Project Proposal: A Formally Verified Static Analyzer for Detecting Runtime Errors in a Basic Imperative Language

Fahim A. Islam

February 14th, 2026

1 Introduction

1.1 Problem Domain & Motivation

Modern software is increasingly generated or modified automatically, including by AI-assisted tools. Ensuring that such software does not fail at runtime is critical for reliability and safety. Static analysis provides a method to reason about program behavior without actual execution of the program[1].

However, static analyzers themselves are complex software systems and may contain mistakes. This has motivated research into building verified analyzers whose guarantees are themselves proven correct, such as the Verasco project [2]. If the analyzer is incorrect, its guarantees are meaningless.

In this project, I aim to implement a small static analyzer for a basic imperative programming language and formally verify, using Dafny, that its results are sound [3] with respect to the operational semantics of the language.

This project draws inspiration from research in:

- program verification
- abstract interpretation
- compiler correctness
- safe reasoning about automatically generated code

1.2 Program

The tool will:

- accept programs written in a small imperative language
- build an abstract AST (Abstract Syntax Tree)
- statically check for potential runtime errors

The analyzer will detect the following runtime errors:

- division by zero
- reading uninitialized variables
- invalid arithmetic expressions

Input: A program represented as an AST.
Output: *SAFE* or *UNSAFE* (possibly with error information).
If a program is marked SAFE: it will execute without runtime errors.

1.3 Scope

I intentionally restrict the language to keep verification tractable.

Features will include:

- integer variables
- assignments
- arithmetic expressions
- conditionals
- bounded loops (dependent on time)

1.4 Specifications of Interest

My goal is to build a static analyzer for a small imperative language and verify the analyzer's correctness in Dafny. The most important property in this domain is **soundness**: when the analyzer claims a program is safe, the program should not encounter the targeted runtime errors during execution. The objective is not only to detect errors, but to also prove that the analyzer's judgments are correct.

Core functional correctness properties

- **Well-formedness of programs**: The analyzer assumes the input AST is syntactically well-formed (e.g., variables are named consistently; correct number of operands and operators). This will be encoded as a predicate called $WellFormed(statement)$ and used as a precondition for analysis. (i.e. $requires\ WellFormed(statement)$)
- **Soundness of error checking**: If the analyzer returns SAFE, then executing the program from any valid initial environment will not produce a runtime error from our error set (e.g., division-by-zero, use of uninitialized variables).

  Informally: $(Analyze(p) == SAFE) ==> Exec(p)$ will run without runtime errors.
- **Correctness of core semantics helpers**: We will define an interpreter/evaluator for expressions/statements that will attempt to replicate runtime behavior. We will verify basic functional correctness for these components (e.g., assignment updates the environment as intended).

Runtime errors of interest:

- Division by zero: evaluate an expression $exp1/exp2$ when $exp2 = 0$.
- Use of uninitialized variables: evaluating an expression containing a variable that has never been assigned a value.

2 Implementation

2.1 Tool and Language Selection

The project will be implemented primarily in **Dafny**. Dafny is well-suited for this project because it supports inductive datatypes for defining a small language AST, and it provides automated verification via preconditions, postconditions, and ghost code/lemmas. The project will not rely on external libraries, networking, filesystem interaction, or GUI features.

Lightweight scripts (e.g., a small Python driver) may be used only for running examples or certain outputs however, the verified core (language semantics and analyzer) will remain entirely in Dafny.

2.2 Architecture of the Code

The tool will operate on programs represented as an abstract syntax tree (AST). At a high level, the project will include the following components:

- **AST Definitions:** Datatypes for expressions and statements (e.g., constants, variables, arithmetic operations; assignment, sequencing, conditionals, and an optional loop construct).
- **Interpreter:** A formal operational semantics describing how programs execute over an environment by mapping variables to integer values. Execution may produce a normal resulting environment or a runtime error (e.g., division by zero, uninitialized variable read).
- **Static Analyzer:** A function that traverses the AST and computes whether a program is *SAFE* or *UNSAFE*. The analyzer will track abstract information such as which variables are definitely initialized and whether expressions are guaranteed to be safe to evaluate.
- **Proof Layer:** Lemmas and predicates connect analyzer results to the interpreter semantics, culminating in a soundness theorem.

A tentative dataflow for the tool is:

$Program\ AST \rightarrow AnalyzeProgram \rightarrow (SAFE\ or\ UNSAFE)$

and soundness will be relative to the interpreter: $SAFE ==> Execute(program)$ without runtime error.

2.3 Verification Effort

The primary verification goal is a **soundness guarantee**: if the analyzer returns *SAFE*, then executing the program according to the formal semantics cannot produce the targeted runtime errors.

To support this, the following components/properties will be verified at **compile-time** using Dafny:

- **Expression safety:** If the analyzer claims an expression is safe under a given abstract state (e.g., set of initialized variables), then evaluating the expression will not trigger a runtime error.
- **Statement safety:** If the analyzer claims a statement is safe, then executing the statement will not trigger a runtime error.
- **Main soundness theorem:** Programs marked *SAFE* will not produce runtime errors during execution.

Out of scope: The analyzer will attempt to be sound but not necessarily complete (i.e., it may mark programs that are truly *SAFE* as *UNSAFE*). The project will also not attempt to verify performance, precision/optimality of the analysis, or user-facing error messages.

2.4 Challenges & Risk Mitigation

- Writing specifications that are strong enough to prove soundness while remaining tractable for Dafny
- Managing proof complexity as language features increase (especially loops).

To mitigate risk, the project will start with simple programs and conditionals. Support for loops or more advanced reasoning (e.g., richer numeric invariants) will be done as an optional extension if time permits. If verification becomes difficult, the language feature set will be reduced while preserving the main soundness theorem for the remaining core language.

3 Development & Timeline

As I will be working individually, there is no need for coordination, division of labor, or recurring group meetings.

Here is the projected timeline of the work that I intend to complete:

- **Step 1**: Proposal completed and submitted.
- **Step 2**: Define the abstract syntax of the language, environments, and the semantics for expressions and statements. Ensure the interpreter compiles and basic execution examples work.
- **Step 3**: Implement the core static analysis passes, including tracking initialized variables and identifying potentially unsafe operations such as division by zero or using uninitialized variables.
- **Step 4**: Begin formal verification of the analyzer. Prove correctness properties for helper functions and establish any necessary lemmas relating analyzer results to the semantics.
- **Step 5**: Complete the main soundness theorem stating that programs marked *SAFE* will not produce runtime errors. Use remaining time for debugging proofs, simplifying specifications, improving automation, and preparing material for the final presentation and report.

4 Conclusion

My intention is to have a formally verified guarantee that our analyzer correctly predicts absence of runtime errors. This project should demonstrate how formal methods can increase trust in automated reasoning systems, which is especially relevant as AI systems increasingly generate code.

References

[1] P. Cousot and R. Cousot. *Static determination of dynamic properties of programs.* Proceedings of the Second International Symposium on Programming (ISOP), 1976.

[2] J. H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. *Verasco: A formally verified C static analyzer.* POPL, 2015.

[3] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. *Comparing techniques for certified static analysis.* 2010.

Acknowledgments

I used an AI-based writing assistant (ChatGPT 5.2) to help refine wording, organization, and clarity of the proposal.