Formalization of SPARK Subset in Coq

Zhi Zhang

Conservatoire National des Arts et Metiers	*Kansas State University*	*AdaCore*	*Altran*
Pierre Courtieu	Zhi Zhang	Jereom Guitton	Trevor Jennings
Maria Virginia Aponte	Robby		
Tristan Crolard	Jason Belt		
	John Hatcliff		

***Funding** : this work is supported by ….*

Outline

- Motivation
- Formalization Work
- Demo
- Future Work

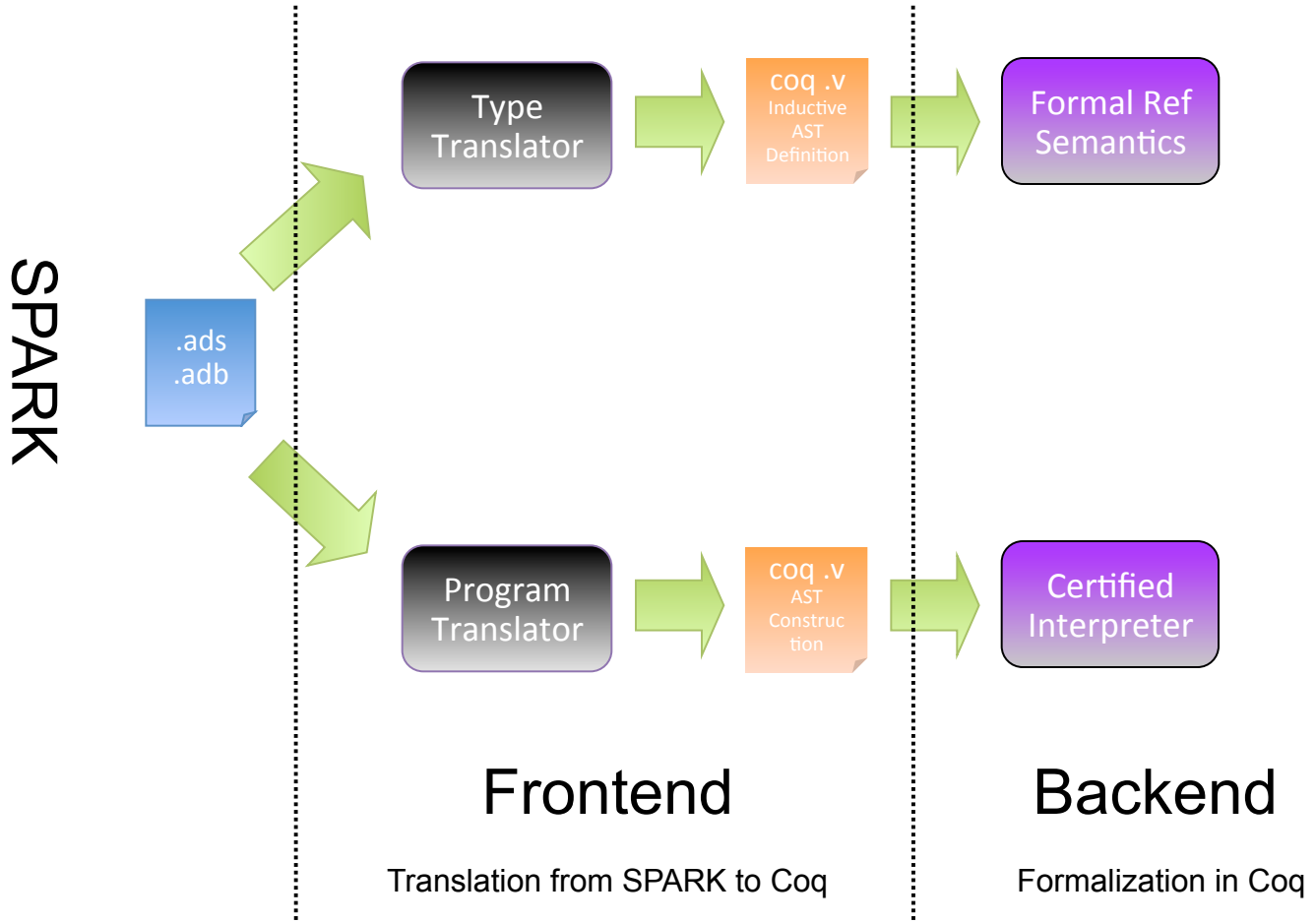
Our Work

- 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

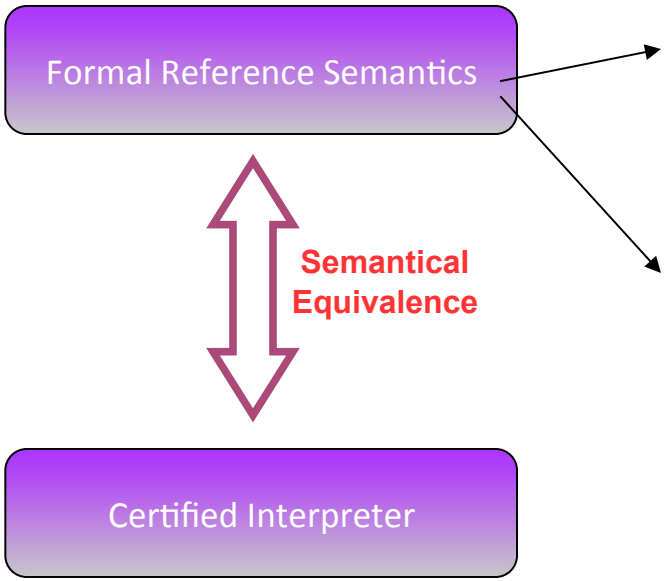
Motivation

- 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 behaves as prescribed by the reference semantics

✓ **Verify SPARK Program Correctness**

It's proved that for any well-formed terminating SPARK program, it will terminates 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;
```

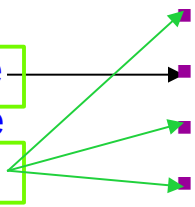
SPARK Code

```
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
- Erros 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

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

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

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)

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*;

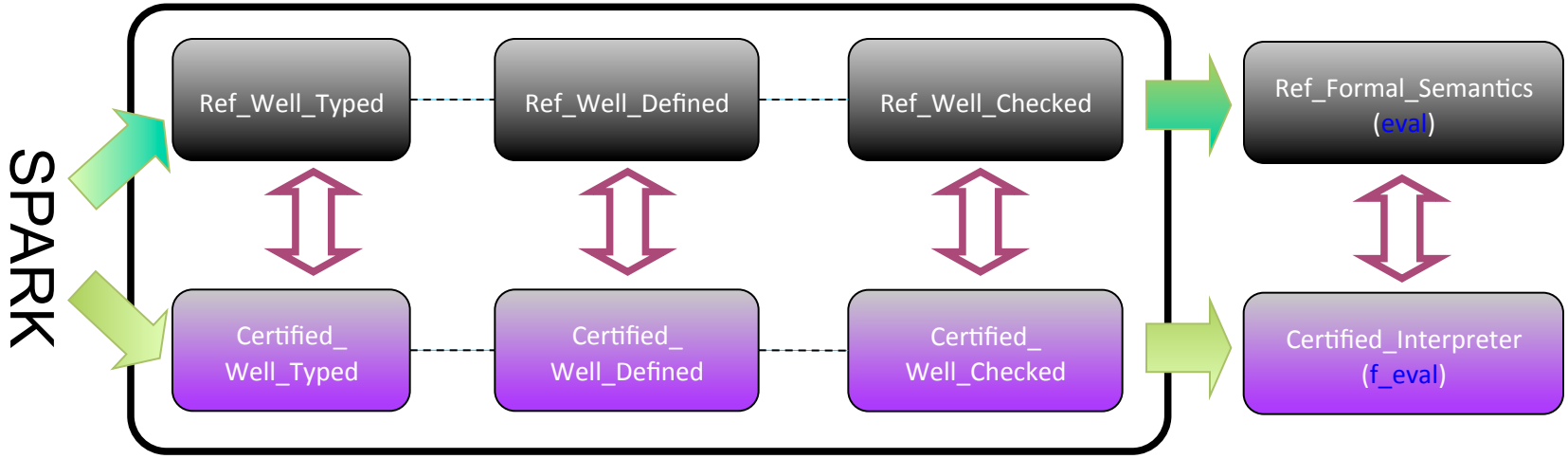
Static Analysis

- 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

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 !