Formalization of SPARK Subset in Coq

Zhi Zhang

Conservatoire National des Arts et Metiers

Pierre Courtieu
Maria Virginia Aponte
Tristan Crolard

Kansas State University

Zhi Zhang
Robby
Jason Belt
John Hatcliff

AdaCore

Jerome Guitton

Altran

Trevor Jennings

Funding *: this work is supported by the National Science Foundation under Grant # 0644288 and by the US Air Force Office of Scientific Research (AFOSR) under contract FA9550-09-1-0138*

Outline

- Motivation
- Formalization Work
- Demo
- Future Work

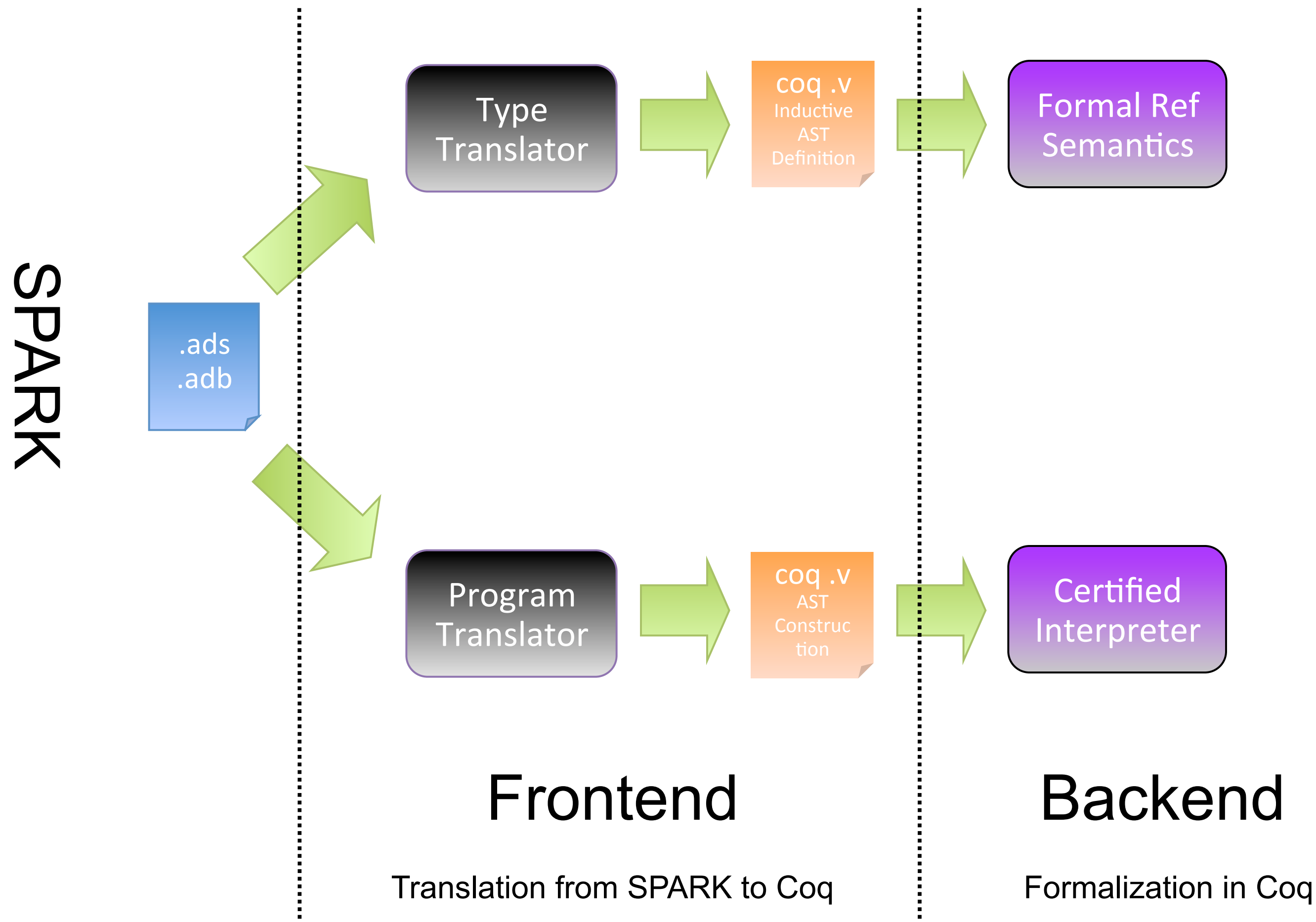
Our Work

- Formalize dynamic semantics of SPARK subset in Coq
  - Perform necessary run time checks
  - Prove correctness for well-formed programs
  - Build a tool chain from SPARK to Coq

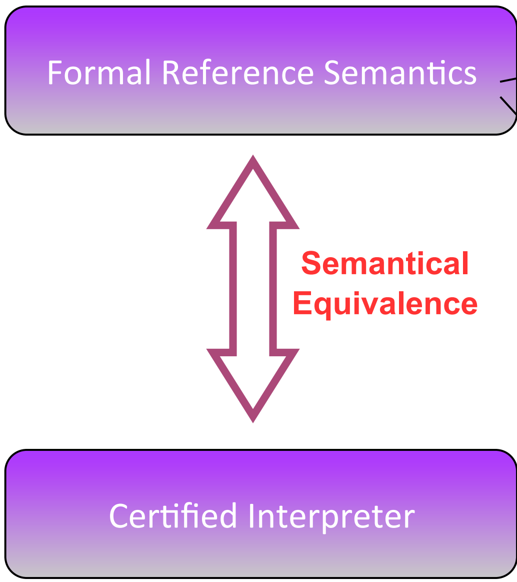
Motivation

- Define Formal Semantics for SPARK
  - Basis for SPARK certification technology
- Strengthen GNATprove Toolchain
  - Justify "*all necessary RTC are in inserted*"
- Provide SPARK Infrastructure for
  - Machine-verified proofs of static analysis
  - Certified SPARK frontend for CompCert

SPARK 2014 To Coq Tool Chain



SPARK 2014 To Coq Tool Chain



✓ **Verify Run Time Checks (RTC)**

It's proved that for any SPARK program, if all necessary RTC are inserted, then it should behave as prescribed by the reference semantics

✓ **Verify SPARK Program Correctness**

It's proved that for any well-formed terminating SPARK program, it will terminate in a valid state

Certified Interpreter

- For formal method practicer
  - validate formalized SPARK semantics experimentally by testing
- For users
  - familiarize oneself with the SPARK 2014 semantics, and
  - help to fix the program if the program exhibits undefined behavior

SPARK Subset Language

```
expr  ::=  c
         | x
         | expr bop expr
         | uop expr

stmt  ::=  x := expr
         | if expr then stmt end if
         | while expr loop stmt end loop
         | stmt; stmt
```

SPARK Subset

```
Inductive expr: Type :=
    | Literal            : constant → expr
    | Identifier         : id → expr
    | Binary_Operation   : binary_operator → expr → expr → expr
    | Unary_Operation    : unary_operator → expr → expr

Inductive stmt: Type :=
    | Assignment : id → expr → stmt
    | If         : expr → stmt → stmt
    | While_Loop : expr → stmt → stmt
    | Sequence   : stmt → stmt → stmt
```

Inductive Definition in Coq

Example

```
If (N <= 1) then
   Result := false;
end if;
```

SPARK Code

```
If (Binary_Operation Less_Than_Or_Equal (Identifier N) (Literal (Integer_Literal 1)))
 (Assignment Result (Literal (Boolean_Literal false)))
```

SPARK AST in Coq

Program States

```
Inductive state: Type :=
    | Normal  : store → state
    | Run_Time_Error : state
    | Unterminated        : state
    | Abnormal             : state
```

Classification of Errors (Ada RM):

- Errors that are required to be detected prior to run time
- Errors that are required to be detected at run time
- Bounded errors
- Erroneous execution

In the future, we would refine the abnormal state into these more precise categories.

Run Time Check Semantics

Checking Rules Mark *What* and *Where* to Check

- ***Do_Division_Check***
  - This flag is set on a division operator (/, mod, rem) to indicate that a zero divide check is required;
- ***Do_Overflow_Check***
  - This flag is set on an operator where an overflow is required on the operation;

Run Time Check Semantics

Run Time Check Flags

```
Inductive run_time_checks: Type :=
    | Do_Division_Check  : run_time_checks
    | Do_Overflow_Check : run_time_checks
```

Semantics for Run Time Checks

```
Inductive do_check: binary_operator → value → value → bool → Prop :=
    | Do_Overflow_Check_On_Plus : forall v1 v2 b,
        (((v1 + v2) >= min_signed)  && ((v1 + v2) <= max_signed)) = b →
        do_check Plus (Int v1) (Int v2) b
    | Do_Division_And_Overflow_Check_on_Divide : forall v1 v2 b,
        ((v2 <> 0) && ((v1 / v2) >= min_signed)  && ((v1 / v2) <= max_signed)) = b →
        do_check Divide (Int v1) (Int v2) b
    …
```

Example

X / Y	Binary_Operation Divide (Identifier X) (Identifier Y)	(Y <> 0) and ((-2^31) <= (X / Y) <= (2^31 -1))
SPARK Expr	SPARK AST in Coq	Run Time Checking (32-bit singed integer)

Language Semantics

- Formal Reference Semantics

  ref_eval: state → procedure → state → Prop

- Certified Interpreter

  certified_eval (s: state) (p: procedure): state

- Semantical Equivalence

  ref_eval s f t ↔ certified_eval (s, f) = t

Certify GNATprove Frontend

Do-178-C Standard

- It allows formal verification to replace some forms of testing in the software certification process;

Do-333 Supplement (formal method supplement to Do-178-C)

- It recommends that when using formal methods *all assumptions related to each formal analysis be described and justified*;

Certify GNATprove Frontend

- Ideally, we want certified frontend, but …
- GNATprove relies on *uncertified GNAT Compiler* to insert the necessary run time check flags
- We want to *formalize and certify* these run time checks to make sure that GNATprove compiler inserts the appropriate run time checks at appropriate places

Check Flags Generator

Run Time Checking Rules

```
Inductive check_flags: expr → run_time_check_set → Prop :=
    | CF_Literal : forall c,
          check_flags (Literal c) nil
    | CF_Plus : forall e1 e2,
          check_flags (Binary_Operation Plus e1 e2)
                      (Do_Overflow_Check :: nil)
    | CF_Divide : forall e1 e2,
          check_flags (Binary_Operation Divide e1 e2)
                      (Do_Division_Check :: Do_Overflow_Check :: nil)
    …
```

Example

SPARK Expr	SPARK AST in Coq	Check Flags
2	Literal (Integer_Literal 2)	nil
X / Y	Binary_Operation Divide (Identifier X) (Identifier Y)	[Do_Division_Check, Do_Overflow_Check]

Semantics With Flagged Checks

- Formal Reference Semantics

  do complete checks

  ref_eval: state → procedure → state → Prop

- Semantics With Flagged Checks

  do selected checks

  ref_eval': *check_points* → state → procedure → state → Prop

- Semantical Correctness

  ref_eval' *checks* s f t → ref_eval s p t

  (where checks are checks generated by the checking rules)

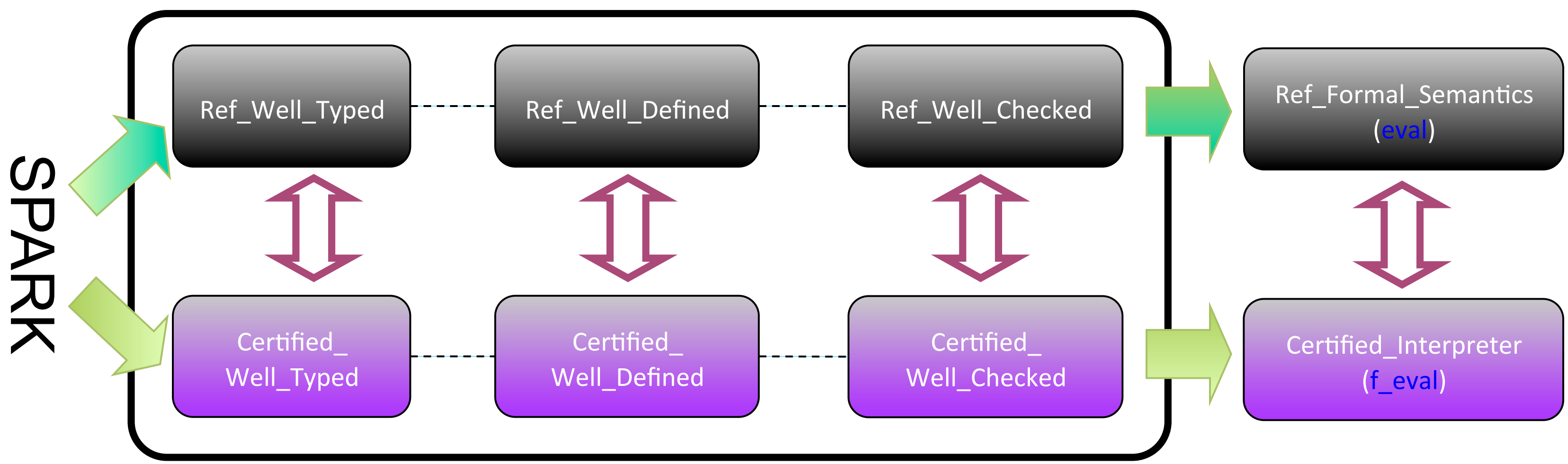
Static Analysis

- Well-Typed
  - programs are correct with respect to the typing rules
  - values with correct in/out mode
- Well-Defined
  - all used variables have been initialized
- Well-Checked
  - necessary checks are inserted in the AST tree

Program Correctness Proof

Machine-verified SPARK Program Correctness

Theorem Program_Correctness: forall f,
    Ref_Well_Typed f →
    Ref_Well_Defined f →
    Ref_Well_Checked f →
    (forall s,
        (exists t s', ref_eval s f t /\ (t = Normal s' \/ t = Run_Time_Error) \/
        (forall k, certified_eval s f = Unterminated)
)



Demo

- Show How the tool chain works
  - Translation from SPARK to Coq
- Run the certified interpreter
  - Its result should be the same as the reference semantics
  - It can capture necessary run time errors
- Static analysis
  - Run certified static analysis for checking program's well-formness
  - All well-formed terminating program will behave correctly

Example 1

```
procedure Test_for_Coq
is
   N: Integer := 25;
   Result: Boolean;
   I: Integer;
   X: Integer;
begin
   Result := true;
   if N <= 1 then
      Result := false;
   end if;
   I := 2;
   while I*I <= N loop
      X := N / I;
      if N = X * I then
         Result := false;
      end if;
      I := I + 1;
   end loop;
end Test_for_Coq;
```

SPARK

```
Definition f_prime :=
Procedure 3 (
    mkprocedure_body 4
     (** Procedure Name *)
     (** Test_for_Coq *) 1
     (** Specification *)
     (nil)
     (** Parameters *)
     (nil)
     (** Variable Declarations *)
     (
     mkobject_declaration 5 (** N *) 1 1 (Some (E_Literal 6 (Integer_Literal 25))) ::
     mkobject_declaration 7 (** Result *) 2 2 None ::
     mkobject_declaration 8 (** I *) 3 1 None ::
     mkobject_declaration 9 (** X *) 4 1 None :: nil)
     (** Procedure Body *) (
      S_Sequence 10 (
       S_Assignment 11 ((** Result *) 2) (E_Literal 12 (Boolean_Literal true)) ) (
       S_Sequence 13 (
        S_If 14 (E_Binary_Operation 15 Less_Than_Or_Equal (E_Identifier 16 (** N *) 1) (E_Literal 17 (Integer_Literal 1))) (
          S_Assignment 18 ((** Result *) 2) (E_Literal 19 (Boolean_Literal false))
         ) ) (
        S_Sequence 20 (
         S_Assignment 21 ((** I *) 3) (E_Literal 22 (Integer_Literal 2)) ) (
         S_While_Loop 23 (E_Binary_Operation 24 Less_Than_Or_Equal
             (E_Binary_Operation 25 Multiply (E_Identifier 26 (** I *) 3) (E_Identifier 27 (** I *) 3)) (E_Identifier 28 (** N *) 1)) (
           S_Sequence 29 (
            S_Assignment 30 ((** X *) 4) (E_Binary_Operation 31 Divide (E_Identifier 32 (** N *) 1) (E_Identifier 33 (** I *) 3)) ) (
            S_Sequence 34 (
             S_If 35 (E_Binary_Operation 36 Equal
                 (E_Identifier 37 (** N *) 1) (E_Binary_Operation 38 Multiply (E_Identifier 39 (** X *) 4) (E_Identifier 40 (** I *) 3)))
               S_Assignment 41 ((** Result *) 2) (E_Literal 42 (Boolean_Literal false))
              ) ) (
             S_Assignment 43 ((** I *) 3) (E_Binary_Operation 44 Plus (E_Identifier 45 (** I *) 3) (E_Literal 46 (Integer_Literal 1)))
          ) ) ) )
     )).
```

SPARK AST in Coq

1) Translation from SPARK to AST in Coq
2) Check program's well-formedness with certified static analyses (well-typed, well-defined, well-checked)
3) Run certified interpreter on the Coq AST

Example 2

```
procedure Test_for_Coq
is
   N: Integer := 25;
   Result: Boolean;
   I: Integer;
   X: Integer;
begin
   Result := true;
   if N <= 1 then
      Result := false;
   end if;
   I := 0; -- Error !
   while I*I <= N loop
      X := N / I;
      if N = X * I then
         Result := false;
      end if;
      I := I + 1;
   end loop;
end Test_for_Coq;
```

SPARK

```
Definition f_prime :=
Procedure 3 (
    mkprocedure_body 4
     (** Procedure Name *)
     (** Test_for_Coq *) 1
     (** Specification *)
     (nil)
     (** Parameters *)
     (nil)
     (** Variable Declarations *)
     (
     mkobject_declaration 5 (** N *) 1 1 (Some (E_Literal 6 (Integer_Literal 25))) ::
     mkobject_declaration 7 (** Result *) 2 2 None ::
     mkobject_declaration 8 (** I *) 3 1 None ::
     mkobject_declaration 9 (** X *) 4 1 None :: nil)
     (** Procedure Body *) (
      S_Sequence 10 (
       S_Assignment 11 ((** Result *) 2) (E_Literal 12 (Boolean_Literal true)) ) (
       S_Sequence 13 (
        S_If 14 (E_Binary_Operation 15 Less_Than_Or_Equal (E_Identifier 16 (** N *) 1) (E_Literal 17 (Integer_Literal 1))) (
          S_Assignment 18 ((** Result *) 2) (E_Literal 19 (Boolean_Literal false))
         ) ) (
        S_Sequence 20 (
         S_Assignment 21 ((** I *) 3) (E_Literal 22 (Integer_Literal 0)) ) (
         S_While_Loop 23 (E_Binary_Operation 24 Less_Than_Or_Equal
             (E_Binary_Operation 25 Multiply (E_Identifier 26 (** I *) 3) (E_Identifier 27 (** I *) 3)) (E_Identifier 28 (** N *) 1)) (
           S_Sequence 29 (
            S_Assignment 30 ((** X *) 4) (E_Binary_Operation 31 Divide (E_Identifier 32 (** N *) 1) (E_Identifier 33 (** I *) 3)) ) (
            S_Sequence 34 (
             S_If 35 (E_Binary_Operation 36 Equal
                 (E_Identifier 37 (** N *) 1) (E_Binary_Operation 38 Multiply (E_Identifier 39 (** X *) 4) (E_Identifier 40 (** I *) 3)))
               S_Assignment 41 ((** Result *) 2) (E_Literal 42 (Boolean_Literal false))
              ) ) (
             S_Assignment 43 ((** I *) 3) (E_Binary_Operation 44 Plus (E_Identifier 45 (** I *) 3) (E_Literal 46 (Integer_Literal 1)))
          ) ) ) )
     )).
```

SPARK AST in Coq

1) Translation from SPARK to AST in Coq
2) Check program's well-formedness with certified static analyses (well-typed, well-defined, well-checked)
3) Run certified interpreter, which captures the *division by zero* exception

Future Work

- Extend the language subset
  - Add function call
  - Add array, records, subtypes
  - … and so on
- Add run time checks optimizations and prove its correctness
- Certified CompCert frontend for SPARK

END !

Thanks