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

P. Courtieu[1], V. APonte[1], T. Crolard[1], Zhi Zhang[2], Robby[2],
J. Belt[2], J. Hatcliff[2], J. Guitton[3], T. Jennings[4]

[1] CNAM
[2] Kansas State University
[3] AdaCore
[4] Altran

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 [Spark2014]. 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 well-typed and well initialized 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.

1 Motivation

1.1 Certification process of SPARK technology can be stressed by the use of formal semantics

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 [Hi-Lite]. On the other hand, the DO-333 supplement [2] (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.

1.2 Enforce the theoretical foundation of the GNATprove toolchain

The Ada reference manual [3] introduces the notion of *errors*, corresponding in particular to error situations that must be detected at run time, and as well to erroneous

executions that need not to be detected. In Ada, the first ones 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 [4]. 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.

1.3 Formal proofs of analyzers and certified compilers

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

2 Description of the work

2.1 The verifying semantics 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 [5], we have developed a tool called Jago [6] 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 of 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 is being developed in Coq.

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.

2.2 Properties

It is proved that well typed and well formed SPARK toy programs will not generate some class of errors at execution.

3 Conclusion and future work

3.1 Validation of the semantics

The usual question about formal semantics is the following: how certain are we that the formalized semantics are the right ones? There are two answers: first the semantics should be read and validated by Ada and SPARK experts and second we provide an interpreter proved correct with respect to the semantics, on which we can perform tests to compare the results of the interpreter and compiled program.

3.2 Going further: proving 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 consisting in the removal of 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.

4 Related work

Formal semantics were previously defined for SPARK Ada 83 in [7, 8]. 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.

References

1. RTCA DO-178, "Software Considerations in Airborne Systems and Equipment Certification", RTCA and EUROCAE, 2011.
2. RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A, RTCA and EUROCAE, 2011.

3. http://www.ada-auth.org/standards/ada12.html
4. http://www.open-do.org/projects/hi-lite/gnatprove/
5. http://www.sireum.org
6. https://sireum.assembla.com/code/sireum-bakar/git/nodes/master/sireum-bakar-jago
7. Formal Semantics of SPARK - Static Semantics, William Marsh, October, 1994
8. Formal Semantics of SPARK - Dynamic Semantics, Ian O'Neill, October, 1994