Towards The Formalization of SPARK 2014 Semantics With Explicit Run-time Checks Using Coq*

Pierre Courtieu,Maria Virginia Aponte, Tristan Crolard
Conservatoire National des Arts et Metiers
Pierre.Courtieu@cnam.fr, maria-virginia.aponte_garcia@cnam.fr, tristan.crolard@cnam.fr

Zhi Zhang, Robby, Jason Belt, John Hatcliff
Kansas State University
zhangzhi@ksu.edu, robby@ksu.edu, belt@ksu.edu, hatcliff@ksu.edu

Jerome Guitton
AdaCore
guitton@adacore.com

Trevor Jennings
Altran
trevor.jennings@altran.com

ABSTRACT

We present the first steps of a broad effort to develop a formal representation of SPARK 2014 suitable for supporting machine-verified static analyses and translations. In our initial work, we have developed technology for translating the GNAT compiler's abstract syntax trees into the Coq proof assistant, and we have formalized in Coq the dynamic semantics for a toy subset of the SPARK 2014 language. SPARK 2014 programs must ensure the absence of certain run-time errors (for example, those arising while performing division by zero, accessing non existing array cells, overflow on integer computation). The main novelty in our semantics is the encoding of (a small part of) the run-time checks performed by the compiler to ensure that any well-formed terminating SPARK programs do not lead to erroneous execution. This and other results are mechanically proved using the Coq proof assistant. The modeling of on-the-fly run-time checks within the semantics lays the foundation for future work on mechanical reasoning about SPARK 2014 program correctness (in the particular area of robustness) and for studying the correctness of compiler optimizations concerning run-time checks, among others.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification, Formal Methods, Correctness Proofs; F.4.1 [**Mathematical Logic and Formal Language**]: Mathematical Logic—*mechanical theorem proving*

General Terms

Reliability, Security, Verification

Keywords

SPARK, Coq Proof Assistant, Formal Semantics, Machine-Verified Proof

1. MOTIVATION

We believe that the certification process of SPARK technology can be stressed by the use of formal semantics. Indeed, the software certification process as required by the DO-178-C [1] standard allows formal verification to replace some forms of testing. This is one of the goals pursued by the SPARK toolchain resulting from the Hi-Lite project [2]. On the other hand, the DO-333 supplement [3] (formal method supplement to DO-178-C) recommends that when using formal methods "all assumptions related to each formal analysis should be described and justified". As any formal static analysis must rely on the behavior of the language being analyzed, a precise and unambiguous definition of the semantics of this language becomes clearly a requirement in the certification process.

We aim also at strengthen the theoretical foundation of the GNATprove toolchain. The Ada reference manual [4] introduces the notion of *errors*. These correspond to error situations that must be detected at run time as well as erroneous executions that need not to be detected. In Ada, the former are detected by run-time checks (RTCs) inserted by the compiler. Both must be guaranteed never to occur during the process of proving SPARK (or Ada) subprograms within the GNATprove toolchain [5]. This can be ensured either by static analysis or by generating verification conditions (VCs) showing that the corresponding error situations never occur at that point in the subprogram. The generated VCs must be discharged in order to prove the subprogram. Tools within the GNATprove toolchain strongly rely on the completeness of this VCs generation process. Our semantics setting on top of a proof assistant open the possibility to formally (and mechanically) verify (to some extent) this completeness. In practice, since VCs are actually generated from the RTCs generated by the compiler, this completeness verification amounts to analyzing the RTCs inserted by the compiler in the abstract syntax tree produced by the GNAT compiler.

Finally, one of our long-term goals is to provide infrastructure that can be leveraged in a variety of ways to support machine-verified proofs of correctness of SPARK 2014 static analysis and translations. To this end, we will build a translation framework from SPARK 2014 to Coq, which puts in place crucial infrastructure necessary for supporting formal proofs of SPARK analysis.

*

HILT'13, November 10–14, 2013, Pittsburgh, PA, USA.

Together with the formal semantics of SPARK, it provides the potential to connect to the Compcert [6] certified compiler framework.

2. FORMALIZATION AND PROOF

2.1 SPARK Translation Toolchain

In the long path through the definition of complete semantics for SPARK 2014, a very important step is to build a tool chain allowing the experimentation of the behavior of these semantics on real SPARK 2014 programs. In the front end of this tool chain, as part of the Sireum analysis framework [7], we have developed a tool called Jago [8] that translates XML representation of the GNAT compiler's ASTs into a Scala-based representation in Sireum. This open-source framework enables one to build code translators and analysis tools for SPARK 2014 in Scala. Scala's blending of functional and object-oriented program styles have proven quite useful in other contexts for syntax tree manipulation and analysis. Integrated into the Jago is a translation of GNAT ASTs into the Coq proof assistant. In the backend of the tool chain, a certified interpreter encoding run-time checks for the Coq AST has been developed in Coq.

2.2 Formalizing Language Semantics

A major difference between SPARK and other programming languages is that its informal specification as given by the Ada reference manual requires the language to detect specific run-time error cases in order to enforce programs robustness. With Coq AST generated from SPARK by Jago, we are now developing its semantics capturing run-time errors in Coq and working toward adding more and more language features, such as procedure call, in our current formalization framework. At this early stage, our formal semantics consider only a small subset of SPARK 2014, and only a small subset of run-time errors. It performs appropriate run-time checks as they are specified by the SPARK and Ada reference manuals. We call these semantics, the *reference semantics*, and we implemented an interpreter certified with respect to them. Thus, our reference semantics can be both manually reviewed by SPARK experts, and also be experimented on real source code programs. For those who are interested, we have posted the source code of our formalization on GitHub [9].

2.3 Program Correctness Proof

In Coq, we have formalized a well-formed SPARK program as a well-typed, well-defined and well-checked program. A well-typed program has all its language constructs being consistent with respect to the typing rules and all its variables have correct in/out mode with respect to their reading and writing permissions. A well-defined program is a program with all its used variables initialized. And a well-checked program is a program having right checks inserted at the right places in AST trees. It is proved that for any well-formed SPARK programs in our formalized language subset, they will execute as we expect and will never exhibits undefined behavior.

3. RELATED WORK

Formal semantics were previously defined for SPARK Ada 83 in [10, 11]. This definition includes both the static and the dynamic semantics of the language and rely on a precise notation inspired by the Z notation. Formalizing the full SPARK subset was clearly a challenging task and the result is indeed quite impressive: more than 500 pages were required for the complete set of rules. However, these semantics were not executable (it was only given on paper) and no tool was used to check the soundness of the definition. Moreover, no property was proved using these semantics, and more importantly, run-time checks were only considered as side conditions on semantics rules without being formally described.

4. CONCLUSION AND FUTURE WORK

We have implemented a prototype tool chain from SPARK 2014 language subset to Coq and its semantical formalization and proof in Coq. The experiments on running certified interpreter show that our formalized SPARK semantics can capture the desired run-time errors. This's encouraging, but there are still a lot of work needed to do.

Our next step is to prove the correctness of optimizations that remove useless run-time checks. Our interpreted semantics are parameterized by the set of run-time checks to be performed. These semantics may be called with an incomplete set of run-time checks, and can evaluate in that case to an erroneous execution. A future work could be to formalize some optimizations actually performed by the GNAT compiler, and remove those useless run-time checks. The idea would be to prove these optimizations correct, namely to prove that those executions with *less* run-time checks behave exactly as those following the reference semantics, which perform systematically *all* the checks.

What's more, we are also interested in adding more SPARK language features, such as procedure call, pre/post aspects and loop invariant, to expand our current SPARK subset and make it more practical. These SPARK formalization work also paves the way for our further work on machine-verified proof of correctness of SPARK static analysis and translations.

Acknowledgements

The authors would like to thank Emmanuel Polonowski for valuable comments and suggestions on this work.

5. REFERENCES

[1] RTCA DO-178. Software Considerations in Airborne Systems and Equipment Certification. RTCA and EUROCAE, 2011.

[2] AdaCore Hi-Lite project. `http://www.open-do.org/projects/hi-lite/`

[3] RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A. RTCA and EUROCAE, 2011.

[4] Ada reference manual. `http://www.ada-auth.org/standards/ada12.html`

[5] AdaCore Gnatprove tool. `http://www.open-do.org/projects/hi-lite/gnatprove/`

[6] X. Leroy. Formal Verification of a Realistic Compiler. In *Communications of the ACM*, 52(7):107–115, 2009.

[7] Sireum software analysis platform. `http://www.sireum.org`

[8] Jago translation tool. `https://github.com/sireum/bakar/tree/master/sireum-bakar-jago`

[9] Coq code for SPARK 2014 language subset formalization. `https://github.com/sireum/bakar/tree/master/sireum-bakar-formalization`

[10] W. March. *Formal Semantics of SPARK - Static Semantics*. October, 1994.

[11] I. O'Neill. *Formal Semantics of SPARK - Dynamic Semantics*. October, 1994.