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 introduce 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 show that such a language can outperform formally verified C while maintaining the same safety assurances and improving productivity.**

I. Introduction

In recent years, due to many life-threatening software failures, such as the infamous Therac-25 [1] radiation machine and the Boeing 737-MAX [2], formally verified software has become a government-imposed mandate for devices in medicine, transportation, and aerospace. The vast majority of software written in these important industries uses formally verified languages based on C dialects, such as MISRA-C or CompCert [3].

This C-based approach has an Achilles heel: the underlying language C was designed for performance and flexibility, *not* for static reasoning. Pointer arithmetic, undefined behavior, and manual memory management combine to make proofs laborious and brittle. As a result, developer productivity is hindered — developers are required to work harder to make their code pass the intense verification checks. Our hypothesis is simple: replace the C substrate rather than contort it. By grounding a language in the Simply-Typed-Lambda-Calculus (STLC) — where functions are total and side-effects explicit — many desirable guarantees emerge for free, enabling simpler tooling and lighter cognitive load for developers.

II. Comparison With Existing Work

There already exists a large ecosystem of verification-oriented languages. On the functional-programming end of the spectrum, there's Agda, Coq, Idris and F, which wield dependent types to encode proofs alongside programs. On the more imperative-programming end of the spectrum, Rust introduces ownership to tame aliasing, and SPARK Ada strengthens Ada with contracts. However, none of these have become a suitable industry replacement for C because they [4]:

- expose steep proof-engineering overhead which introduces a significant cognitive load (Agda, Coq),
- rely on heavyweight runtime infrastructure with fewer *static* guarantees (Idris, Haskell), or
- remain imperative at their core (SPARK Ada, Rust).
- cannot provide certain guarantees, such as termination proofs (Rust).

Verus stakes out a middle ground: minimal calculus, eager termination, a more familiar C-like syntax compared to other functional languages, and the ability to extend the interpreter with well-checked built-in functions.

III. Language Overview

A. Core Calculus

Verus's semantics are the Simply Typed Lambda Calculus augmented with sum, product, and list types, along with some other basic primitives:

$$\tau = \mathsf{Void} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Num} \mid [\tau_1] \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2$$

Below are the typing rules of STLC. These rules alone are needed to enforce the type-soundness of all expressions and guarantee termination of programs.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (1)$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T} \quad (2)$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma . e) : (\sigma \to \tau)} \quad (3)$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau} \quad (4)$$

Every program must be typable in this grammar; hence recursion must be *structurally decreasing*, guaranteeing termination just like MISRA-C. This also means that Verus is not turing-complete, which makes *many* more static proofs about memory, control flow, etc. tractable.

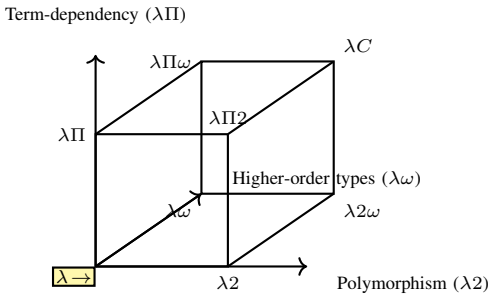

Fig. 1. Depiction of Barendregt's Lambda Cube, with Simply Typed Lambda Calculus's coordinates highlighted in yellow.

This type system lies squarely at the origin of Barendregt's Lambda Cube, while most other functional-oriented verification languages lie in the upper face of the cube [15].

B. Surface Syntax

Adopting braces, semicolons and `let` bindings keeps the learning curve low for C engineers while preserving functional purity.

```
{
    // Create a union/sum type that can
    represent a number, or void.
    type MaybeNum = {Some(Num) | None};

    // Create a division function that
    // returns None for an undefined result,
    // or Some number otherwise.
    let div n: Num, d: Num =
        // If the denominator is zero,
        // return None.
        if d == 0 then MaybeNum of None
        // Otherwise, return Some result.
        else MaybeNum of Some (n / d);

    // A function that tests the division
    // result.
    let test_div n: Num, d: Num =
        // Use pattern matching
        // on the division result:
        match div n d with
            // If the result is some number,
            // print the equation result.
            case of Some(result) =>
                print (n as Str)
                    & " / "
                    & (d as Str)
                    & " = "
                    & (result as Str),
            // If there's no number, print that
            // the division is undefined.
            case of None =>
                print (n as Str)
                    & " / "
                    & (d as Str)
                    & " is undefined";

    // Test the divion of 10/2
    test_div 10 2;
    // Test the division of 10/0
    test_div 10 0;
}
```

Listing 1. Safe integer division with algebraic data types.

```
{
    // Create a lambda function and bind
    // it to the name `factorial`
    let factorial =
        \n: Num -> product (range 1 n + 1);

    // Get the first command-line argument,
    // after the executable path,
    // and check whether it exists.
    match (args ())@1 with
        // If it exists, calculate the
        // factorial of the passed number
        case of Some(x) => {
            let n = x as Num;
            // Print the result.
            print "Factorial of "
                  & (n as Str)
                  & " is "
                  & ((factorial n) as Str);
        },
        // If it doesn't exist,
        // print an error message.
        case of None =>
            print "Please provide a number.";
}
```

Listing 2. Total factorial using range/product combinators

```
{
    // Declare a 2D point structure.
    type Point = {
        // The X component of the point.
        x: Num,
        // The Y component of the point.
        y: Num
    };

    // A function that shifts a point
    // by a delta-x and delta-y component.
```

```
    let move p:Point, dx:Num, dy:Num =
        {x: p@x + dx, y: p@y + dy};

    // Force the variable `p` to be declared
    // as a Point type by the type-checker.
    let p: Point = {x: 5, y: 6};

    // Shift `p` by 1 and 2 in the X and Y
    // components respectively.
    //
    // Use type inference to declare `p2`
    // as a point.
    let p2 = move p 1 2;

    // Print the shifted point.
    print p2;
}
```

Listing 3. Point structure and manipulation with functions.

C. Safety Guarantees

STLC provides many critical properties *by construction* — these are guarantees resulting from the structure of the type-system alone:

- **Progress** (no stuck terms): A well-typed expression is either a value or can make progress towards the final resulting expression. In other words, every well-typed program has a possible evaluation step remaining or the program is a constant value.
- **Preservation**: Evaluation preserves types. This helps prevent undefined behavior and induction to be applied heavily to infer most types in programs.
- **Strong normalization**: All reduction sequences terminate.

Because of the strong normalization property of STLC, as proven by Tait in 1967, memory safety checks and the proven absence of undefined behavior are possible without the near 200 aditional MISRA-C safety rules [5].

IV. Evaluation Strategy

A. Front-End

A recursive-descent parser built with `nom` produces an abstract syntax tree (AST). The type-checker is a direct transcription of the STLC rules. If the syntax of the input program is invalid, the program is rejected and is not pipelined through the other evaluation steps.

B. Type-Checker

If the parser returns a valid AST, then it is passed through the type-checker to verify the input program. The type-checker infers types which are not listed, and confirms that no mismatched types exist in the program.

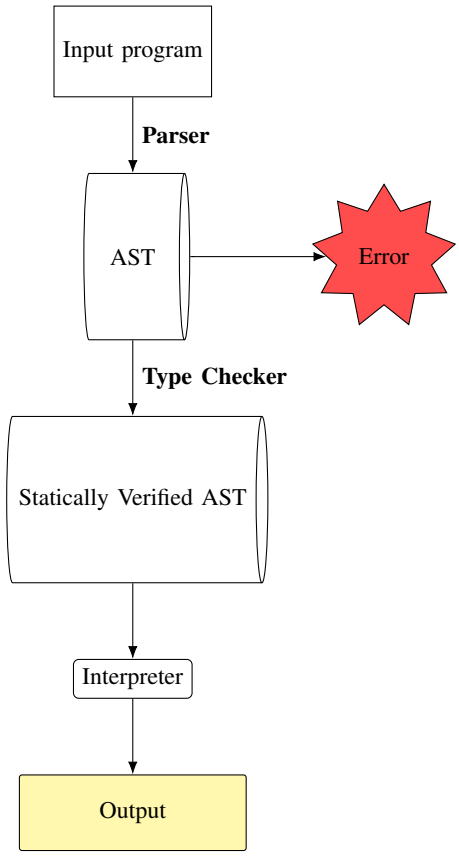

Fig. 2. Flow diagram of Verus's verification and execution.

This includes confirming the validity of function applications, variable assignments, and member accesses of structures, and the indexing of list elements.

C. Back-End

Instead of emitting assembly, Verus uses a minimal interpreter to evaluate the code. A compiler could be built for Verus, but that is beyond the scope of this work. The interpreter receives an expression and an evaluation environment in which to interpret the expression. The initial environment is populated with standard library built-in functions. Lambda functions close over the environment they're evaluated in — they collect the variables from their parent scope and add them to their own scope when they're called later on.

D. Runtime Model

- **Functions**: Functions are stored as closures which capture symbols used in their function body from the parent environment.
- **Heap**: While heap usage is not restricted, calculating the maximum heap usage during runtime is

decidable given that STLC is strongly normalizing. So, memory usage can be restricted in practice similar to MISRA-C.

- **Interpreter**: Verus is implemented with an interpreter in place of a compiler. This was done for simplicity and the sake of time — a compiler would be more desirable.

V. Rationale: Why STLC Over C?

a) Simpler proofs: Soundness of STLC fits on a whiteboard [6]; CompCert's proof spans ∼120,000 lines of Coq [7], for example. Verus transfers proof obligations from each program to the *language design* – no control-flow annotations are required.

b) Fewer developer constraints: MISRA-C has many restrictions on user-code that are lifted in a STLC-based approach. For example, sum-types (`union` in C) are disallowed in MISRA-C [8], while our STLC-based approach allows for statically checked sum-types with pattern matching.

c) Expressive abstractions: Algebraic data types, pattern matching, and higher order functions increase developer productivity by allowing more flexibly written code [9]. These features are built into the type system in Verus, while C can only provide similar code structure with macros.

d) Deterministic execution: Because all functions terminate and the side effects are explicit, the analysis of the worst-case execution time is tractable. This is highly desirable for mission-critical applications.

VI. Trade-Offs and Open Challenges

i. **Mutable state.** Pure lambda-calculus forbids mutation [10]; embedded software often demands it [11]. However, compiler optimizations, such as those implemented for languages like Haskell, can be effective in making functional-style code just as performant as imperative code [12].

ii. **Low-level bit-tricks.** Cryptographic software in C relies on "bit-twiddling" [13]. A `Bits` primitive may be introduced with verified semantics, but this was not implemented in this work.

iii. **Ecosystem inertia.** Tooling, linters and IDE integrations must reach parity with current C workflows to gain traction. This is unlikely given the inertia behind C.

iv. **The Verus Interpreter.** Most formally verified software is designed for use on embedded devices. The Verus interpreter is not desirable for these use-cases — a compiler must be implemented to effectively address this. This could be done by modifying existing functional language compilers.

VII. Possible Future Work

1. **A Verus compiler.** Transitioning to compilation instead of interpretation would certainly make Verus a more attractive option for typical industry applications. It would also significantly boost runtime performance.
2. **A larger standard library.** More built-in functions and types in the standard library could drastically improve developer productivity by reducing the amount of user-code required to achieve common operations and abstractions.
3. **The addition of memory usage analysis.** Although statically proving the required memory for programs is desirable and tractable in Verus's language model, it is not currently implemented in this work. Future work could expand on the current infrastructure to include this in the type-checker.

VIII. Conclusion

This work demonstrates that a Simply Typed Lambda Calculus provides a compelling foundation for formally verified programming languages in safety-critical domains. By implementing a prototype STLC-based language with a more C-like syntax, we have shown a fundamentally different approach to formal verification than MISRA-C and CompCert. While traditional approaches take a Turing-complete language (C) and impose restrictions to make it safe [14], our approach begins with an inherently safe, non-Turing-complete foundation and carefully adds features to increase expressiveness without compromising safety guarantees.

STLC programs are guaranteed to terminate by construction, eliminating the need for control flow annotations to explicitly prove termination. Potentially accelerating development cycles while maintaining reliability standards for industries where formal verification is mandated. Our implementation demonstrates that STLC-based languages allow for many language features that are impossible in C-based approaches, such as algebraic data types, pattern matching, and higher-order functions, all while maintaining stronger static guarantees.

Verus demonstrates the potential for improving the developer experience in formally verified systems without sacrificing safety. While this prototype serves as a proof-of-concept, future STLC-based would need polished compilation capabilities, memory analysis tools, and expanded libraries to be viable alternatives in industry settings. Our results suggest that lambda calculus offers a promising path forward for critical systems development–one that balances rigor with practicality and reduces the burden currently imposed by C-based verification frameworks.

REFERENCES

[1] A. Fabio, "Killed by a Machine: The Therac-25," *Hackaday*, Oct. 26, 2015. [Online]. Available: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/. [Accessed: May 7, 2025].

[2] G. Travis, "How the Boeing 737 Max Disaster Looks to a Software Developer," *IEEE Spectrum*, Apr. 18, 2019. [Online]. Available: https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer. [Accessed: May 7, 2025].

[3] F. Boutekkouk, E. Bouaghi, and O. Bouaghi, "C Software Formal Verification Review, Challenges and Future Directions," in *Proceedings of the 2024 IEEE International Conference on Software Engineering and Formal Methods (SEFM 2024), Sep. 2024, pp. 1–8.

[4] K. Havelund, "A Half Century of Formal Methods—Challenges and Trends," in *Proc. FM-50 Workshop*, Lecture Notes in Computer Science, vol. 13448, 2022. [Online]. Available: https://www.havelund.com/Publications/fm-50-2022.pdf. [Accessed: May 7, 2025].

[5] W. W. Tait, "Intensional interpretations of functionals of finite type I," *J. Symbolic Logic*, vol. 32, no. 2, pp. 198–212, Aug. 1967.

[6] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[7] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," in *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*, Salt Lake City, UT, USA, Jan. 2006, pp. 42–54.

[8] Motor Industry Software Reliability Association, *MISRA-C:2004 — Guidelines for the use of the C language in critical systems*, Coventry, UK: MISRA, Oct. 2004.

[9] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Apr. 1989. DOI: 10.1093/comjnl/32.2.98.

[10] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, 2nd ed. Amsterdam, The Netherlands: North-Holland, 1984.

[11] A. Vahid and T. Givargis, *Embedded Systems: A Contemporary Design Tool*. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[12] G. L. Burn, S. Peyton Jones, and J. D. Robson, "The Spineless G-Machine," in *Proc. 1988 ACM Symp. Lisp and Functional Programming*, Snowbird, UT, USA, Jul. 1988, pp. 244–258.

[13] H. S. Warren, Jr., *Hacker's Delight*, 1st ed. Boston, MA, USA: Addison-Wesley Professional, 2002.

[14] R. C. Seacord, *Secure Coding in C and C++*, 1st ed. Boston, MA, USA: Addison-Wesley, 2005.

[15] H. P. Barendregt, "Introduction to generalized type systems," *Journal of Functional Programming*, vol. 1, no. 2, pp. 125–154, 1991. DOI: 10.1017/S0956796800020025.