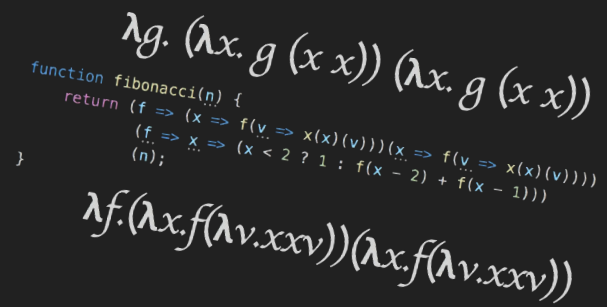
Verus: A Lambda Calculus Alternative to MISRA-C and CompCert

● *Adam McDaniel* ● *Alexander Krneta* ● *Matthew Jones* ● *Kapildev Neupane* ●



Introduction

Motivation To Improve Formally Verified Languages

- Formally verified languages dominate safety-critical and mission-critical applications.

- Medical devices, aerospace, automotive, rail, and defense + military industries all require the use of formally verified languages for most applications.

- If we can decrease the burden on developers writing formally verified code, we could increase the throughput of software development in these very important industries.

Research Question

- Widely used formally verified languages like MISRA-C and CompCert have very restrictive rules enforced by the compiler.
- Many of these restrictions are required because C code is very difficult to statically analyze.
- Languages based on Lambda-Calculus are renowned for their static analysis capabilities.
- **Research question:** Can we remove more developer restrictions in formally verified languages by using a programming model based on Lambda-Calculus?

Comparison of Paradigms

C-Based Approach	STLC-Based Approach
Imperative based paradigm -- weak static guarantees.	Functional based paradigm -- *strong static guarantees*, and *formal semantics built into the mathematical model* of the language.
No sum types -- *union types are disallowed.*	Tagged union types are supported and are *statically checked for soundness.*
Reasoning about state, aliasing, and mutation are extremely difficult.	Reasoning about state, aliasing, and mutation are *notoriously simple!*
The substrate language is turing complete, making proof searches much more difficult and extensive. Not all programs can be analyzed, forcing outright rejection of some complex programs that still might terminate.	Decidable proof search: STLC is not turing complete, making proofs for memory usage and output correctness possible in all programs.

Methodology

Our Approach

- To answer the research question, we implemented an STLC-based language with more developer-friendly features than MISRA-C, but with the same safety constraints.

- For example, it must be possible to statically analyze memory usage and program termination properties, while only *adding* features to the language.

- If we can enforce the same guarantees that MISRA-C provides, but with more features, STLC and other lambda calculus based approaches may be able to improve developer productivity in safety-critical industries.

Our Language

divide.vs

```
type MaybeNum = {Some(Num) | None};

let div n:Num, d:Num =
    if d == 0 then MaybeNum of None
    else MaybeNum of Some (n / d);

let test_div n:Num, d:Num =
    match div n d with
        case of Some(result) => print (n as Str)
                                & " / "
                                & (d as Str)
                                & " = "
                                & (result as Str),
        case of None => print (n as Str)
                                & " / "
                                & (d as Str)
                                & " is undefined";
test_div 10 2;
test_div 10 0;
```

factorial.vs

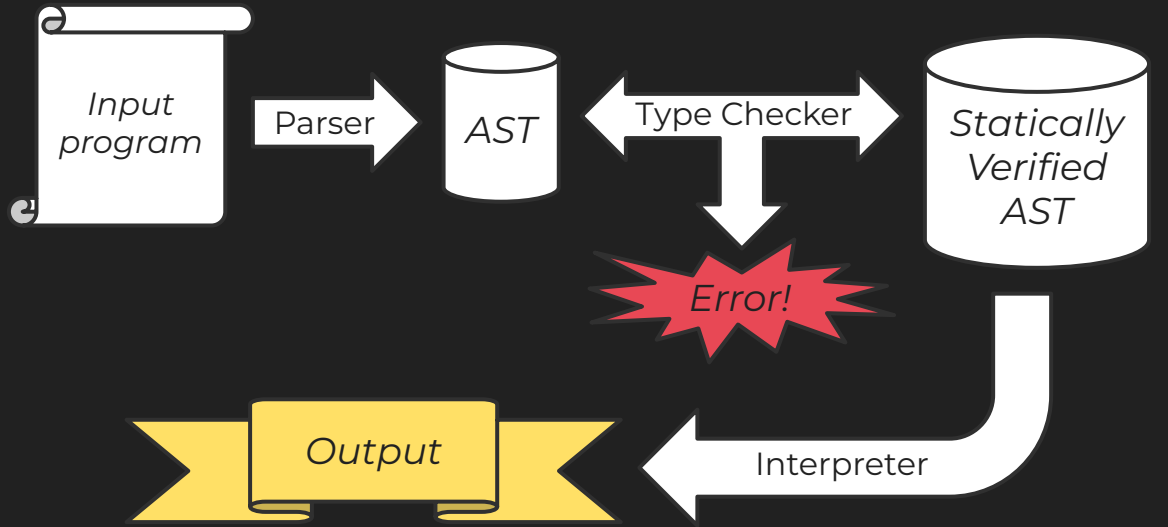
```
let factorial = \n: Num -> product (range 1 n + 1);

let n = (args())@1 as Num;
let result = factorial n;
print "Factorial of " & (n as Str) & " is " & (result as Str);

help print;
help product;
```

point.vs

```
type Point = {
    x: Num,
    y: Num
};

let p: Point = {x: 5, y: 6};

let move p:Point, dx:Num, dy:Num = {x: p@x + dx, y: p@y + dy};
let p2 = move p 1 2;

print p2;
```

Our Implementation

- We created an interpreter with a strict statically checked type system based on STLC.
- It parses the input program into an AST, checks the AST for invalid types usage.
- The guarantees that MISRA-C enforces are ***instead enforced by the strict type-checking rules.***
- These rules *statically guarantee termination and prevent unhandled runtime errors.*




```
(dune) /Users/adam/Documents/rust/verus$ verus --help
```

```
verus
```

```
Usage: verus [INPUT_FILE] [ARGS]...

Arguments:
  [INPUT_FILE]
          The input file to typecheck and evaluate

  [ARGS]...

Options:
  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

```
$ verus examples/factorial.vs 5
Factorial of 5 is 120
 INFO Result: ()
 INFO Program executed successfully
$
```

```
$ verus examples/enum.vs
10 / 2 = 5
10 / 0 is undefined
 INFO Result: ()
 INFO Program executed successfully
$
```

Results

Comparison of Achieved Features

MISRA-C	Our Language
❌ Proof annotations for function and loop termination.	✅ No proof annotations -- all programs are guaranteed to terminate!
❌ No union types allowed.	✅ Supports union types along with pattern match statements!
❌ No support for anonymous functions.	✅ Allows anonymous lambda functions which may be composed with one another -- increasing flexibility.
❌ Relies on linking foreign code to support basic language features.	✅ Has lots of built-in primitives which can be statically checked independently of user code to prevent misuse and logical errors.

Shortcomings Of Our Implementation

- ❌ We do not support pointers, although these are *definitely possible*. The STLC-based approach is compatible with pointers.
  - Memory safety can also be enforced with ownership and borrowing.

- ❌ We did not statically calculate the exact required memory for input programs to run properly, but this is *trivial* given the structure of STLC.
  - Other Lambda-Calculus variants will be much more difficult to reason about, though.

- ❌ We did not have enough time to implement a compiler, so extended functionality must be implemented with built-in functions added to the interpreter, rather than through linking.
  - STLC is a subset of many functional languages, so implementing a compiler for it is not too difficult.

Conclusions

Conclusions

- Lambda-Calculus based approaches allow to statically checked language features that other paradigms do not allow:
  - Iterative-based programs are significantly harder to verify due to loop and function termination proofs.
- Our implementation is proof of this: many language features that are impossible to be checked in C can be introduced with the same guarantees as MISRA-C.
- **STLC and other Lambda-Calculus based approaches are likely a better substrate for formally-verified languages.**
  - Since we can lift many restrictions from iterative-based paradigms with STLC, STLC-based languages should result in better developer productivity in safety-critical industries.

Questions?