Machine Verified Reasoning

Damiano Testa

University of Warwick

February 12, 2026

Introduction

Can we be sure that

- a computer program has no bugs, or
- a mathematical proof is truly sound?

These questions live in software verification and proof checking.

Motivation

Part of today's abstract is AI-generated: is it *reliable*?

What if we asked an AI to:

- summarize a research article?
- plot the distribution of exam marks from a spreadsheet?
- produce Python code to do so?
- prove Fermat's Last Theorem?

Trust vs. Verification

- Can I **trust** what AI tells me?
- Can I **verify** what it tells me?

If we can verify a program is bug-free, we can be more aggressive with optimisations.

If we can check correctness of a mathematical proof, we don't have to worry about the collapse of the whole mathematical edifice.

Optimisations (Why it matters)

- Highly optimised code can be hard to maintain; clever tricks risk brittleness.
- If we can prove optimisations are semantically correct, we gain performance *and* robustness.

Strassen's Algorithm (2×2 case)

Multiply two 2×2 matrices using *fewer than 8 multiplications*.

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Key idea: reduce to 7 multiplications at the cost of extra additions.

Strassen: Strategy

Compute the following 7 products and 18 additions/subtractions:

$$
\begin{aligned}
M_1 &= (a+d)(e+h) \\
M_2 &= (c+d)e \\
M_3 &= a(f-h) \\
M_4 &= d(g-e) \\
M_5 &= (a+b)h \\
M_6 &= (c-a)(e+f) \\
M_7 &= (b-d)(g+h)
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

$$
C = AB = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}
$$

Why this is interesting

- Standard (2×2): 8 multiplications, 4 additions.
- Strassen (2×2): 7 multiplications, 18 additions.
- Recursively yields

$$\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81}) \quad \text{vs.} \quad \mathcal{O}(n^3).$$

This more complex algorithm yields better results.

The code is more efficient, but is also likelier to contain bugs!

Scale and Automation

What if we wanted to verify industrial-scale artefacts?

The Linux kernel? All the articles in the arXiv?

Doing this by hand is infeasible — use computers!

Disclosure: completing either goal above is science-fiction today.

λ-Calculus

λ-calculus is often used for software and mathematical verification.

- Origin: Alonzo Church, 1930s.
- Purpose: formal system for computation.
- Components: variables, abstraction, application.

Syntax

Start with a (potentially) infinite alphabet a, b, c, …, z, …

A λ-expression is anything built from the following rules.

- Variables are λ-expressions (e.g. d).
- Abstraction: $(\lambda x.\, Y)$ is a λ-expression, where x is a variable and Y is any λ-expression.
- Application: $(X\, Y)$ is a λ-expression, where X and Y are λ-expressions themselves.

Example:

$$(\lambda x.\, (\lambda y.\, z)\; y\; (\lambda a.\, x))$$

Alphabet a, b, c, $\ldots$, z, $\ldots$

- Variables are λ-expressions (e.g. d).
- Abstraction: $(\lambda x.\, Y)$ is a λ-expression.
- Application: $(X\, Y)$ is a λ-expression.

The game that we play is to give interpretations of λ-calculus.

Abstractions are abstractions *of functions*.

$(\lambda v.E)$ represents the function of the variable v whose "value" is E.

Note that v *may* appear in E (and, if it does not, then the function is constant).

$(\lambda v.E)$ represents the function of the variable v whose "value" is E.

Here are some more examples:

- $(\lambda x.x)$ is the identity function: any x maps to itself.
- $(\lambda x.y)$ is the constant function: any x maps to the fixed value y.
- $(\lambda x.(\lambda y.y))$ assigns to any x the identity function.

The final example can also be interpreted as a function of *two* variables x and y that always returns the second value.

Hence, the function $(\lambda x.(\lambda y.y))$ is the second projection function:

$$(x, y) \mapsto y.$$

Alphabet a, b, c, $\ldots$, z, $\ldots$

- Variables are λ-expressions (e.g. d).
- Abstraction: $(\lambda x.\, Y)$ is a λ-expression.
- Application: $(X\, Y)$ is a λ-expression.

Application is "evaluation of a function".

Seeing $(\lambda x.x)\, E$, we imagine that we *evaluate* the function $(\lambda x.x)$ (the identity) at the λ-expression E.

However, nothing in our rules tells us (yet) that indeed $(\lambda x.x)\, E = E$.

Computation rules

So far, λ-calculus is an abstraction of functions and their applications, without any of the *relations* that such rules should satisfy.

These relations bring in the *computational* aspect of the theory.

- α-conversion (rename bound variables): $(\lambda x.\, x)\, y \equiv (\lambda z.\, z)\, y$.
- β-reduction (substitution): $(\lambda x.\, E)\, F \;\to\; E\{F/x\}$.

α-conversion is mostly innocuous and generally helps with readability.

β-reduction is where our analogy with functions really takes off: now we can *define* functions, and also *evaluate* them!

Encoding via Church booleans

```
true  := (λt. λf. t)
false := (λt. λf. f)
and   := (λp. λq. p q p)
```

```
pair := λa. λb. λs. s a b
fst  := λp. p (λx. λy. x)
snd  := λp. p (λx. λy. y)
```

Some checks

```
and true q
   = (λp. λq. p q p) true q
   → (λq. true q true) q
   → true q true
   = (λt. λf. t) q true
   → (λf. q) true
   → q
```

```
and false q
   = (λp. λq. p q p) false q
   → (λq. false q false) q
   → false q false
   = (λt. λf. f) q false
   → (λf. f) false
   → false
```

Types

So far, we can compute.

However, we are missing a layer of information!

This is where the *types* come in.

Types run parallel to λ-expressions.
They assign meaning to the functions.

Types

The functions that we constructed were "just functions".
Functions usually have a specified domain and codomain.

E.g. "addition" is not just any function with 2 inputs and 1 value.

- Addition of natural numbers
- Multiplication of a scalar and a vector
- The pairing between a vector space and its dual

These are all functions taking 2 inputs and returning 1 value.

However, their domains and codomains distinguish them:

- $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$: Addition of natural numbers
- $\mathbb{R} \to V \to V$: Multiplication of a scalar and a vector
- $V \to V^* \to \mathbb{R}$: The pairing between a vector space and its dual

Getting creative with types

Let's define some types that encode *propositions*.

The *type* `True` is the type that has a unique term, denoted `()`.
The *type* `False` is the empty type with no terms.

We rig things up so that, exhibiting a term of any given proposition is equivalent to proving the proposition.

If `P` and `Q` are propositions, a term of type `P → Q` is a construction that

- takes a proof of the proposition `P` (namely, a term of type `P`) and
- returns a proof of `Q` (namely, a term of type `Q`).

A glimpse of the Curry–Howard correspondence

The Curry–Howard correspondence provides

- an encoding of *propositions* as *types* in such a way that
- *proofs* of propositions correspond to *terms* of their types.

Example: $(B \to C) \to (A \to B) \to (A \to C)$

$$\lambda f.\, \lambda g.\, \lambda x.\, f\,(g\,x)$$

is both function composition and a proof of implication transitivity.

Conclusion

λ-calculus and the Curry–Howard correspondence provide a machine-verifiable setting for checking proofs.

Similar setups also allows software verification, for instance, by proving theorems about specifications of algorithms.

The digitalization of reasoning allows for type-correct creativity as well as AI-driven development.

Thank you!

Questions?