18 NOVEMBER 20205

Formally Verifying YUL-Level Static Analyses and Optimization with Rocq

The FORYU Project (Funded by the Ethereum Foundation)
http://github.com/costa-group/foryu

Samir Genaim

The Costa Group
Complutense University of Madrid

Project members: Elvira Albert and Enrique Martin-Martin

CONTEXT

Formal Guarantees for Compiler Correctness

Modern compilers heavily rely on static analysis and optimization components to efficiently transform high-level source code into highly optimized, high-performance machine code.

Static analysis and optimization tools can have implementation bugs.

The correctness of the entire compilation process depends on these components being correct.

Provide formal, mathematical guarantees that static analysis results are correct.

The (desired) solution is to use formal verification of these components.

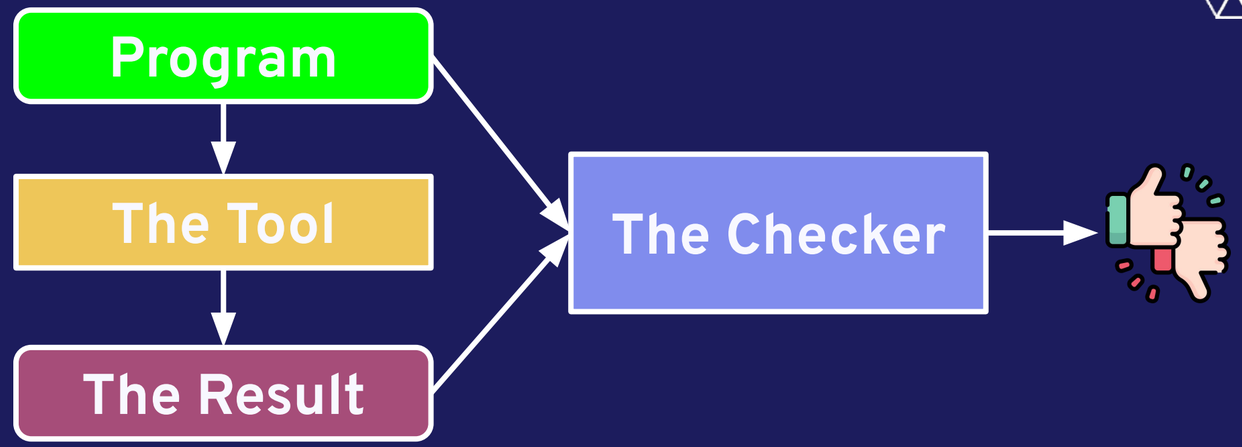
Formal Verification of Formal Methods Tools

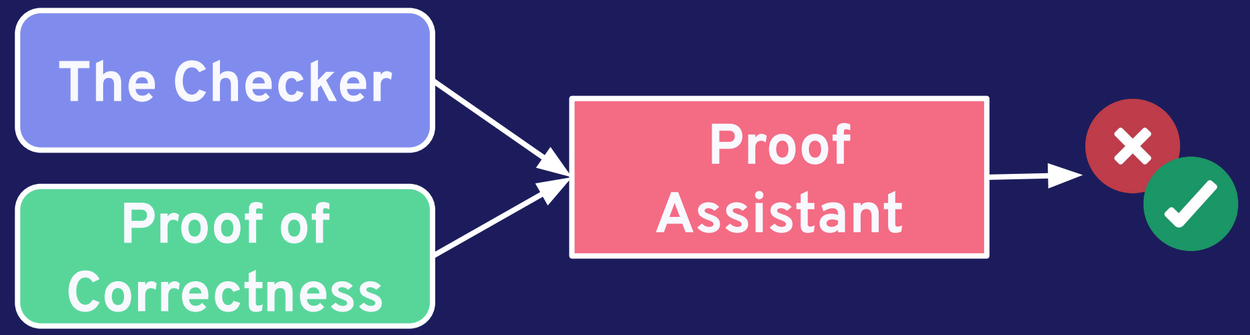
A common approach for verifying correctness of programs in general is the use of proof assistants (Coq, Lean, Isabelle, etc), which are interactive software environments used to construct, check, and certify formal, mathematical proofs of program properties with high rigor.

- The tool must be written in the language supported by the proof assistant (or a formal model must be created).
- The proof is mathematical using the logic supported by the tool.
- **The Verification Challenge: Writing a complete proof is complex, as the tools are often intricate and utilize multiple programming languages.**

Formal Verification via Certified Checkers

Instead of verifying a complex, evolving tool, we can write a simple Checker that verifies the tool's output after every run. We then formally verify the correctness of this Checker. This shifts the high verification burden from the large tool to a smaller, simpler component.





- The Checker is typically written in the language supported by the proof assistant.
- The Checker is formally certified, which provides a high-confidence guarantee that if the results passes the check, they are correct.

The Need for a Formal Semantics

Writing formal statements about properties of programs (those inferred by the tool) requires modeling the program's semantics within the proof assistant, a fundamental step regardless of whether the verification targets the tool or a corresponding checker.

- The semantics and corresponding data types for programs and states are defined in the proof assistant language.
- Semantics are typically modeled using an Operational Semantics, which describes execution as a sequence of discrete steps.
- Operational Semantics can be implemented as a simple function for a single step, or as a transition relation where the transitive closure defines possible executions.
- The specific semantic model chosen depends entirely on the program property targeted for verification.

The FORYU Project

This project, funded by the Ethereum Foundation, aims to develop a formal semantics for the YUL intermediate language using the Rocq proof assistant. The long-term objective is to leverage this semantics for verifying the correctness of the static analysis and optimization components within the Solidity compiler.

- Develop a formal operational semantics for YUL.
- **Key Feature**: The semantics is Parametric in the Dialect.
- Implemented the EVM dialect (executable where possible) as a concrete instance.
- Currently focused on the SSA-CFG level (non-SSA supported). Future work includes semantics at the AST level.
- **Proof-of-Concept**: Used to verify the results of liveness analysis.
- http://github.com/costa-group/foryu

The Dialect Interface

```
Module Type DIALECT.
 Parameter value_t : Type.
 Parameter opcode_t : Type.
 Parameter dstate_t : Type.
 ...
 Parameter execute_op_code :
  dstate_t → opcode_t → list (value_t) →
  (list value_t * dialect_state * Status.t).
...
 End DIALECT.
```

The type of basic values (e.g. 256 bits)

A type defining the supported opcodes

A type for the dialect state (e.g., store and memory).

A function to execute one opcode, returning output values, new dialect state, and status.

An implementation of this interface for the EVM dialect is available as EVM_Dialect.

A Smart Contract

```
Module SmartContract (D: DIALECT).
 Record t : Type := {
  name : string;
  functions : list Function(D).t;
  main: FunctionName.t;
 }.
 Definition get_function ...
 Definition get_block ...
 Definition get_instr ...
 ...
End SmartContract.
```

The core SmartContract type, which is parametric in the DIALECT D.

Each function is a Control Flow Graph (CFG), where each block holds a sequence of instructions of the form:

- [v1,v2] := [0x32,v0] **ASSIGNMENT**
- [v3] := ADD [v1,v2] **DIALECT CALL**
- [v4,v5] := “f” [v3,0x44] **FUNCTION CALL**

Blocks have information on their kind: Normal, Branching, Return, etc.

Blocks have PHI functions to support SSA.

YUL State

```
Module State(D: DIALECT).

 Record t : Type := {
   call_stack : CallStack(D).t;
   status : Status.t;
   dialect_state : D.dstate_t;
 }.
 ...
End State.
```

The core YUL State Type, which is parametric in the DIALECT D.

A call stack, where each frame has information on which function/block/instruction is executing, the values of the variables, etc.

The execution status can be: Running, Reverted, Terminated, or Error.

The dialect-specific state (managed by the DIALECT module).

YUL Semantics

```
Module SmallStep (D: DIALECT).
  ...
  Definition step (s: State(D).t)
                  (p: SmartContract(D).t) :
                  State(D).t := ...

  Fixpoint eval (n: nat)
                (s: State(D).t)
                (p: SmartContract(D).t) :
                State(D).t := ...
End SmallStep.
```

The semantics is parametric in the DIALECT.

A function to execute one semantic step starting from state s.

A function to execute n steps starting from state s, effectively defined by repeatedly applying step.

PROOF OF CONCEPT

Liveness Analysis

A set of variables X is live at the beginning of block b if any variable not in X is never accessed, before it is modified, in any execution starting at the entry of b.

- We implemented a certified checker for the liveness analysis results.
- We formally verified the correctness of this checker by proving the following theorem:

```
Theorem snd:
forall p r,
  live_checker p res = true →
  forall fname bid st1 st1' n,
    get_pc st1 = (fname,bid,0) →
    eval n p st1 = st1'→
    forall st2 v,
      ~VarSet.In v (r fname bid) →
      equiv_up_to_v st1 st2 v →
        exists st2',
          eval n p st2 = st2' /\
            equiv_up_to_v st1' st2' v
```

CONCLUSIONS

Conclusion & Future Work

The FORYU project has successfully established a viable path for verifying compiler components by focusing on certified checkers. The verification of live variable analysis serves as a strong proof-of-concept, validating our approach and paving the way for more robust, formally-backed compiler development.

- **Verification Success**: Formal verification of the liveness analysis checker confirms the feasibility of our methodology.
- **Application**: Plan to apply the certified checker to a broader range of smart contracts.
- **Extend Scope**: Plan to formally verify additional static analysis components (e.g., constant propagation).
- **Usability**: Plan to build user-friendly interfaces (e.g., command-line interface) for easy integration.
- **AST Semantics**: Define formal semantics at the AST level to support verification of early-stage compiler optimizations.

18 NOVEMBER 20205

Q&A