for *C++* [4] takes an interesting approach as it is based on a model checker (*NuSMV*) but still manages to produce relatively good performance. Much work has also been dedicated to the *Java* programming language, and various tools have been developed, such as *FindBugs* [3] that focuses on simplicity over complexity, or *soot* [11] which is a framework that provides intermediate representations of java bytecode. *ScalaSCA* is not the first foray into static analysis on Scala code, and some more complex tools that stay accurate even in the presence of first-class functions are available, e.g. *Insane* [7]. Dynamic programming languages have features which tend to complicate

static analyses, as they generally expose less information to the analyser. Previous research has been investigating static analysis for languages such as PHP [8] or Javascript [5].

7. REFERENCES

[1] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.

[2] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, Jan. 2002.

[3] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM.

[4] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer. Goanna: Syntactic software model checking. In S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Automated Technology for Verification and Analysis*, volume 5311 of *Lecture Notes in Computer Science*, pages 216–221. Springer Berlin Heidelberg, 2008.

[5] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[7] E. Kneuss, V. Kuncak, and P. Suter. Effect analysis for programs with callbacks. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 48–67. Springer Berlin Heidelberg, 2014.

[8] E. Kneuss, P. Suter, and V. Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 300–314, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2 edition, 2004.

[10] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[11] 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, pages 13–. IBM Press, 1999.

[12] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 602–629, Berlin, Heidelberg, 2005. Springer-Verlag.