

The Mathematics of Double-Checking Programs

Alexander V. Gheorghiu AMIMA, University College London and University of Southampton

Imagine you own a large company delivering essential services, akin to industry giants like Amazon, Google or Meta. Your business boasts an enormous client base, thousands of servers delivering these services and generates millions (if not billions) of pounds in revenue per year. To keep up with the ever-evolving tech landscape, you constantly need to add new features, correct bugs and otherwise work on the system delivering your service. This relentless drive is powered by a dedicated fleet of engineers maintaining the system.

It is a Friday afternoon, and Bob, one of your diligent software engineers, is working late at the London office to meet a critical deadline. In an effort to fix a minor bug, he uploads some new code to the server. Business continues to run smoothly for now, but a crucial question looms large: How can you be sure that Bob's code will not crash the system? After all, he's human, it's late and a small mistake could easily slip through. With so much money and your reputation at stake, the risk of your service going down over the weekend is a chilling prospect.

This scenario underscores a common dilemma faced by many businesses today. In a world increasingly driven by software, there is a pressing need to ensure that code performs as intended. Companies like Amazon, Google and Meta invest substantial resources to mitigate the risk of system failures. As the digital economy expands, the ability to manage and minimise these risks is becoming ever more critical.

There are not only financial concerns for big business but safety and privacy ones for individuals too. For example, how can you trust that your bank account is safe? Can you trust that the online login system for your bank works correctly so that no one who should not have access to your account does? Ensuring system stability is not just about maintaining operations; it is about safeguarding trust, reputation and reliability in an interconnected world where downtime can have far-reaching consequences.

A pressing example was the *CrowdStrike Incident* on 19 July 2024. The cybersecurity firm CrowdStrike released a flawed update for its Falcon Sensor security software, leading to significant disruptions on Microsoft Windows computers using the software. This update caused around 8.5 million systems to crash and become unable to restart properly, marking what has been described as the largest outage in the history of information technology.

There is an apparently obvious answer to the problem of program and system correctness. Just *double-check* that Bob's code is correct before implementing it. This is surprisingly subtle. What does it mean to double-check it? One answer is to implement an algorithm that checks that code is OK before it is committed to your operations. Unfortunately, providing such an algorithm is a mathematical impossibility.

A *program* is the specification of a process by which an input yields an output, called computation. For example, the program ' $x + 1$ ' takes an integer as an input and outputs another integer; namely, one more than the input. There are many equivalent accounts of what it means to be a computation, perhaps the most celebrated is the concept of a *Turing machine* [1].

Alonzo Church in 1935 and Alan Turing in 1936 proved that there are limits to what one may actually say about and do with computations. An important consequence of which was a

negative answer to the *Entscheidungsproblem* set by Hilbert and Ackermann in 1928, Is there an algorithm that considers, as input, a statement and answers 'yes' or 'no' according to whether the statement is universally valid?

Let a *semantic* property of a program be one about the program's behaviour (for example, Does it always output 0?). This is in contrast to a *syntactic* property such as, Does the program contain an if-then-else statement? Call a property *trivial* if it is either true or false for every program. In his 1951 doctoral thesis, Henry Rice gave the following generalisation of the limitative results by Church and Turing:

Theorem (Rice [2]). *All non-trivial semantic properties of programs are undecidable. In other words, there is no algorithm that can determine whether a given program possesses a specified non-trivial semantic property.*

The presence of bugs is a semantic property. Consequently, Rice's theorem tells us that it is impossible to write a program that can automatically check for the absence of bugs in other programs. This would require taking a program and a specification as input and determining whether the program satisfies the specification. This does not mean it is impossible to prevent certain types of bugs, but the general problem remains unsolvable.

So, what are we to do? Of course, you can simply ask Bob and his colleagues to double-check each other's code before submitting it. But, how do you ensure that there is an adequate standard for them to do that? And, if one check is good, are two better? What you really desire is a *mathematical* proof that Bob's code is correct. To this end, you hire Alice.

Suppose Bob had the following specification for a project, 'Write a function that computes the integer part of the square root of a non-negative integer x .' Bob chooses to use a binary search and writes the following Python code B (note that $//$ denotes integer division):

```
def sqrt(x):
    lt, rt = 0, x
    while lt <= rt:
        y = (lt + rt) // 2
        if y * y <= x < (y + 1) * (y + 1):
            return y
        elif y * y < x:
            lt = y + 1
        else:
            rt = y - 1
```

Inspecting Bob's code, one might be convinced that it is correct, but it is Alice's job to *prove* that the program satisfies the brief.

We can think of the code as a sequence of arithmetical operations. In doing so, the problem becomes analogous to checking a sequence of equations. When doing such checks, one must be careful to apply the laws of arithmetic correctly. For example, consider the following sequence of arithmetical steps:

$$\begin{aligned} a = b &\Rightarrow a^2 = ab \\ &\Rightarrow a^2 - b^2 = ab - b^2 \\ &\Rightarrow (a - b)(a + b) = b(a - b) \\ &\Rightarrow a + b = b \\ &\Rightarrow b + b = b \\ &\Rightarrow 2 = 1. \end{aligned}$$

This argument uses many familiar laws of arithmetic, but it has clearly gone awry. The subtlety is in dividing by the factor $(a-b)$. Such divisions are allowed only when the factor is non-zero, but this is not the case since we assumed $a = b$.

To check Bob's code, Alice needs to establish some fixed laws governing how programs manipulate expressions about the state of the variables. These laws need to be precisely stated so that no unsound inferences such as the division-by-zero example above are admitted. Having such laws, Alice merely needs to follow how the expression $x > 0$ evolves as Bob's program executes and verify that the property $y^2 \leq x < (y+1)^2$ holds at the end.

This landmark observation was made by Tony Hoare [3] in 1961. He came up with the *Hoare triple*,

$$\{P\} C \{Q\}$$

in which C is some program code, P is a logical assertion about variables before executing C and Q is a logical assertion about the program state after executing C . We call P and Q the pre- and post-conditions of the triple, respectively. The laws governing these triples are known as *Hoare logic*. An example of a law in Hoare logic is as follows:

$$\{P[E/x]\} x := e \{P\}.$$

This says, 'If P is true with term E (representing the number e) substituted for the variable x , then after the command $x := e$ is executed, P will be true'. Since programs usually contain sequences of commands, we also have the following transitive law:

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}.$$

This expresses the following: 'If C_1 executes with pre-condition P and yields post-condition Q and C_2 executes with pre-condition Q and yields post-condition R , then $C_1; C_2$ executes with pre-condition P and yields post-condition R .

Thus, to check that Bob's code B behaves as desired, Alice needs to show only that the following Hoare triple holds:

$$\{x > 0\} B \{y^2 \leq x < (y+1)^2\}.$$

That is, 'If x is a positive number, then executing B yields a number y such that x is between (inclusive of the lower bound, exclusive of the upper) y^2 and $(y+1)^2$ '.

Alice writes a *proof* for the correctness of Bob's code. Rather than using the laws of arithmetic, Alice does this using Hoare logic.

A standard way of doing such 'verification' proofs is by annotating the code line by line with the post-condition (using the laws of Hoare logic). One first writes the pre-condition and then writes the post-condition after each line with a command. Thus, Alice could write the following:

```
def sqrt(x):
  #Precondition: x >= 0
  lt, rt = 0, x
  #{ 0 <= lt <= rt <= x }
  while lt <= rt:
    y = (lt + rt) // 2
    #{ 0 <= lt <= y <= rt <= x }
    if y * y <= x < (y + 1) * (y + 1):
      #{ y^2 <= x < (y + 1)^2 }
      return y
    elif y * y < x:
      #{ y^2 < x }
```

```
lt = y + 1
#{ 0 <= lt <= rt <= x }
else:
  #{ x <= y^2 }
  rt = y - 1
  #{ 0 <= lt <= rt <= x }
```

Importantly, the condition above the while loop

```
#{ 0 <= lt <= rt <= x }
```

is a loop *invariant* and is analogous to an induction invariant. That is, it is true before and after each iteration of a loop in a program.

This process may be articulated to such precision that it can be executed by a computer. Alice needs to specify only the laws and the pre- and post-conditions suitable for the brief. This method enables *automatic* verification of thousands of lines of codes. It is a seminal piece of mathematics that supports the vital role of technology in the modern world. However, today's businesses do not run on thousands of lines of code but rather on tens of thousands, hundreds of thousands or even *millions* of lines of code. Hoare logic on its own does not scale to meet industry needs.

In 1999, through foundational work in formal logic, Peter O'Hearn and David Pym [4] invented the 'separating' conjunction $*$, which led to a refinement of Hoare logic called *separation logic* [5]. The separating conjunction renders program verification compositional: one does not need to tackle all million lines of code at once, as one can look at fragments of the code and fragments of the variables that it touches. It is governed by the law

$$\frac{\{P\} C \{P\}}{\{P * R\} C \{P * R\}} (\text{mod}(C) \cap \text{fv}(R) = \emptyset).$$

This is known as the *frame* rule, named after the frame problem in artificial intelligence [6]. It says that a program that executes safely in a small state (satisfying P) can also execute in any bigger state (satisfying $P * R$) and that its execution will not affect the additional part of the state (and so R will remain true in the post-condition). This rule has the side condition that the variables modified by C (i.e., $\text{mod}(C)$) are disjoint from the free variables in R (i.e., $\text{fv}(R)$).

Separation logic and its subsequent developments have been scaled to meet the rigorous demands of modern technology. Consequently, your business is safer, and your bank is more secure. Rice's theorem still looms large, so no guarantee is possible, but while these double-checking proofs using modern techniques in logic and informatics are being implemented, we can sleep more soundly. Indeed, entire companies, such as Facebook's *Infer*, have been established to bring the rigour of mathematical proof to industry.

The power of separation logic lies in its unique operator, the separating conjunction $*$. This operator originates from a nuanced observation about the relation between implication (\rightarrow) and conjunction (\wedge , logical 'and') in formal logic. We may write $P \vdash Q$ to denote that Q is a logical consequence of P . The connection is the following:

$$P \vdash Q \rightarrow R \quad \text{iff} \quad P \wedge Q \vdash R.$$

This is closely related to the notion of a Galois connection in order theory and adjunction in algebraic topology.

By making subtle modifications to the definition of implication (\rightarrow), one derives a special kind of implication called the ‘magic wand’ (\multimap). The difference between \rightarrow and \multimap is fundamentally about how one ‘discharges’ assumptions in an argument. O’Hearn and Pym [4] derived \multimap from the magic wand by the adjunction above:

$$P \vdash Q \multimap R \quad \text{iff} \quad P * Q \vdash R.$$

The subtle difference between \rightarrow and \multimap results in a significant distinction between \wedge and $*$.

Let \mathfrak{M} be a model of a system, such as the memory in the computer on which B is running. Let P and R be properties, such as ‘ $x = E$ ’, meaning ‘memory unit x has value E ’. If $P \wedge Q$ is true in \mathfrak{M} , then each of P and Q is true in \mathfrak{M} . However, if $P * R$ is true in \mathfrak{M} , then there exists a decomposition of \mathfrak{M} into parts \mathfrak{M}_1 and \mathfrak{M}_2 such that P is true in \mathfrak{M}_1 and R is true in \mathfrak{M}_2 . This property enables the frame rule in separation logic.

The separating conjunction and magic wand have many other uses than program verification, but we defer discussion to another time.

In practice, while powerful, formal verification of the kind described here is highly resource-intensive and time-consuming, often requiring significant computational effort and expert knowledge. Therefore, it is limited to those companies described above with the need and resources to use it. Additionally, it may struggle to catch certain types of bugs, particularly those related to performance, real-world data or hardware-induced issues, as it focuses primarily on logical correctness rather than dynamic behaviour. It is for these reasons that not every tech company uses it. Nonetheless, as the mathematical discipline of logic has matured over recent years, formal verification has become more and more tractable.

Other ways of checking correctness are also available. For example, tech companies typically employ protocols such as *unit* and *integration* testing. Unit testing involves testing individual components or functions of a program or system in isolation to

ensure that they work as expected. By contrast, integration testing focuses on verifying the interactions between different modules or components to ensure that they function correctly together by catching issues that may not be apparent during unit testing.

Over the last century, logic as a mathematical discipline has evolved significantly. It has expanded beyond set theory, theories of arithmetic and model theory. Rather than being confined to the foundations of mathematics, it has become a versatile area in applied mathematics, enabling representation and reasoning about *systems*, broadly conceived. The concept of ‘system’ here encompasses, inter alia, transport networks (airports, train lines and bus routes), telecommunication networks (the internet, postal services and radio), manufacturing, logistics and security policies. The question remains: Can we trust these systems to function as intended? The answer lies in rigorous verification. We need to double-check these systems. We need *proofs* of their correctness.

REFERENCES

- 1 Turing, A. (1936) On computable numbers, with an application to the Entscheidungsproblem, *J. Lond. Math. Soc.*, vol. 58, pp. 345–363.
- 2 Rice, H. (1953) Classes of recursively enumerable sets and their decision problems, *Trans. Am. Math. Soc.*, vol. 74, no. 2, pp. 358–366.
- 3 Hoare, C. (1969) An axiomatic basis for computer programming, *Commun. ACM*, vol. 12, no. 10, pp. 576–580.
- 4 O’Hearn, P. and Pym, D. (1999) The logic of bunched implications, *Bull. Symb. Log.*, vol. 5, no. 2, pp. 215–244.
- 5 Reynolds, J.C. (2002) Separation logic: A logic for shared mutable data structures, *Proc. 17th Annu. IEEE Symp. on Logic in Computer Science*, Copenhagen, Denmark, pp. 55–74.
- 6 McCarthy, J. and Hayes, P.J. (1981) Some philosophical problems from the standpoint of artificial intelligence, in *Readings in Artificial Intelligence*, Morgan Kaufmann, Burlington, pp. 431–450.