Sulong, and Thanks for All the Fish*

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Roland Schatz
Oracle Labs
Austria
roland.schatz@oracle.com

Jacob Kreindl
Johannes Kepler University Linz
Austria
jacob.kreindl@jku.at

Christian Häubl
Oracle Labs
Austria
christian.haeubl@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Dynamic languages rely on native extensions written in languages such as C/C++ or Fortran. To efficiently support the execution of native extensions in the multi-lingual GraalVM, we have implemented Sulong, which executes LLVM IR to support all languages that have an LLVM front end. It supports configurations with respect to memory-allocation and memory-access strategies that have different tradeoffs concerning safety and interoperability with native libraries. Recently, we have been working on balancing the tradeoffs, on supporting inline assembly and GCC compiler builtins, and on executing a complete libc on Sulong.

CCS CONCEPTS

• **Software and its engineering** → **Virtual machines**; **Interpreters**; **Dynamic compilers**;

KEYWORDS

Sulong, LLVM, GraalVM

ACM Reference Format:
Manuel Rigger, Roland Schatz, Jacob Kreindl, Christian Häubl, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Fish. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3191697.3191726

1 INTRODUCTION

The GraalVM [19, 20] is a multilingual virtual machine that can execute Ruby [14], JavaScript, R [16], and Python [21]. The individual language implementations are built on a common framework (called Truffle), which is written in Java [18]. Thus, a common compiler (called Graal) can optimize the execution of these languages during run time [2, 7, 15, 17].

Programmers of dynamic languages make frequent use of foreign function interfaces to call native extensions written in languages such as C/C++ or Fortran. To support native extensions efficiently, executing them on Truffle would be desirable, as the compiler could optimize across language boundaries and mitigate the overhead that other native function interfaces incur [4, 5]. To tackle this issue and to provide an efficient native function interface, we have implemented a Truffle language implementation as part of project *Sulong* [9].

Sulong's Truffle interpreter executes LLVM IR [6], which is a machine-independent format that specifies an instruction set resembling RISC assembly. By executing LLVM IR, Sulong can execute programs written in many languages, including C/C++ and Fortran. Recently, we have been working on enabling the use of native libraries in programs while still providing a partially safe execution, on supporting execution of a complete libc, and on supporting inline assembly and GCC builtins.

2 ARCHITECTURE

Figure 1 gives an overview of the Sulong system.

LLVM. Sulong's interpreter executes LLVM IR, which is part of the LLVM compiler framework [6] and can be produced by many language front ends. For example, LLVM's Clang front end can compile C/C++ code, Flang and GCC (with the DragonEgg plugin) can compile Fortran code to LLVM IR.

LLVM IR interpreter. Sulong's interpreter is written in Java using the Truffle language implementation framework. Truffle is a language implementation framework that facilitates the implementation of interpreters based on Abstract Syntax Trees (ASTs). Each operation is implemented as a node that returns its result in an `execute()` method. If a function is executed frequently, Truffle uses the Graal compiler to compile it to machine code, by inlining through all nodes' `execute()` methods (which is a form of *partial evaluation* [3]). As described below, Sulong's interpreter can be configured to use different strategies for handling allocations and accessing memory in the user program; depending on its strategy, we refer to Sulong as *Native Sulong*, *Safe Sulong*, or *Managed Sulong*.

Native Sulong. Native Sulong[1] prioritizes compatibility with native libraries over safety [10]. It supports passing objects allocated by the LLVM IR interpreter in the user program (e.g., an allocation by `malloc` in C) to machine-code functions. Native Sulong achieves this by allocating unmanaged memory for all user-program objects,

*The title is a reference to Douglas Adams' fourth book *So Long, and Thanks for All the Fish* of the Hitchhiker's Guide to the Galaxy.

<Programming'18> Companion, April 9–12, 2018, Nice, France

[1] Available at https://github.com/graalvm/sulong and shipped as part of the GraalVM at http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html.

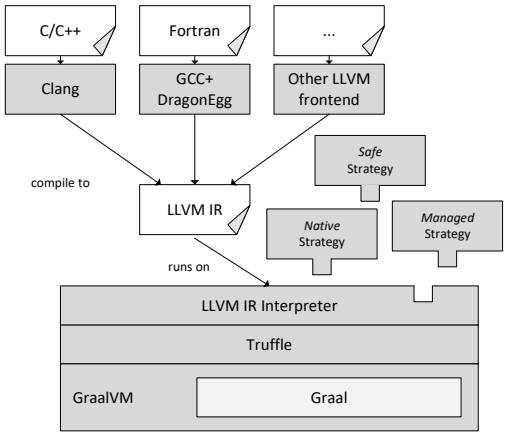

Figure 1: Overview of the Sulong system.

and by using raw pointers to memory as references. However, a drawback is that buffer overflows and other memory errors can corrupt memory content or crash the JVM.

Safe Sulong. Due to the seriousness of buffer overflows—they belong to the most dangerous software errors overall [1]—we have implemented Safe Sulong, which prioritizes safety over native compatibility [8, 12]. It allocates Java objects for allocations in the user program and detects buffer overflows through automatic checks of the underlying JVM, which then aborts execution. It can also detect other errors such as use-after-free errors, double-free errors, and accesses to non-existent variadic arguments and has found errors in open-source programs that state-of-the-art bug-finding tools failed to detect [12]. Additionally, we have shown that exposing run-time information tracked by Safe Sulong, such as object bounds or types, helps programmers to improve the robustness of their libraries [13]. The main drawback of Safe Sulong is that in addition to the application itself it also requires all libraries used by the application to be available as LLVM IR as well because it cannot pass Java objects to native libraries.

Managed Sulong. We are currently working on combining the advantages of Safe Sulong and Native Sulong, providing memory safety were possible, while retaining the ability to use native libraries. To this end, we are implementing *Managed Sulong*, which can handle both managed and native allocations. Additionally, we are experimenting with executing machine code on the Truffle platform, using either an x86 Truffle interpreter or by lifting machine code to LLVM IR, which could then be executed in a safe way.

3 EXECUTING LIBC ON SULONG

Most LLVM IR instructions can be mapped to similar Java operations or methods in Java's standard library. However, C/C++ projects contain elements such as inline assembly and compiler builtins, which resemble external calls after being compiled to LLVM IR. The call target, either an inline assembly snippet or a compiler builtin, needs to be implemented by Sulong's interpreter.

The x86 architecture offers about 1000 instructions, which leads to a high implementation effort even for this single instruction set architecture. Similarly, several compilers provide their own builtins and, for example, GCC alone provides several thousand builtins.

We have studied the usage of inline assembly [11] and builtins in GitHub projects to add support for the commonly used x86 inline assembly instructions and GCC builtins. As a result, Native Sulong can now execute system libraries such as libc (namely *musl libc*[2]) on the JVM, with the exception of system calls, for which it still relies on the operating system. As part of our future work, we also want to support the execution of libc with Safe Sulong and Managed Sulong.

4 CONCLUSION

We have introduced Sulong and its configurations that differ in their tradeoffs regarding safety and interoperability with native libraries. Further, we have discussed that Sulong can execute platform-specific elements in LLVM IR such as x86 inline assembly and compiler builtins. In the context of the GraalVM, Sulong is an important part to efficiently execute programs written in low-level languages such as C/C++ and Fortran.

ACKNOWLEDGMENTS

We want to thank Oracle Labs, which partly funded the authors from Johannes Kepler University Linz. We thank all Sulong, Truffle, and Graal contributors, the members of Oracle Labs, and the members of the Institute for System Software at the Johannes Kepler University Linz for their contributions. Finally, we thank Stefan Marr, Stephen Kell, Bram Adams, and David Leopoldseder for their assistance in analyzing the usage of inline assembly and/or GCC compiler builtins to help their implementation in Sulong.

REFERENCES

[1] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, Vol. 2. IEEE, 119–129.

[2] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[3] Yoshihiko Futamura. 1999. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

[4] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of DLS 2015*. 78–90.

[5] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2724525.2728790

[6] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[7] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[2] https://www.musl-libc.org/

[8] Manuel Rigger. 2016. Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. In *ECOOP 2016 Doctoral Symposium*. http://ssw.jku.at/General/Staff/ManuelRigger/ECOOP16-DS.pdf.

[9] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. 2016. Sulong - Execution of LLVM-based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '16)*. ACM, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3012408.3012416

[10] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[11] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. [n. d.]. An Empirical Analysis of x86-64 Inline Assembly in C Programs. In *Virtual Execution Environments (VEE 2018)*. https://doi.org/10.1145/3186411.3186418

[12] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. [n. d.]. Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*. https://doi.org/10.1145/3173162.3173174

[13] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Introspection for C and its Applications to Library Robustness. *The Art, Science, and Engineering of Programming* 2 (2018).

[14] Chris Seaton. 2015. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. Ph.D. Dissertation. University of Manchester.

[15] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/2989225.2989236

[17] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 165, 10 pages. https://doi.org/10.1145/2544137.2544157

[18] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. 13–14. https://doi.org/10.1145/2384716.2384723

[19] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[20] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[21] Wei Zhang. 2015. *Efficient Hosted Interpreter for Dynamic Languages*. Ph.D. Dissertation. University of California, Irvine.