Verus: A Lambda Calculus Alternative to MISRA-C and CompCert

https://github.com/adam-mcdaniel/verus

1st Adam McDaniel
EECS Dept.
University of Tennessee
Knoxille, USA
amcdan23@vols.utk.edu

2nd Alexander Krneta
EECS Dept.
University of Tennessee
Knoxille, USA
akrneta@vols.utk.edu

3rd Matthew Jones
EECS Dept.
University of Tennessee
Knoxille, USA
mjone205@vols.utk.edu

4th Kapildev Neupane
EECS Dept.
University of Tennessee
Knoxille, USA
kneupan1@vols.utk.edu

***Abstract*—Formally verified programming languages, such as MISRA-C and CompCert, play a crucial role in safety-critical systems by preventing classes of bugs that could result in catastrophic failures. However, these languages are based on C, which introduces unnecessary complexity and limitations due to its imperative and low-level nature. We propose a new formally verified language based on the simply-typed lambda calculus, offering stronger static guarantees, improved expressiveness, and reduced developer burden. This project aims to investigate whether such a language can replace formally verified C while maintaining the same safety assurances and improving productivity.**

***Index Terms*—compilers, functional programming, formal verification**

I. Introduction

Formal verification techniques are essential for ensuring the safety of software used in medical, automotive, and aerospace industries. Tools like MISRA-C and CompCert impose strict constraints on C to ensure properties like memory safety and termination. However, these constraints make development cumbersome and limit the efficiency of formally verified C programs.

A. Research Need

Formal verification strategies are *absolutely necessary* for safety-critical systems; software failures in C have led to catastrophic consequences. The Therac-25 radiation therapy machine malfunctioned due to integer overflows and race conditions in C, leading to lethal overdoses. The Boeing 737 Max control system, also written in C, suffered from safety-critical defects that contributed to two crashes.

C is the dominant language for safety-critical systems, but its imperative nature presents significant verification challenges. The flexibility of pointer arithmetic, manual memory management, and weak type safety force verification tools like MISRA-C and CompCert to adopt overly conservative constraints, requiring extensive manual proof effort from developers.

B. Research Question

How can we design a formally verified language on a more expressive foundation than C, while maintaining the same safety guarantees and improving developer productivity? What trade-offs arise when replacing formally verified C with a formally verified functional language?

II. Related Work

Several functional and dependently typed languages have been explored for formal verification, including:

- **Agda**: A dependently typed functional programming language that ensures correctness via proofs.
- **F***: A functional language with dependent types, designed for verified programming.
- **Dafny**: A verification-aware programming language that enforces correctness through specifications.
- **Linear Haskell**: A linear type system extension to Haskell that ensures safe resource usage.

While these languages focus on verification, they are not intended as direct replacements for formally verified C in safety-critical applications. Our work seeks to bridge this gap.

III. Method

A. Language Design

We propose a formally verified language based on the Simply Typed Lambda Calculus (STLC), incorporating:

- **Sum and Product Types**: For expressiveness without requiring unsafe unions.
- **Total Functions**: Guaranteed termination, eliminating the need for manual proofs.
- **C-like syntax**: Providing a better programming interface for those transitioning from MISRA-C and CompCert compared to other formally verified languages like Agda.
- **C Foreign Function Interface**: For easily interacting with existing IO tools and writing *useful* programs.

B. Implementation Approach

We will implement a prototype compiler that:

1) Parses a C-like syntax using **nom** in Rust.
2) Type-checks according to the simply-typed lambda calculus typechecking rules, which are straightforward and easy to implement. These will enforce our safety guarantees.
3) Compiles to the high-level C-like language Clover. Clover provides a convenient interface for compiling to LLVM with a simple C FFI system. This will significantly reduce the complexity of the frontend compiler.

C. Validation

Finally, we will compare the new language approach to existing tools:

1) **Expressiveness**: We'll measure the expressiveness against standards like MISRA-C and CompCert by comparing equivalent example programs in each language.
2) **Performance**: Evaluate the efficiency of compiled code.

IV. Schedule and Project Management

Writing a compiler is a daunting task. To simplify the pipeline of compilation, we'll leverage Clover to make our backend significantly less complex. This will allow our frontend to be significantly smaller, and make the complexity of the project managable for a single semester. Implementing simply-typed lambda calculus is much easier than typesystems like System F, which can be done in < 1000 lines depending on the implementation language. Below is the planned development schedule accounting for the difficulty of each component.

TABLE I
Project Schedule

Week	Task
1	Implement a recursive descent parser with **nom** for the AST.
2-3	Implement a minimal type checker for the STLC type system.
4-6	Develop code generation for the Clover backend.
7	Write example code and compare to existing tools.

V. Project Team

Our project team consists of:

- **Adam McDaniel**
- **Alexander Krneta**

Here's a list of related projects our team has worked on before:

A. Compilers

- Sage Programming Language
- Oak Programming Language
- Harbor Programming Language
- Tsar Programming Language
- BASM Assembler
- Free Programming Language
- Bootstrapped Wisp Compiler
- Extended BrainF$ Compiler

B. Interpreters

- Sage-Lisp
- Wisp
- Dynamic BrainF$ Web Interpreter
- Dune Scripting Language
- Maroon Functional Programming Language

C. Formal Methods

- Reckon - A Prolog-like Theorem Prover

VI. Expected Results

We expect to demonstrate that a functional language with strong type guarantees can provide the same level of formal verification as MISRA-C and CompCert while reducing developer burden. The outcome will include:

- A formally verified language prototype.
- Benchmarks comparing verification effort and performance with existing tools.
- Insights into trade-offs between imperative and functional verification approaches.

VII. Conclusion

Our proposed research seeks to modernize formally verified programming for safety-critical systems by replacing C with a lambda calculus-based approach. By leveraging advanced type systems, we aim to make verification more expressive, efficient, and developer-friendly.