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 core 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 but nontrivial part of) the run-time checks performed by the compiler to ensure that well-formed 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

1.1 Background

1.2 Motivations

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.

1.3 Contributions

The major contributions in this paper are:

- Formalize a core subset of SPARK 2014 Language in Coq Proof Assistant and run-time checks are formalized as an important integrant to provide run-time errors free SPARK programs.
- Verify a nontrivial subset of run-time check flags generated by Gnat front end, and at the same time it also helps to patch Gnat front end in case of any implementation errors resulting in missing or wrong run-time check flags.
- Optimize run-time checks and prove its soundess.
- Build a tool chain from SPARK to Coq formalization and make it possible to integrate our SPARK formalization work into GnatProve tool chain.
- Run-time check evaluation on real SPARK programs.

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 allowing its application in formal proof and verification on real SPARK 2014 programs. It also makes it possible to integrate our SPARK formalization work into GnatProve toolchain to serve as a trusted soundness verification for run-time check flags generated by Gnat front end.

SPARK Translation Toolchain 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.

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.

Run-Time Checks Verification In GnatProve tool chain, the run-time check flags set by its front end during semantic analysis will trigger the corresponding check in its back end by applying formal verification method. To verify these run-time check flags, a check verification function is developed to match them against the run-time checks required by SPARK reference semantics, and report any mismatchs. For easy debug, any check mismatching information will be mapped back to the SPARK source location.

3 SPARK Formalization

3.1 Syntax of SPARK 2014 Subset

The subset of SPARK 2014 that we have formalized is significant, which includes array/record (non-nested) and procedure calls. Furthermore, it also supports some intersting SPARK / Ada language structures, such as nested procedures and subtypes.

SPARK AST syntax is represented with inductive type definitions in Coq. And each AST node is annotated with an unique AST number, which will be used to record the type for each ast node, or it can be used later to track back to the SPARK source program when an run time error is detected, or it can be used to locate the position in source program where the run time check flags inserted by Gnat front end is incorrect.

Here, we list some of SPARK language structures and show how we formalize them in Coq. Expression (*expr*) can be literal, unary expression, binary expression or name, and each expression is annotated with an AST number (*astnum*), which is represented by natural number. For type *name*, it can be identifier, indexed component or selected component. Indexed component is constructed with the constructor *Indexed_Component*, whose first *astnum* denotes the indexed component and the second *astnum* denotes the prefix expression represented by *idnum* and *expr* is for index expression.

```
Inductive expr: Type :=
| Name: astnum → name → expr
| ...
with name: Type :=
| Identifier: astnum → idnum → name
| Indexed_Component: astnum → astnum → idnum → expr → name
| Selected_Component: astnum → astnum → idnum → idnum → name.
```

For procedure *Call* in statement *stmt*, its first *astnum* is the AST number for the procedure call statement, and the second *astnum* is the AST number for the called procedure represented by *procnum* followed by a list of arguments of type *list expr*.

```
Inductive stmt: Type :=
| Assignment: astnum → name → expr → stmt
| Call: astnum → astnum → procnum → list expr → stmt
| ...
```

Range constrainted scalar types are used commonly in SPARK programs, they can be declared with either subtype declaration, derived type definition, or integer type definition. A *Subtype* declares a subtype, represented with *typenum*, of some previously declared *type* with an additional *range* constraint (e.g. subtype T is Integer range 1 .. 10). A *Derived_Type*, whose name is represented by *typenum*, defines a derived type whose characteristics are derived from those of a parent *type* with an additional *range* constraint (e.g. type U is new Integer range 1 .. 10). A *Integer_Type* defines a new integer type, represented with *typenum*, with an additional *range* constraint (e.g. type W is range 0 .. 10).

Array_Type and *Record_Type* are constructors for defining aggregate data types array and record.

```
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.
```

3.2 Run-Time Check Flags

In SPARK, run time checks flags are automatically inserted at SPARK AST by the front end during semantic analysis, and their corresponding run time checks are then discharged by formally verifying their generated verification conditions with the Gnat-Prove tool chain. So SPARK can guarantee the absence of run time errors for developing safety critical systems.

For our formalized SPARK 2014 subset, the following check flags are sufficient, which are enforced on the expression nodes.

– Do_Overflow_Check: This flag is set on an operator where its operation may cause overflow, such as binary operators *(+, -, *, /)*, unary operator *(-)* and type conversion from one base type to another when the value of source base type falls out of domain of the target base type.

– Do_Division_Check: This flag is set on division operators, such as *(/, mod, rem)*, to indicate a zero divide check.
– Do_Range_Check: This flag is set on an expression which appears in a context where range check is required, such as right hand side of an assignment, subscript expression in an indexed component, argument expressions for a procedure call and initialization value expression for an object declaration.

3.3 Semantical Formalization With Run-Time Checks

The major semantical difference between SPARK and other programming languages is that verification for absence of run time errors are required by the languge itself. So in our semantical formalization for SPARK language, run time checks is an important integrant and they are always performed at appropriate points during the language semantic evaluation. The program will be terminated with a run time error message once any of its run time checks fails during the program evaluation.

Value In SPARK semantics, return value for an expression evaluation can be either a normal value (basic or aggregate value) or a run time error status detected during expression evaluation. Similarly, for a well-formed SPARK program, it should either terminate in a normal state or a detected run time error, which is expected to be detected and raised during program execution.

```
Inductive Return (A: Type): Type :=
| Normal: A → Return A
| Run_Time_Error: error_type → Return A.
```

Run Time Check Evaluation A small but significant subset of SPARK run time checks are formalized in Coq, including overflow check, division check and range check. Overflow checks are performed to check that the result of a given orithmetic operation is within the bounds of the base type, division checks are performed to prevent divide being zero, and range checks are performed to check that the evaluation value of an expression is within bounds of its target type with respect to the context where it appears. A small fragment for overflow check formalization in Coq is:

```
Inductive overflow_check_bin:binop → value → value → status → Prop :=
| Do_Overflow_Check_On_Binops: ∀ op v1 v2 v,
    op = Plus ∨ op = Minus ∨ op = Multiply ∨ op = Divide →
    Val.binary_operation op v1 v2 = Some (BasicV (Int v)) →
    (Zge_bool v min_signed) && (Zle_bool v max_signed) = true →
    overflow_check_bin op v1 v2 Success
| ...
```

Now we only model the 32-bit singed integer for SPARK program, where Coq integer (Z) is used to represent this integer value with a range bound between *min_signed* and *max_signed*. This integer range constraint is enforced through the above overflow check semantics when we define the semantics for the language. As we can see, overflow checks are required only for binary operators *(+, -, *, /)* among the set of binary operators in our formalized SPARK subset. And it returns either *Success* or *Exception* with overflow signal.

Expression Evaluation In an expression evaluation, for an arithmetical operation, run time checks are always performed according to the checking rules required for the arithmetical operators in SPARK reference manual, and a run time error returns whenever the check fails, otherwise, a normal operation result is returned. Further checks on the normal result value maybe required depending on the context where the expression appears. One such example is that range check should be performed on the index expression value before it can be used as an index value for an indexed component.

The following is a snippet of how the expression evaluation is formalized in Coq with run time checks enforced during its semantics evaluation. For a binary expression (Binop ast_num op e1 op e2), if both e1 and e2 are evaluated to some normal values, then all necessary run time checks required for the operator *op* are performed, e.g. overflow check for + and both overflow check and division check for /, and a normal binary operation result is returned when the checks succeed. In name evaluation for indexed component, an additional range check is required to be performed according to the target type of the array, which is fetched from a preconstructed symbol table.

```
Inductive eval_expr: symboltable → stack → expr → Return value → Prop :=
| Eval_Binop: ∀ st s e1 v1 e2 v2 ast_num op v,
    eval_expr st s e1 (Normal v1) →
    eval_expr st s e2 (Normal v2) →
    do_run_time_check_on_binop op v1 v2 Success →
    Val.binary_operation op v1 v2 = Some v →
    eval_expr st s (Binop ast_num op e1 e2) (Normal v)
| ...
with eval_name: symboltable → stack → name → Return value → Prop :=
| Eval_Indexed_Component_RTE: ∀ st s e msg ast_num x_ast_num x,
    eval_expr st s e (Run_Time_Error msg) →
    eval_name st s (Indexed_Component ast_num x_ast_num x e)
                   (Run_Time_Error msg)
| Eval_Indexed_Component: ∀ st s e i x_ast_num t l u x a v ast_num,
    eval_expr st s e (Normal (BasicV (Int i))) →
    fetch_exp_type x_ast_num st = Some (Array_Type t) →
    extract_array_index_range st t (Range l u) →
    do_range_check i l u Success →
    fetchG x s = Some (AggregateV (ArrayV a)) →
    array_select a i = Some v →
    eval_name st s (Indexed_Component ast_num x_ast_num x e)
                   (Normal (BasicV v))
| ...
```

Statement Evaluation In the context of statement evaluation, range checks will be enforced during statement evaluation for both assignments and procedure calls. For the case of assignment evaluation, range check for the right hand side expression of the assignment is enforced if the target type of the left side of the assignment is some range constrainted type. For the case of procedure calls, range checks are required to be enforced on arguments against the type of the IN and IN OUT formal parameters when passing in the arguments before running the called procedure, and do range checks on values of OUT and IN OUT formal parameters on the procedure return.

For a normal assignment evaluation, first evaluate its right hand side expression *e*, if it returns a normal value, then fetch the type of its left hand side name *x*, perform a range check before updating its value if it's a range constrainted type.

```
Inductive eval_stmt: symboltable → stack → stmt → Return stack → Prop :=
| Eval_Assignment: ∀ st s e v x t l u s1 ast_num,
    eval_expr st s e (Normal (BasicV (Int v))) →
    fetch_exp_type (name_astnum x) st = Some t →
    extract_subtype_range st t (Range l u) →
    do_range_check v l u Success →
    storeUpdate st s x (BasicV (Int v)) s1 →
    eval_stmt st s (Assignment ast_num x e) s1
| ...
```

Declaration Evaluation For an object declaration, range check is required for its initialization expression if the type of the object being declared is range constrainted. Type declaration and procedure declaration should have no effect on the final stack.

4 Run-Time Checks Verification

In SPARK GnatProve tool chain, formal verification for the absence of run time errors for SPARK program relies on the run time check flags that are initially generated and inserted to SPARK AST by Gnat front end. Gnat front end is expected to place the correct run time check flags during its static semantic analysis for SPARK AST. But the fact is that Gnat front end itself is not formallly verified, and that's why it's often the case that run time check flags are missing or misplaced in SPARK AST. It leads to the problem that a SPARK program proved to be free of run-time errors can still gets into a run time error status because of the missing or incorrect checks placed by Gnat front end. So it's meaningful to formally verify the run-time check flags with a formal verification way to make the GnatProve tool chain sound.

To verify the run time check flags, a check-flag-annotated SPARK language and its corresponding semantics are formally defined, where each expression and subexpression nodes are attached with a set of run time check flags. Then a run-time check generator simulating Gnat front end is defined and formally proved correct with respect to the SPARK reference semantics. It formalizes the run time-check generation procedure and generates run-time check flags according to the SPARK semantics by transforming a SPARK program into a check-flag-annotated SPARK program. Finally, the run-time check flags generated by Gnat front end are verified against our formally proved run-time check flags.

4.1 Check-Flag-Annotated SPARK Language

Syntax Check-flag-annotated SPARK language is the same as the SPARK language except that each expression node is annotated with a set of run-time check flags *check_flags*, which denotes explicitly what kinds of run time checks need to be verified during expression evaluation on the annotated expression.

```
Inductive expr_x: Type :=
| Name_X: astnum → name_x → check_flags → expr_x
| ...
with name_x: Type :=
| Indexed_Component_X: astnum → astnum → idnum → expr_x → check_flags → name_x
| ...
```

Semantics SPARK reference semantics is formalized with run-time checks being always performed according to the run-time check requirements by SPARK reference manual. While in the check-flag-annotated SPARK semantics, run-time checks are triggered to be performed only if the corresponding check flags are set for the attached expression node. For example, in the following name evaluation for check-flag-annotated indexed component, range check is required only if the *Do_Range_Check* flag is set for the index expression *e*, otherwise, the index value *i* is used directly as array indexing without going through range check procedure *do_range_check*.

```
Inducitve eval_name_x: symboltable_x → stack → name_x → Return value → Prop :=
| Eval_Indexed_Component_X: ∀ e cks1 cks2 st s i x_ast_num t l u x a v ast_num,
    exp_check_flags e = cks1 ++ Do_Range_Check :: cks2 →
    eval_expr_x st s (update_check_flags e (cks1++cks2)) (Normal (BasicV (Int i))) →
    fetch_exp_type_x x_ast_num st = Some (Array_Type t) →
    extract_array_index_range_x st t (Range_X l u) →
    do_range_check i l u Success →
    fetchG x s = Some (AggregateV (ArrayV a)) →
    array_select a i = Some v →
    eval_name_x st s (Indexed_Component_X ast_num x_ast_num x e nil) (Normal (BasicV v))
| ...
```

4.2 Run-Time Checks Generator

Check Generator Run-time check generator is a translator from a SPARK program to a check-flag-annotated SPARK program by generating run-time check flags according to the run-time checking rules required by SPARK reference manual and inserting these check flags at the corresponding AST node. In expression check generator *compile2_flagged_exp*, *check_flags* denote the run-time checks on the expression required by its context, such as range check for expression used in indexed component, and other expression check flags are generated according to the operation type to be performed by the expression.

```
Inductive compile2_flagged_exp: symboltable → check_flags →
                                expression → expression_x → Props
```

Soundness Proof Run-time check generator is proved sound with respect to the SPARK reference semantics and check-flag-annotated SPARK semantics. For an expression *e*, if it's evaluated to some value *v* in state *s* by SPARK reference semantic evaluator *eval_expr*, and *e'* is the check-flag-annotated expression generated from *e* by expression check generator *compile2_flagged_exp*, then *e'* should be evaluated to the same value *v* in check-flag-annotated SPARK semantic evaluator *eval_expr_x*. Similar soundness proof has been done for statement check generator.

```
Lemma expression_checks_soundness: ∀ e e' st st' s v,
  eval_expr st s e v →
    compile2_flagged_exp st nil e e' →
      compile2_flagged_symbol_table st st' →
        eval_expr_x st' s e' v.

Lemma statement_checks_soundness: ∀ st s stmt s' stmt' st',
  eval_stmt st s stmt s' →
    compile2_flagged_stmt st stmt stmt' →
      compile2_flagged_symbol_table st st' →
        eval_stmt_x st' s stmt' s'.
```

4.3 Run-Time Checks Optimization

Optimization Strategy We have formalized some simple but helpful optimizations for literal operations and remove those checks that can be obviously verified at compilation time, which is also the optimization strategy taken by the Gnat front end.

Soundness Proof The idea to prove the correctness of these optimizations is to prove that SPARK program executions with optimized run-time checks behave exactly the same as those following the SPARK reference semantics, which perform systematically all the checks.

4.4 Run-Time Checks Verification

One of the major goals of our formalization work for SPARK language is to verify the soundness of run-time check flags produced by Gnat front end. It's done by comparing the Gnat generated run-time check flags with the expected ones provided by formally verified check generator with respect to the SPARK semantics. Run-time check flags are verified to be correct if they are superset of the expected ones required by the SPARK reference manual.

5 Evaluation

5.1 Run-Time Checks Generator Function

Check generator function for expression:

```
Function compile2_flagged_exp_f (st: symboltable)
                                (checkflags: check_flags)
                                (e: expression): expression_x
```

5.2 Application To SPARK 2014 Programs

6 Related Work

Formal semantics were previously defined for SPARK Ada 83 in [**?**,**?**]. 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.

7 Conclusions and Future Work