&Truth

What Program Verification Does

by Ilya Sergey

October 29, 2026

At &Truth, we are building tools to make mathematical guarantees about software practical for everyday development. That includes software people have maintained for decades and code that AI systems have just generated. In either case, someone has an intention about the code: a calculation should produce the right answer, or a system should allow access only to authorized users. We want to express that intention precisely enough for a computer to check a proof that the code fulfills it.

What does such a proof actually establish? How can a piece of code, with variables that keep changing and loops that might run thousands of times, become the subject of a mathematical theorem?

In this post, we will look at the foundations and origins of mathematical reasoning about programs, which is at the core of &Truth's technology. We will start with a familiar mathematical function, then work through a small program that computes it and prove that it gives the right answer. Along the way, we will do a little arithmetic and read a few lines of code.

Mathematics as a Way to Describe the World

People have been doing mathematics for thousands of years, and the problems of connecting quantities to the world are much older than computers. For example, the Egyptian Rhind Mathematical Papyrus, more than 3,500 years old, contains practical problems involving arithmetic and geometry, including calculations for administrative and building work. What mathematics adds to such practical calculation is the habit of defining things precisely and proving claims about them.

Once we have defined a mathematical object, we can make statements about it and prove them. A proof shows that a conclusion follows from the definitions and assumptions we have chosen. This lets us reuse the conclusion whenever those assumptions hold, including in situations we have never encountered before. If the assumptions describe a real situation accurately, we can draw conclusions about it without testing each case.

Consider the *factorial* function. Its inputs are the natural numbers, written $\mathbb{N}$: $0, 1, 2, \ldots$. The factorial of n, written $n!$, is the product of the integers from 1 to n:

$$n! = 1 \times 2 \times \cdots \times n, \qquad 0! = 1.$$

Thus $3! = 6$, $4! = 24$, and $5! = 120$. The value at zero may look like an arbitrary convention, but it has a useful counting interpretation: there is exactly one way to arrange no objects, namely, the empty arrangement.

The definition also gives us a rule for moving from one factorial to the next:

$$(n+1)! = (n+1)\,n!.$$

For example, once we know that $4! = 24$, we obtain $5!$ by multiplying 24 by 5. Keep this rule in mind; it will be the mathematical fact that connects successive stages of the program we will study later.

Factorials are useful: they count arrangements. Six distinct people can stand in a line in $6! = 720$ ways: six choices for the first position, five for the next, and so on. The same reasoning works for any number of distinct objects.

From Counting to a Theorem

Suppose we choose two of those six people to represent the group, ignoring their order. Selecting the first two people in each line counts every pair $2!\,4!$ times: the pair can be ordered in $2!$ ways, and the remaining people in $4!$ ways. The number of pairs is therefore

$$\frac{6!}{2!\,4!} = \frac{720}{48} = 15.$$

If every pair is equally likely, a particular pair has probability $1/15$. Factorials help us calculate probabilities by counting possible outcomes. Such questions belong to *combinatorics*, the study of counting and arranging discrete objects. Its connection to probability is centuries old: Pascal's work of 1654 applied counting to games of chance.

The same reasoning gives a general theorem. Let us write $\binom{n}{k}$ (pronounced "n choose k") for the number of ways to choose k objects from n distinct objects, ignoring their order:

Theorem.

$$\forall n, k \in \mathbb{N}, \quad k \le n \implies \binom{n}{k}\, k!\,(n-k)! = n!.$$

The symbol $\forall$, called a *universal quantifier*, means "for every"; the arrow means "implies." The theorem covers every pair of natural numbers with $k \le n$.

The theorem turns counting into a routine calculation: divide $n!$ by $k!\,(n-k)!$. This matters when there are too many possible groups to enumerate, such as choosing ten people from a hundred. Proving it establishes that the division removes exactly the repeated arrangements. We can then use the formula for any allowed group size, including in probability calculations, without checking all the possibilities ourselves.

Having a Computer Check the Mathematics

We can encode this theorem in Lean, a language for expressing mathematical statements and proofs that a computer can check:

```
import Mathlib.Data.Nat.Choose.Basic

theorem choosing_and_arranging :
    ∀ n k : ℕ, k ≤ n →
      n.choose k * k.factorial * (n - k).factorial = n.factorial := by
  intro n k hk
  exact Nat.choose_mul_factorial_mul_factorial hk
```

Here `ℕ` denotes the natural numbers, `n.choose k` denotes $\binom{n}{k}$, and `n.factorial` denotes $n!$. The quantifiers and the condition `k ≤ n` express the same statement as above. Lean can check that the proof following `by` establishes it. Having computers check proofs in this way is a foundation for our work at &Truth: it gives us a way to check the reasoning behind a software guarantee.

To calculate the chance of drawing a particular group of ten people from a hundred, with all groups equally likely, we need a numerical answer. Our formula involves $100!$, a number with 158 digits. (More efficient methods avoid computing the full factorials; we will stick with the direct formula for this post.) Doing that arithmetic by hand is slow and prone to mistakes, so we write a program and let a computer do the work. But how do we know that the program calculates it correctly?

Reasoning about Programs, Mathematically

Proving that a program computes the intended result was already a concern in the early days of electronic computing. Alan Turing, one of the founders of computer science, addressed this problem in his 1949 paper "Checking a Large Routine". There, he proposed that programmers should accompany their programs with precise claims about intermediate stages of the computation. Someone checking the program could then verify those claims individually and use them to establish that the whole computation was correct, much as mathematicians check one another's proofs.

His example computes a factorial using repeated addition: a machine can calculate 4×6 by starting with 6 and adding another 6 three times. Turing deliberately leaves multiplication out of the routine. By combining additions and decisions about when to repeat them, it can carry out each multiplication needed for a factorial.

A machine does not acquire the meaning of $n!$ merely because we write that symbol on paper. It needs a sequence of operations it can execute. At the time of writing his paper, Turing also had no programming notation sufficiently familiar to his audience, so he used a flowchart instead. The paper mentions a further practical limit, that the result must fit within the available storage, giving the bound $n! < 2^{40}$. He sets this limit aside in the main argument.

Here is a modern rendering of his example in pseudocode with C-style syntax:

```
// Requires n >= 1.
// integer means an exact integer with no fixed size limit.
integer factorial(integer n) {
    integer r = 1;
    integer u = 1;

    while (r < n) {
        integer v = u;
        integer s = 1;

        while (s <= r) {
            u = u + v;
            s = s + 1;
        }

        r = r + 1;
    }

    return u;
}
```

An assignment such as `u = u + v` replaces the value in `u` with the sum of the current values of `u` and `v`. A `while` loop checks its condition, executes the code in braces if the condition is true, then checks again. Each pass through the loop body is an *iteration*. Here one loop sits inside another, so they are called *nested loops*.

We have kept the short variable names from the original paper. The outer loop advances `r` toward the input `n`. During each outer iteration, `v` saves the current value of `u`, while the inner loop repeatedly adds that saved value to `u`. The variable `s` controls those additions.

For a small input, we can follow the program by hand, doing a sort of "mental debugging." With `n = 3`, it starts with `r = 1` and `u = 1`. The first outer iteration adds 1 once, giving `u = 2`, then advances `r` to 2. The second saves 2 in `v` and adds it twice, taking `u` through 4 to 6. Now `r` reaches 3, and the program returns 6. That agrees with $3! = 6$, so our "test" passes. We want more: a guarantee that covers *every* allowed input.

Stating What a Program Should Do

The precise statement of a program's intended behavior is called its *specification*. For this program, we can state it as a theorem in plain English:

> **Program theorem.** For every positive integer n, if the program above finishes, it returns $n!$, assuming its arithmetic uses exact integers without a fixed size limit.

Turing's paper also establishes *termination*, meaning that the program eventually finishes; we omit that argument to keep this post short.

The assumptions of the theorem are important. For example, we have chosen positive inputs to match the domain used in the adapted presentation of Turing's proof. We have also specified exact arithmetic. An ordinary C `int` has a fixed range, so this theorem about the pseudocode would need extra assumptions before it could justify a real C implementation. In particular, we would have to account for that range and show that the intermediate calculations stay within it.

What remains for us is to prove that the code, with its changing variables and loops, correctly implements the mathematical factorial function.

Why the Program Computes a Factorial

The informal proof of the desired specification goes as follows.

First, let's notice that the program builds its answer one factorial at a time. At the beginning, `r = 1` and `u = 1`, so `u` already holds $1!$.

Now suppose `u` holds $r!$ and we need the next factorial, $(r+1)!$. That requires $r+1$ copies of $r!$. We already have one copy in `u`, so we need to add `r` more. The variable `v` saves the old value while `u` grows.

Next, let's look at what `s` counts. It starts at one, matching the single copy already in `u`. Each inner iteration adds another copy from `v` and increases `s` by one. After each iteration, `s` therefore tells us how many copies have accumulated. For example, starting with $3! = 6$, the additions take `u` through 12, 18, and 24, while `s` advances from one to four.

When the inner loop ends, `s` has just moved from `r` to `r + 1`, making `s <= r` false. We now have exactly $r+1$ copies of $r!$, so

$$u = (r+1)\,r! = (r+1)!.$$

The program then increases `r` by one. Once again, `u` holds the factorial of `r`, ready for the next outer iteration. The same reasoning applies at every stage, starting from $1!$, so it covers any positive input.

The formal argument follows from the following two claims. At the start of every outer iteration, `u` holds $r!$. Whenever the inner loop checks its condition, `v` holds $r!$ and `u` holds $s \cdot r!$. Relationships between variables like these, which hold each time a loop comes round, are called *loop invariants*. They are exactly the definite assertions Turing asked programmers to write down: his paper tabulates them, and a checker verifies each one separately, much as we have just done.

Finally, `r` advances one at a time and the outer loop runs only while `r < n`. When it exits, `r` has reached `n` exactly, and `u` holds the answer $n!$.

Notice why the inner condition includes equality: changing `s <= r` to `s < r` loses one addition. For input 2, the changed program returns 1. A test with input 1 would not notice, since both versions return 1 there. This error is easy to make, and a rigorous proof would catch it.

From a Small Program to Useful Guarantees

This is *program verification*: proving that a program satisfies a precise statement of its intended behavior. Our proof connected the operations in the code to a mathematical definition, covering *every* input allowed by the assumptions, unlike tests that only check a program's outcome on a finite number of concrete inputs.

Even for this tiny and somewhat impractical program, the proof took several paragraphs. Doing all that work by hand for every function, as Turing suggested 77 years ago, would make verification very expensive. Much has changed since then: tools called *program verifiers* can derive the mathematical claims needed to establish a program's specification and construct proofs of many of them automatically, or with a bit of help from a human or an AI. In our approach, Lean checks whether each proposed proof establishes the stated specification.

To support verification of realistic software, we must make the connection between a *program* and its *mathematical encoding* faithful, and much of the unavoidable work needs to be done to implement this connection. In particular, the mathematical model of a modern programming language must cover the operations the code uses, the representation of its memory, and tricky features such as concurrency and input/output, while the specification language must be able to express the properties that users care about.

At &Truth, we are building tools that automate this reasoning about user intent for existing software and AI-generated code. Our aim is for developers to be able to express what their software should do and obtain independently checkable evidence, such as a formal proof in Lean, that it does so, at a cost that makes specification and verification routine parts of development.

In future posts, we will discuss how to express more interesting aspects of program behavior and the methods that automate the reasoning we have just done by hand. The question behind Turing's small example will stay with us: how do the instructions we give a machine implement the intention stated by a human? Learning to answer it precisely is the first step toward making that answer available for the software we use every day.