Semantical Formalization, Run-Time Checks Proof and Verification for SPARK 2014

Authors

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

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 Introduction

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 [**?**] 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 [**?**]. On the other hand, the DO-333 supplement [**?**] (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 also aim to strengthen the theoretical foundation of the GNATprove toolchain. The Ada reference manual [**?**] 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 [**?**]. 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. Together with the formal semantics of SPARK, it provides the potential to connect to the Compcert [**?**] certified compiler framework.

2 Overview

In the long path through the definition of complete semantics for SPARK 2014, a very important step is to build a tool chain to make it possible in the future to be integrated into SPARK 2014 tool set. Now most of the formalization work are not really used or adopted by a real programming language partially because of the big gap between formalization and its real application. So we build a prototype of the tool chain from SPARK 2014 to Coq and build a bridge between SPARK formalization and its real application in SPARK GnatProve tool set. For the users of SPARK programming language, it also helps to convince them why SPARK is safety-critical programming language by the experimentation of the behavior of SPARK semantics on real SPARK 2014 programs.

Insert An Overview Graph

2.1 The Frontend of The Tool Chain From SPARK to Coq

In the front end of this tool chain, Gnat2XML, developed by AdaCore, translates SPARK programs to a fully resolved Abstract Syntax Tree (AST) XML representation with an accompanying XML schema. As part of the Sireum analysis framework [5], we have furtherly developed a tool called Jago [4] 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. Scalas blending of functional and object-oriented program styles have proven quite useful in other contexts for syntax tree manipulation and analysis. Integrated into Jago are two kinds of translations: (1) type translation to translate Gnat2XML-generated XML schema to (inductive) type definition in Coq; (2) program translation to translate Gnat2XML-generated AST XML representation into Coq based representations.

2.2 SPARK 2014 Formalization and Proof in Coq

With Coq inductive type definition for SPARK AST syntax produced by Jago type translator, formal semantics encoding run-time checks for SPARK has been developed within Coq, which is refered as SPARK reference semantics. Besides, a formal semantics for SPARK AST extended with run-time check flags are defined, where run-time checks are performed only if the appropriate check flags are set for the operations. And an AST translator from a SPARK AST to a run-time check flagged AST is provided and proved correct with respect to the SPARK reference semantics.

2.3 Run-Time Checks Comparison

To verify the run-time check flags that are inserted by GnatPro frontend, a run-time check comparison function is developed to match the GnatPro generated checks against the expected checks required by our formalized SPARK reference semantics, and report any mismatchs. For easy debug, any check mismatching information will be mapped back to the SPARK source location.

3 Formalization for A Subset of SPARK 2014 Semantics

3.1 Syntax of SPARK 2014 Subset

```
Inductive expr: Type :=
  | Literal: astnum → literal → expr
  | Name: astnum → name → expr
  | BinOp: astnum → binary_operator → expr → expr → expr
  | UnaOp: astnum → unary_operator → expr → expr
with name: Type :=
  | Identifier: astnum → idnum → name
  | Indexed_Component: astnum → astnum → idnum → expr → name
  | Selected_Component: astnum → astnum → idnum → idnum → name.

Inductive stmt: Type :=
  | Null: smt
  | Assignment: astnum → name → expr → stmt
  | If: astnum → expr → stmt → stmt → stmt
  | While: astnum → expr → stmt → stmt
  | Call: astnum → astnum → procnum → list expr → stmt
  | Sequence: astnum → stmt → stmt → stmt.

Inductive type_decl: Type :=
  | Subtype: astnum → typenum → type → range → type_decl
  | Derived_Type: astnum → typenum → type → range → type_decl
  | Integer_Type: astnum → typenum → range → type_decl
  | Array_Type: astnum → typenum → type → type → type_decl
  | Record_Type: astnum → typenum → list (idnum*type) → type_decl.
```