Computational Logic: Its Origins and Applications

Author for correspondence:
Prof. Lawrence C. Paulson FRS,
University of Cambridge
e-mail: lp15@cam.ac.uk

*

Lawrence C. Paulson FRS

Computer Laboratory, University of Cambridge, England, e-mail: lp15@cam.ac.uk

Computational Logic is the use of computers to establish facts in a logical formalism. Originating in 19th-century attempts to understand the nature of mathematical reasoning, the subject now comprises a wide variety of formalisms, techniques and technologies. One strand of work follows the "LCF approach" pioneered by Robin Milner FRS, where proofs can be constructed interactively or with the help of users' code (which does not compromise correctness). A refinement of LCF, called Isabelle, retains these advantages while providing flexibility in the choice of logical formalism and much stronger automation. The main application of these techniques has been to prove the correctness of hardware and software systems, but increasingly researchers have been applying them to mathematics itself.

1. Introduction

Computers control everything from heart pacemakers to aircraft, putting everyone at risk when they fail. Computer-related failures regularly cause disruption, damage and occasionally death. A recent example is the WannaCry ransomware attack, which hit computers around the world, notably in the National Health Service [78]. This attack was made possible by a specific defect, MS17-010, affecting several versions of the Windows operating system. In 1994, a floating-point division error in the Pentium processor forced Intel to recall millions of chips at a cost of $475 million [57]. Many such failures are caused by carelessness and could be prevented by the introduction of more disciplined development methods. The ultimate discipline is to use formal logic and mathematics to prove the correctness of designs. This is called *formal verification*.

Formal verification typically relates a *formal specification* to a *model implementation*. That is, we start with a complete description of all services to be delivered by the system (the *specification*) and prove that it is satisfied by the model implementation, where both have been expressed in some logical formalism. A proof that all services will be delivered includes a proof that no crashes or other failures will prevent this. However, any formalised implementation must leave out some details of the actual implementation: for example, finite-precision arithmetic may be assumed to be exact, encryption to be unbreakable or hardware components to be infinitely fast. (Readers may remember schoolbook physics problems involving frictionless surfaces, or when friction is included, it is assumed to be linear.) If the model of the implementation is too unrealistic, proofs about it will overlook certain failure modes. No model can include all real-world details, but it is crucial to include the aspects of the implementation that we are most concerned about. For example, in computer security, many systems can be broken without defeating the encryption itself [51]. Then it makes sense to assume encryption to be unbreakable if that simplifies the proofs. Validating the model is complementary to the formal proofs and could involve a variety of methods, ranging from professional judgement to systematic testing. (It is a misconception to think that formal proof eliminates the need for testing.)

Formal verification requires the support of specialised computer software. *Computational Logic* is concerned with formal logic as implemented on computers, where it is frequently applied to solve problems related to computation. However, formal verification is increasingly being applied to systems that interact with the real world and deal with phenomena modelled by differential equations, or that have to achieve goals expressed in terms of probability. Therefore, it has become necessary to formalise mathematical knowledge.

2. A Brief History of Formal Logic

A *logical formalism* is a precisely defined symbolic language that includes logical primitives such as "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$), "implies" ($\rightarrow$), "for all" ($\forall$) and "there exists" ($\exists$). It may include the ability to define new symbols as abbreviations for other expressions. It will include rules for making simple logical deductions: for example, from $\phi \wedge \psi$, the conclusions ϕ and ψ can both be derived; from ϕ we can derive $\phi \vee \psi$. From such basic primitives, vast swathes of mathematics can be derived.

Formal logic emerged from philosophy in the 19th century. Gottlob Frege's *Begriffschrift* (concept language) was the first recognisably modern treatment of logic. His work introduced the project of reducing the whole of mathematics to logic [10,21]. But Frege's work ended with the discovery of Russell's paradox:

> Let R be the set of all sets that are not members of themselves. Then is R a member of itself?

If R is a member of itself (written $R \in R$) then it cannot be a member of R, and therefore $R \notin R$; on the other hand, if $R \notin R$ then $R \in R$. We get a contradiction either way. Note that as long as we confine ourselves to English, Russell's paradox may resemble amusing puzzles such as

"This statement is false", but once we use a formal language with strict rules, we are in trouble. For $R \notin R$ and $R \in R$ are both true, from which all other statements can be proved using basic logic. There are many other paradoxes [77, p. 60], e.g. Burali-Forti, which involves constructing the greatest ordinal number Ω, but then $\Omega + 1 < \Omega$. As Gödel remarks,

> By analysing the paradoxes to which Cantor's set theory had led, [Russell] freed them from all mathematical technicalities, thus bringing to light the amazing fact that our logical intuitions (i.e. intuitions concerning such notions as truth, concept, being, class) are self-contradictory. [21, p. 452]

From that moment, the development of formal logic was determined by the approach taken to solving the paradoxes. *Set comprehension* — the idea that every property ϕ created a set, $\{x \mid \phi(x)\}$ — somehow had to be constrained to deny the existence of $\{x \mid x \notin x\}$, the Russell set. One approach was to restrict comprehension to some *already existing* set, say A, yielding the smaller set $\{x \in A \mid \phi(x)\}$: the elements of A satisfying the property ϕ. The universe of sets is not itself a set, and it is built up from the empty set using a few primitives such as union and powerset. This approach leads to modern axiomatic set theory [11,48].

Another solution to the paradoxes involves the notion of *types*. Whitehead and Russell's original conception of type [77, p. 37] was rather unclear [21], but by 1940, Church [14] had developed types to a form that survives today. Rather than postulating one universal collection of everything that exists, mathematical objects are classified into distinct types, representing smaller collections. Every construction has a fixed type. When forming a set, all the elements must have the same type, say τ, and the set itself will have a different type, which we may write as τ set. Thus the fatal proposition $R \in R$ cannot even be written, because if R has type τ then it cannot have type τ set at the same time.

Types seem to be unpopular with mathematicians, and it is usual to assert that mathematics is founded on set theory. Whitehead and Russell [77], provided no notation or symbols for types, making them invisible, while Church [14] had one single type ι of individuals encompassing all objects that were neither sets nor functions. However, Church's ideas entered into computer science, particularly through his collaboration with Alan Turing, and by the early 1960s, the logical notion of type was starting to get conflated with programming language features (previously called "modes") that distinguish between integer and floating-point arithmetic, for example. The ubiquity and utility of types in computer programming may be one reason why computer scientists tend to prefer elaborately typed formalisms.

Yet another solution to the paradoxes was the philosophy of *constructive mathematics* or *intuitionism* [40] . Briefly, this is the view that mathematical objects exist only in our imaginations, in contrast to the Platonist or realist view that mathematical objects have their own reality [22, p. 323]. Intuitionism demanded a new treatment of logic, in which the disjunction $\psi \vee \phi$ could be true only if we *knew* which of the two statements, ψ or ϕ, was true. Many of the basic tautologies of Boolean logic must then be rejected, starting with $\psi \vee \neg\psi$ (as we have no general way of knowing which alternative holds) and with $\neg\neg\phi$ a weaker statement than ϕ. At the level of quantifiers, $\exists x \, \phi(x)$ could be true only if we *knew* some specific value a for which $\phi(a)$ held. For that reason, $\exists x \, \phi(x)$ was a stronger claim than $\neg[\forall x \, \neg\phi(x)]$, although in classical logic the two formulas are equivalent.

Intuitionism has not caught on with mainstream mathematicians. However, there are strong links between intuitionistic logic and computation: the knowledge mentioned in the previous paragraph had to be computable. Recent decades have seen the introduction of formalisms [15, 53] that identify intuitionistic formulas with elaborate types: every logical symbol has an equivalent at the type level, and to prove a proposition amounts to exhibiting an element of the corresponding type. These formalisms are highly expressive and typically, all expressions denote computable functions. And there is a direct identification between logical propositions and types [54]. For example, proving the formula $(\forall x \in A)(\exists y \in B) \, \phi(x, y)$ yields a computable function f such that if $x \in A$ (that is, x belongs to type A) then $f(x)$ returns some $y \in B$ paired

with some sort of "evidence" affirming that the property $\phi(x, y)$ holds.[1] Then we would have shown that f had the type $(\prod x \in A)(\sum y \in B)\,\phi(x, y)$, which we can identify with the formula above. Here, $\prod$ and $\sum$ are the type-level counterparts of the quantifiers $\forall$ and $\exists$. The elements of $(\prod x \in A)B(x)$ are functions f that map any given $a \in A$ to some $f(a) \in B(a)$, generalising the function type $A \to B$ by allowing the result type to depend on the value of the argument; similarly, the elements of $(\sum x \in A)B(x)$ are pairs $\langle a, b \rangle$ where $a \in A$ and $b \in B(a)$, generalising the Cartesian product $A \times B$. This "propositions as types" paradigm is the focus of much current research; see Wadler [74] for a detailed historical introduction.

Formal verification today can therefore be done within a wide variety of logical formalisms, which include

  (i) those based on axiomatic set theory, and with no concept of type;
 (ii) the *simple* type theory of Church, where a few primitive types can be combined using operators such as $\times$ (Cartesian product), $+$ (disjoint sum), $\to$ (function space) and the set type constructor mentioned above;
(iii) *dependent* type theories where types can take parameters, as in $B(x)$, and with variable binding as in $(\prod x \in A)\,B(x)$, thereby allowing the type of $n \times n$ matrices, for example.

The typed approaches have received the most attention in computational logic, and below we shall focus on simple type theory. The third approach is also the focus of much research.

There are many other formalisms (for example, modal and relevance logics) and computational logic techniques [43] not covered below. In particular, *model checking* techniques [44] are widely used. They can automatically verify system properties expressed in temporal logics (for reasoning about time). However, they rely on state enumeration rather than proof construction, so are out of scope here.

3. Mechanised Logic: the LCF Tradition

If we want to prove theorems in a formal logic, computer power is necessary. Whitehead and Russell's development of basic logic, classes and relations only reaches (a precursor to!) 1+1=2 after hundreds of pages [77, p. 360]. Mathias [55] has pointed out that the representation of the number 1 as defined by Bourbaki expands to 4,523,659,424,929 symbols. Better choices can mitigate such issues, but the fact remains that formal proofs are extremely long and detailed.

Ideally, we would like the computer to prove our theorems automatically. If that is impossible, then we would like to give only strategic advice, and let the machine work out the details. As a general rule, greater automation requires the choice of a less expressive formalism. *Boolean logic* (where we only have and, or, not) is decidable, so in principle, all questions can be settled automatically. The problem is known as SAT (for Boolean SATisfiability) and although a general solution is thought to be exponential in the number of Boolean variables, highly efficient heuristic approaches exist. In a dramatic demonstration of this technology, Heule and Kullmann [39] solved the Boolean Pythagorean Triples problem, an open problem in Ramsey Theory, generating a 200TB proof.

Our logic becomes more expressive if we introduce the quantifiers "for all" and "there exists". If we forbid quantification over functions and sets, this is called *first-order logic*. This theory is *complete*: all theorems can be proved, given unlimited resources. But there is no general way to determine how much effort might be required or whether the claim is false. Automatic theorem proving in first-order logic is a significant field in its own right; one of its main achievements, though 20 years ago, is the solution of the Robbins Conjecture [56]. Incidentally, two early titans of computational logic — Hilary Putnam [16] and J. Alan Robinson [68] — were trained as philosophers, and Putnam is chiefly remembered for his essays.

Unfortunately, fully automatic theorem proving cannot meet the needs of verification. Arithmetic, functions and sets are necessary for specifying even simple systems. No automatic

---

[1]Thus, a form of the axiom of choice is actually provable in this system [54, p. 516].

procedure can decide arbitrary statements involving integer arithmetic: this is Hilbert's famous *Entscheidungsproblem*, which was settled by Turing [71]. Another reason is that specifications may extend to hundreds of pages, and fully automatic theorem proving does not scale well. Interactive methods are the only feasible way to prove theorems based on huge specifications.[2] And the name most associated with interactive theorem proving is Robin Milner (FRS 1988).

Milner was interested in an obscure formalism called Logic for Computable Functions (LCF), and built a simple proof checker while at Stanford University. It had two major drawbacks: proofs often required long sequences of repetitive, obvious steps, and the proofs themselves took up too much memory. Upon taking a position at the University of Edinburgh, he created an ambitious new system called Edinburgh LCF [28], incorporating several innovations. One was to provide the user with a specialised programming language, called ML (for metalanguage) so that users could extend the system with their own code. That would enable them to deal with any issue of repetitive steps by coding their own automation. Another was to eliminate the storage of proofs through a software technique known as *abstract data types*. By reserving the privilege of creating theorems to a small *kernel* of code, the need to store proofs was eliminated. User-supplied automation, however faulty, could never cause Edinburgh LCF to proclaim an invalid theorem. Edinburgh LCF also introduced an entire framework of techniques and terminology:

- *theory files* containing specifications and definitions of the concepts of interest: theories can build on other theories, allowing hierarchical developments of any size.
- *backward proof*: one begins with a statement of the desired result and successively reduces it (using *proof tactics*) to simpler statements until all have been proved.
- *tacticals* for combining proof tactics in simple ways, for example, repetition or executing a series of tactics
- built in *automation*, for example to perform routine simplification
- along with the ability to *extend* this automation by writing additional ML code.

Today, almost all of the main interactive theorem provers incorporate these techniques, and in many cases, actual code from Edinburgh LCF. ML turned out to be useful for programming in general, and has had a huge impact on the design of modern programming languages such as Haskell, OCaml and Scala. The meanings of LCF and ML go well beyond their literal meanings as acronyms.[3]

Another significant figure in interactive theorem proving was Michael JC Gordon (FRS 1994). He created the HOL system [27], which survives to this day in several variants, including HOL4 and HOL Light [34]. All of these are descendants of Edinburgh LCF, which he also helped to create [28]. Even more significant is Gordon's development of techniques for verifying computer hardware; when he started working on this topic in the early 1980s, essentially all verification research was focused on software. Gordon developed techniques that scaled from a single-transistor device to an entire computer. His key insight was that the right formalism for this task was *simple type theory*, also known as *higher-order logic*, hence HOL.

The choice of higher-order logic was radical at the time. First-order logic was dominant, thanks perhaps to its strong theoretical properties such as completeness. Gordon noted that such properties were irrelevant to verification, and recognised that the additional expressiveness of higher-order logic was necessary [25]. His choice represented a return to the stronger logics of the early twentieth century. With his students, Gordon extended his techniques to cover a variety of digital systems, including floating-point hardware, probabilistic algorithms, and many other applications. Typical of these is Harrison's verification of a floating-point algorithm for the computation of the exponential function [35]. The proof required a complete formalisation of the IEEE floating-point standard, in all its complexity, as well as the error analysis for the algorithm itself.

---

[2]Interactive theorem provers are also called *proof assistants*.
[3]ML can also refer to machine learning.

While on the topic of radical choices, we must not overlook research into constructive type theories. Inspired originally by the work of Martin-Löf [54] and by Coquand and Huet [15], researchers built a series of interactive theorem provers, all following the Edinburgh LCF paradigm. For reasons of space, I will only mention Coq [6], which has become perhaps the most popular interactive theorem prover in the world. Typical of all these approaches is that through dependent types, the type system carries out much of the reasoning burden that would otherwise be done by explicit proof steps. Among the landmark achievements carried out using Coq are the formal verification of a C compiler [50] and the formalisation of the odd order theorem [24]. A full treatment of this line of work would require a separate article.

4. A New Theorem-Proving Architecture: Isabelle

During the 1980s, many different logical formalisms were competing for researchers' attention. In addition to LCF, type theories were evolving rapidly, along with a wide variety of formalisms for proving software correct. One drawback of the LCF approach was its requirement that every primitive inference rule had to be implemented as program code, and then a second time as a proof tactic. An error in the latter code could prevent the user from obtaining the theorem they thought they had proved, while an error in the former code could allow the system to assert false statements as true.

To understand this drawback in more detail, consider one of the simplest logical inference rules: if ϕ and ψ are theorems, then so is their conjunction, $\phi \wedge \psi$. It is often written as

$$\frac{\phi \quad \psi}{\phi \wedge \psi.}$$

However, note that mathematical proofs frequently involve temporary assumptions, such as when we consider the two cases of whether some integer n is even or odd. Each of the two cases is proved with the help of the corresponding assumption about n. So people generally adopt a so-called *natural deduction* calculus, where all reasoning takes place within a *context*: a set of formulas $\theta_1, \ldots, \theta_n$ temporarily assumed to be true.[4] In order to minimise the need for subscripting, we typically use the large Greek letters Γ and Δ to stand for contexts. If we chose to make this explicit, our inference rule could be written (forming what is known as a *sequent calculus*)

$$\frac{\Gamma \Longrightarrow \phi \qquad \Delta \Longrightarrow \psi}{\Gamma, \Delta \Longrightarrow \phi \wedge \psi.}$$

Note that the two contexts are combined when we form the conclusion, since it depends on everything assumed by the premises. And this is the simplest possible example. A number of rules modify contexts by "discharging" assumptions. The treatment of quantified variables introduces further technicalities.

To implement this rule using the LCF architecture (which is also adopted by the HOL family and Coq), we begin by designing data structures that can represent the full syntax of our formalism. Then we collect all the inference rules that need to be implemented; taken together, they will define the abstract data type `thm`, the type of theorems. Each rule will take the form of a function coded in ML. The function for the rule above will take as its arguments the two premises, namely $\Gamma \Longrightarrow \phi$ and $\Delta \Longrightarrow \psi$. This rule is simple, but in other cases, the code must make syntactic checks on the premises to ensure that they have the correct form, and if not, signal an error. Finally, the function returns $\Gamma, \Delta \Longrightarrow \phi \wedge \psi$ as a value of type `thm`, thus certifying it to be a theorem. This particular function is called `CONJ` in LCF and its descendants.

Defining type `thm` as outlined above implements a logic in ML, but a usable system must also support backward proof: the ability to prove a statement of the form $\phi \wedge \psi$ by proving the two parts separately. So we need to write another function, called a *tactic*, which essentially executes the inference rule in reverse. Given the goal of the form $\Gamma \Longrightarrow \phi \wedge \psi$, it returns the two separate

---

[4]Mathematical proofs also introduce temporary quantities, as when we allow x to denote a root of some polynomial. Therefore contexts may include a string of bound variables, but the details [60] are too complicated to present here.

subgoals $\Gamma \Longrightarrow \phi$ and $\Gamma \Longrightarrow \psi$ (and note that here we have one single context, Γ, since we are working backwards).

These subgoals are not theorems but merely statements to work on next. In order to yield a theorem at the end, the tactic also needs to return a piece of code involving CONJ. This theorem-yielding function is called the *validation*, and the design allows most of the code to lie outside the abstract data type thm, on which the soundness of the whole system depends. If the tactic or its validation is somehow wrong, the system's soundness will not be compromised. However, the user is likely to be disappointed, making the effort to prove the theorem but never getting the theorem itself. (Instead of an error will be signalled, or possibly some other theorem delivered.) Gordon's retrospective on these techniques is informative [26], as is an early exposition of mine [59].

So we see that to implement any formal calculus using the LCF architecture, every inference rule must be implemented in code twice. There could be dozens of rules, some of them complicated, and any error would be damaging. Modifying an LCF-style system to implement a new formalism required months of work.

The solution to these problems is to use the LCF idea in a different way. Instead of focusing on the theorems of the given formalism, we should focus on *contextual implications* of the form $\Gamma \Longrightarrow \phi$. In particular, such implications can be joined end-to-end, creating proof trees. This approach has a number of advantages:

- Contexts are managed in the same way for every formalism being supported. Meanwhile, formalisms that manage the context explicitly can still be handled.
- The incremental buildup of a proof tree can express both forward and backward proof.
- Subgoals can even contain shared "placeholder" variables, as may arise when proving something like $\exists x\,(\phi(x) \wedge \psi(x))$. That is to say, we can try to prove something of the form $\phi(-)$ and if we manage to prove $\phi(f(1))$, then the other subgoal will become $\psi(f(1))$.

But the main advantage is simply that inference rules can now be written as assertions. For example, the conjunction rule above becomes

$$\phi, \psi \Longrightarrow \phi \wedge \psi.$$

The rule to eliminate a disjunction (proof by case analysis) becomes

$$\phi \vee \psi, (\phi \Longrightarrow \theta), (\psi \Longrightarrow \theta) \Longrightarrow \theta.$$

The rule to introduce an existential quantifier becomes

$$\phi(a) \Longrightarrow \exists x\,\phi(x).$$

This architecture allows primitive inferences of a formalism to be entered more or less as they might appear in a textbook.

Isabelle is the world's first *generic* theorem prover: designed to support logical inference in general as opposed to any single formalism. Isabelle takes some ideas from higher-order logic to represent formal syntax and logical inference rules [61]. Notational features such as variable binding (the i in $\sum_{i \in I} t_i$) and principles such as mathematical induction could be expressed straightforwardly, and conformance to traditional presentations could be proved [60]. Such meta-logical systems are known as *logical frameworks* [33], the first of which was the AUTOMATH language created by N. G. de Bruijn [17].

Another fundamental Isabelle principle is to provide the basic components of automatic proof search at the lowest level. The resolution techniques of first-order logic had already shown promise. It seemed right to adopt some of those techniques to provide automation for other formalisms. The joining together of implications, which is Isabelle's core inference mechanism, is itself a form of resolution. Over the years, increasingly sophisticated forms of automatic proof search were added to Isabelle [62].

Have these ideas stood the test of time? Certainly Isabelle is the only widely-used interactive theorem prover to be based on a logical framework. Meanwhile, efforts to exploit the power of resolution theorem proving finally resulted in strong integration of the two technologies: the *sledgehammer* tool [66] applies a battery of resolution theorem provers to a problem. It often finds a proof with the help of obscure facts from Isabelle's extensive libraries. Some of the most impressive work done using Isabelle—from the verification of security protocols [63] to a verified operating system kernel [46] to formalised mathematical theorems [2,65]—was only possible thanks to Isabelle's automation.

On the other hand, the idea of generic theorem proving is less successful. It's true that Isabelle has been used to support a great many formalisms [61], including a version of Martin-Löf's intuitionistic type theory, first-order logic, set theory and some modal logics. Isabelle/ZF develops a substantial amount of Zermelo-Fraenkel set theory [64]. One can argue that, even in the case of a single formalism, the use of a logical framework makes for a more flexible notion of proof construction than that found in traditional LCF-like systems. However, Isabelle's facility to support multiple formalisms is hardly used. Its most popular instantiation by far is Isabelle/HOL, which at bottom is yet another implementation of Church's higher-order logic [14]. Constructive type theories tend to be demanding syntactically; that research community has always preferred to build its own tools.

Isabelle's core ideas were in place by 1989 [60]. Since then, Isabelle development has increasingly moved to the Technical University of Munich, under the leadership of Prof. Tobias Nipkow. Of the innumerable contributions made there, the most important one is the Isar structured proof language [76], which although completely formal includes many mathematical idioms and often allows legible, natural-looking proofs. (Hitherto, Isabelle followed LCF in expecting users to generate proofs directly through the ML programming language.) Extensions to Isabelle's core ideas to support higher-order logic are also due to Nipkow and his colleagues, in particular, the notion of type classes [75]. The years have seen a long series of refinements made and case studies undertaken. One notable example is counterexample detection [7], whereby Isabelle can frequently inform the user that the statement under consideration is not a theorem.

5. Formalising Mathematics

The idea of performing mathematics by machine goes back at least as far as Leibniz, who invented a calculating machine. The first mathematician to properly realise this dream was de Bruijn [17], whose AUTOMATH project started in the 1960s and culminated in the formalisation of a monograph on the construction of the reals [45]. De Bruijn's approach was based on dependent type theories of his own devising. At around the same time, Andrzej Trybulec had the idea of formalising mathematics using a form of typed set theory; a substantial amount of mathematics was translated into his Mizar formalism, with a particularly elegant treatment of algebraic structures [70].

However, the main impetus for formalised mathematics has arisen from the requirements of verification, which has been a priority area for research funding. Harrison's early work [35] to verify a floating-point algorithm for the exponential function naturally led to the formalisation of increasing amounts of analysis. In order to verify probabilistic algorithms (which are controlled by virtual coin tosses), Hurd [42] formalised some probability theory, including measure theory and Lebesgue integration. Recent work in this direction includes Hölzl's comprehensive formalisation [41] of Markov chains, an abstract model of probabilistic systems that has numerous applications. Such work has generally been done using versions of HOL and Isabelle/HOL, but floating-point algorithms have also been verified using Coq [8].

Over time, competition arose between the proponents of different verification systems on the basis of how many well-known theorems had been formalised in each. Harrison took the lead here, formalising numerous results in his HOL Light system [34]. These included much of complex analysis [36] as well as a proof of the prime number theorem [37] and many other results. Later, Harrison played a major role in the effort to formally verify Hales's 1998 proof [29] of the

Kepler Conjecture, a 400-year-old problem concerning the optimal packing of spheres. Referees had objected to this proof because of its reliance on extensive computer calculations. In response, Hales launched the Flyspeck project [31] to formally verify his proof. Flyspeck was ultimately successful, confirming and simplifying Hales's argument [32]. Some of the formal proofs were done using Isabelle [58].

This was not the first time a mathematical result was formalised in order to confirm a contested proof. In 2005, Gonthier [23] formalised the Four Colour Theorem within Coq. The 1976 proof by Appel and Haken was notorious for its the use of a bespoke computer program to check nearly 2000 cases. Gonthier used a similar strategy, with a crucial difference: the cases were checked by algorithms that had been verified using Coq.

Other work of this sort concerned proofs that, while enjoying the full confidence of mathematicians, had other issues. My own formalisation of Gödel's second incompleteness theorem [65] falls into this category: most published proofs are sorely lacking in detail. Fleuriot's formalisation [19] of some material from Newton's *Principia* is remarkable for embracing Newton's use of infinitesimals: Fleuriot formally justifies the original proofs through *non-standard analysis*, which gives a rigorous foundation to infinitesimal reasoning. Above all, we have Gonthier's monumental project to verify the odd order theorem [24]; the issue here is simply size, as the theorem was originally proved in a 255-page journal article.

Generally speaking, the point of formalised mathematics is to clear up doubts, resolve ambiguities and identify errors. Mathematicians regularly make mistakes, as examined by Lakatos [49] in his history of Euler's polyhedron formula; he points out that even definitions can be wrong. Famous modern mathematicians can also be fallible:

> When the Germans were planning to publish Hilbert's collected papers and to present him with a set on the occasion of one of his later birthdays, they realised that they could not publish the papers in their original versions because they were full of errors, some of them quite serious. Thereupon they hired a young unemployed mathematician, Olga Taussky-Todd, to go over Hilbert's papers and correct all mistakes. Olga laboured for three years. [69, p. 201].

Increasingly, the impetus for formalising mathematics comes from mathematicians themselves. Fields medallist Vladimir Voevodsky wrote of his horror at finding errors in his own work:

> And who would ensure that I did not forget something and did not make a mistake, if even the mistakes in much more simple arguments take years to uncover? [73]

Voevodsky's *homotopy type theory* [72] is a new approach to the formalisation of mathematics. It gives Martin-Löf's intuitionistic type theory a new interpretation involving topological spaces, where types denote homotopy types.[5] Moreover, this interpretation suggests the new *univalence axiom*, which says that isomorphic constructions can be identified [3]. These ideas have generated widespread excitement, and experiments are under way using the Coq proof assistant.

In the opposite direction, axiomatic set theory has also been used to formalise mathematics. Sets and types both denote collections, with one great difference: typically a variable can have only one type but can belong to any number of sets. So set theory may offer the greatest flexibility in formalising mathematics. Work of my own concerns proving a number of properties to be equivalent to the axiom of choice [67] as well as formalising Gödel's proof of the relative consistency of the axiom of choice [64]. This work can be criticised as being only about set theory itself. However, recent work by Zhan shows that set theory can also be used to formalise traditional mathematical results from group theory, point-set topology and real analysis [79].

---

[5]A *homotopy* between two continuous functions f and g is a continuous "deformation" of f into g. With this concept one can define an equivalence relation on topological spaces, and the resulting equivalence classes are called *homotopy types*.

6. Obstacles to Formalising Mathematics

The obstacles to the formalisation of mathematics are perhaps not what people imagine. Gödel's theorem is not especially relevant to mathematics in practice. Even the *Entscheidungsproblem* is not that relevant: when we do have decision procedures, they can be too expensive to actually run. Anyway, our ambition is to *assist* mathematicians, not to replace them by computers. The obstacles we face take a variety of forms.

Foundational. A mathematician's imagination is unconstrained, but every formal calculus is a rigid framework and ultimately a prison. For example, axiomatic set theory begins by creating a world big enough for 19th-century mathematics, then escalates it through powerful principles to create an inconceivably large universe of mathematical objects. It contains such oddities as the limit of $\{\aleph_0, \aleph_{\aleph_0}, \aleph_{\aleph_{\aleph_0}}, \ldots\}$, a cardinal λ satisfying the equation $\lambda = \aleph_\lambda$. (The $\aleph$-sequence enumerates the infinite cardinalities, starting with $\aleph_0$: the cardinality of the integers.) Boolos [13] frankly states that he can't believe that such an object exists. And yet, category theory [52] begins by creating the category of sets and functions, assuming at a stroke the existence of a mathematical object incorporating the entire universe of sets, and building upon it. So one moment our universe seems far too large, and a moment later, it has become too small.

Conceptual. Intuition plays a huge role in mathematics. A mathematician easily sees that a certain function — perhaps defined by a complicated integral — is continuous. Many convergence properties are obvious. But in some cases, the formal proofs are enormous [37]. There are a variety of responses to this situation. One is to provide additional automation to deal with specific tasks, such as proving asymptotic properties [18]. Another is to accept that some apparently obvious facts, such as the Jordan curve theorem,[6] really are extremely difficult to prove rigorously [30]. Another is to accept that many apparently obvious statements are actually false [49]. But for all that, the use of any logical formalism requires immense patience.

Notational. The language of mathematics has evolved over centuries. It can be inconsistent and ambiguous — compare $\sin^2 x$ with $y^2 x$ — but it is familiar and reasonably effective. Any formalism implemented on a computer will be code, and look like it. For example, the fundamental theorem of algebra states that if n is a positive integer and $a_0, a_1, \ldots, a_n$ are complex numbers with $a_n \neq 0$, then $a_n z^n + \cdots + a_1 z + a_0 = 0$ for some complex z. The HOL Light version relaxes the precondition to $a_0 = 0$ or otherwise that the a_k are not all zero for $k = 1, \ldots, n$:

```
!a n. a(0) = Cx(&0) \/ ~(!k. k IN 1..n ==> a(k) = Cx(&0))
        ==> ?z. vsum(0..n) (\i. a(i) * z pow i) = Cx(&0)
```

The elegance of the mathematical notation is completely lost. Isabelle does a bit better, allowing the Greek alphabet and many mathematical symbols, but all such languages are still code.

One further complication deserves a mention. Many expressions of mathematics can be *undefined*, e.g. some limits, derivatives, integrals or simply x/y when $y = 0$. A formal calculus can deal with this problem in a number of different ways [2, §5.1]. So we may find that $x/y = x/y$ (when $x/0$ denotes some arbitrary value) or that $x/y = x/y$ if and only if $y \neq 0$ (when we regard primitive assertions about undefined values as necessarily false) or we may find that the very act of writing x/y is forbidden unless we can prove $y \neq 0$. This last approach can be inconvenient [9, §4], indeed each approach has its good and bad points. The idea that x/y is always something horrifies some mathematicians (especially if we go further, as in Isabelle, and actually stipulate that $x/0 = 0$). Elaborate treatments of definedness yield greater expressive power: paid for, as usual, by more complicated notations and harder proofs.

7. The Way Forward

In the course of this paper we have narrowed our focus several times, deliberately seeking out research as opposed to applications. This is necessary because computational logic tools,

---

[6]which states that every simple closed curve partitions the plane into an interior and an exterior region

interpreted broadly, are now widely used in the development of hardware, software and in the provision of online services. We are in danger, perhaps, of overlooking the main motivation for computational logic: proving the correctness of computer systems. So let's take a moment to consider a basic problem in computer science: compiler correctness.

A compiler is a piece of software that translates program code (written in a programming language such as C or Java) into some sort of binary form that can be executed on computer hardware. Compilers are complex, not least because they use a wide variety of sophisticated techniques to generate binary code that will run as fast as possible. If the compiler is somehow faulty then it could introduce errors into the code that is actually executed. People have sought to prove compilers correct using computational logic techniques [50]. But even if we finally obtain a correct compiler, how do we translate it into executable binary code? A further complication is that the semantics of a language like C has little in common with the way C is used in real software, which relies on innumerable undefined behaviours.

Kumar et al. [47] have solved this chicken-and-egg problem. They have designed their own programming language, CakeML, with a clean semantics. They express a simple compiling algorithm as a mathematical function in HOL4 (a descendant of Gordon's original HOL system). They translate the same algorithm to CakeML and compile it to binary (for the popular x86 architecture) *using HOL4 itself*, giving high assurance of correctness. This yields a trustworthy version of the compiler in binary form. Their approach is an instance of the well-known technique of *bootstrapping*, where a compiler is built for a simple but adequate language and then used as a tool to further its own development until the full language is supported. In this case, the bootstrapping process includes verification, and is possible because HOL4 can cope with the full binary code of the CakeML compiler. This is one representative of a large body of research concerned with treating mathematical functions defined within a theorem prover as executable code.

Readers who want to pursue these topics further will find surveys available on formally verified mathematics [1] and on the history of interactive theorem proving [20,38]. Isabelle can be downloaded via the URL https://isabelle.in.tum.de.

References

1. J. Avigad and J. Harrison.
   Formally verified mathematics.
   Commun. ACM, 57(4):66–75, Apr. 2014.
2. J. Avigad, J. Hölzl, and L. Serafin.
   A formally verified proof of the central limit theorem.
   Journal of Automated Reasoning, 59(4):389–423, Dec. 2017.
3. S. Awodey, Á. Pelayo, and M. A. Warren.
   Voevodsky's univalence axiom in homotopy type theory.
   Notices of the AMS, 60(9):1164–1167, Oct. 2013.
4. M. Ayala-Rincón and C. A. Muñoz, editors.

Interactive Theorem Proving —– 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings. Springer International Publishing, 2017.

5. P. Benacerraf and H. Putnam, editors.
   Philosophy of Mathematics: Selected Readings.
   Cambridge University Press, 2nd edition, 1983.

6. Y. Bertot and P. Castéran.
   Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.
   Springer, 2004.

7. J. C. Blanchette and T. Nipkow.
   Nitpick: A counterexample generator for higher-order logic based on a relational model finder.
   In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

8. S. Boldo, M. Joldes, J. Muller, and V. Popescu.
   Formal verification of a floating-point expansion renormalization algorithm.
   In Ayala-Rincón and Muñoz [4], pages 98–113.

9. S. Boldo, C. Lelay, and G. Melquiond.
   Coquelicot: A user-friendly library of real analysis for Coq.
   Mathematics in Computer Science, 9(1):41–62, 2015.

10. G. S. Boolos.
    Gottlob Frege and the foundations of arithmetic.
    In *Logic, Logic, and Logic* [12], pages 143–154.

11. G. S. Boolos.
    The iterative conception of set.
    In *Logic, Logic, and Logic* [12], pages 13–29.

12. G. S. Boolos.
    Logic, Logic, and Logic.
    Harvard University Press, 1998.

13. G. S. Boolos.
    Must we believe in set theory?
    In *Logic, Logic, and Logic* [12], pages 120–132.

14. A. Church.
    A formulation of the simple theory of types.
    Journal of Symbolic Logic, 5:56–68, 1940.

15. T. Coquand and G. Huet.
    The calculus of constructions.
    Information and Computation, 76:95–120, 1988.

16. M. Davis and H. Putnam.
    A computing procedure for quantification theory.
    J. ACM, 7(3):207–215, July 1960.

17. N. G. de Bruijn.
    A survey of the project AUTOMATH.
    In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

18. M. Eberl.
    Semi-automatic asymptotics in Isabelle/HOL.
    Online at https://www.newton.ac.uk/seminar/20170705133014302, July 2017.
    Presented at the Isaac Newton Institute for Mathematical Sciences.

19. J. D. Fleuriot.
    A Combination of Geometry Theorem Proving and Nonstandard Analysis, with Application to Newton's Principia.
    Springer, 2001.

20. H. Geuvers.
    Proof assistants: History, ideas and future.
    Sadhana, 34(1):3–25, Feb 2009.

21. K. Gödel.
    Russell's mathematical logic.

In Benacerraf and Putnam [5], pages 447–469.
First published in 1944.

22. K. Gödel.
Some basic theorems on the foundations of mathematics and their implications.
In S. Feferman, editor, *Kurt Gödel: Collected Works*, volume III, pages 304–323. Oxford University Press, 1995.

23. G. Gonthier.
The four colour theorem: Engineering of a formal proof.
In D. Kapur, editor, *Computer Mathematics*, LNCS 5081, pages 333–333. Springer Berlin Heidelberg, 2008.

24. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry.
A machine-checked proof of the odd order theorem.
In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, LNCS 7998, pages 163–179. Springer, 2013.

25. M. J. C. Gordon.
Why higher-order logic is a good formalism for specifying and verifying hardware.
In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.

26. M. J. C. Gordon.
Tactics for mechanized reasoning: A commentary on Milner (1984) 'the use of machines to assist in rigorous proof'.
Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 373(2039), 2015.

27. M. J. C. Gordon and T. F. Melham.
Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.
Cambridge University Press, 1993.

28. M. J. C. Gordon, R. Milner, and C. P. Wadsworth.
Edinburgh LCF: A Mechanised Logic of Computation.
LNCS 78. Springer, 1979.

29. T. C. Hales.
An overview of the Kepler conjecture.
ArXiv Mathematics e-prints, Nov. 1998.

30. T. C. Hales.
The Jordan curve theorem, formally and informally.
The American Mathematical Monthly, 114(10):882–894, Dec. 2007.

31. T. C. Hales et al.
A formal proof of the Kepler conjecture.
arXiv.org, abs/1501.02155, Jan. 2015.

32. T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller.
A revision of the proof of the Kepler conjecture.
Discrete & Computational Geometry, 44(1):1–34, 2010.

33. R. Harper, F. Honsell, and G. Plotkin.
A framework for defining logics.
J. ACM, 40(1):143–184, 1993.

34. J. Harrison.
HOL Light: A tutorial introduction.
In M. K. Srivas and A. J. Camilleri, editors, *Formal Methods in Computer-Aided Design: FMCAD '96*, LNCS 1166, pages 265–269. Springer, 1996.

35. J. Harrison.
Floating point verification in HOL Light: the exponential function.
Formal Methods in System Design, 16:271–305, 2000.

36. J. Harrison.
Formalizing basic complex analysis.
In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007.

37. J. Harrison.

Formalizing an analytic proof of the prime number theorem.
Journal of Automated Reasoning, 43(3):243–261, 2009.

38. J. Harrison, J. Urban, and F. Wiedijk.
History of interactive theorem proving.
In J. Siekmann, editor, *Handbook of the History of Logic (Computational Logic)*, volume 9, pages 135–214. Elsevier, 2014.

39. M. J. H. Heule and O. Kullmann.
The science of brute force.
Commun. ACM, 60(8):70–79, July 2017.

40. A. Heyting.
The intuitionist foundations of mathematics.
In Benacerraf and Putnam [5], pages 52–61.
First published in 1944.

41. J. Hölzl.
Markov chains and Markov decision processes in Isabelle/HOL.
Journal of Automated Reasoning, 59(3):345–387, Oct. 2017.

42. J. Hurd.
Verification of the Miller-Rabin probabilistic primality test.
Journal of Logic and Algebraic Programming, 56:3–21, 2002.

43. M. Huth and M. Ryan.
Logic in Computer Science: Modelling and Reasoning about Systems.
Cambridge University Press, 2nd edition, 2004.

44. R. Jhala and R. Majumdar.
Software model checking.
ACM Comput. Surv., 41(4):21:1–21:54, Oct. 2009.

45. L. Jutting.
Checking Landau's "Grundlagen" in the AUTOMATH System.
PhD thesis, Eindhoven University of Technology, 1977.

46. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood.
sel4: Formal verification of an operating-system kernel.
Commun. ACM, 53(6):107–115, June 2010.

47. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens.
CakeML: A verified implementation of ML.
In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191. ACM, 2014.

48. K. Kunen.
Set Theory: An Introduction to Independence Proofs.
North-Holland, 1980.

49. I. Lakatos.
Proofs and Refutations: The Logic of Mathematical Discovery.
Cambridge University Press, 1976.

50. X. Leroy.
Formal certification of a compiler back-end or: programming a compiler with a proof assistant.
In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

51. G. Lowe.
Some new attacks upon security protocols.
In *9th Computer Security Foundations Workshop*, pages 162–169. IEEE Computer Society Press, 1996.

52. S. Mac Lane.
Categories for the Working Mathematician.
Springer, 1971.

53. P. Martin-Löf.
An intuitionistic theory of types: Predicative part.
In H. Rose and J. Shepherdson, editors, *Logic Colloquium '73*, Studies in Logic and the Foundations of Mathematics 80, pages 73–118. North-Holland, 1975.

54. P. Martin-Lof and Z. A. Lozinski.

Constructive mathematics and computer programming.
Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences, 312(1522):501–518, 1984.

55. A. R. D. Mathias.
A term of length 4,523,659,424,929.
Synthese, 133(1):75–86, Oct 2002.

56. W. McCune.
Solution of the Robbins problem.
Journal of Automated Reasoning, 19(3):263–276, 1997.

57. T. R. Nicely.
Pentium FDIV flaw, 2011.
FAQ page online at http://www.trnicely.net/pentbug/pentbug.html.

58. T. Nipkow, G. Bauer, and P. Schultz.
Flyspeck I: tame graphs.
In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, LNAI 4130, pages 21–35. Springer, 2006.

59. L. C. Paulson.
Natural deduction as higher-order resolution.
Journal of Logic Programming, 3:237–258, 1986.

60. L. C. Paulson.
The foundation of a generic theorem prover.
Journal of Automated Reasoning, 5(3):363–397, 1989.

61. L. C. Paulson.
Isabelle: The next 700 theorem provers.
In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

62. L. C. Paulson.
A generic tableau prover and its integration with Isabelle.
Journal of Universal Computer Science, 5(3):73–87, 1999.

63. L. C. Paulson.
Inductive analysis of the Internet protocol TLS.
ACM Transactions on Information and System Security, 2(3):332–351, Aug. 1999.

64. L. C. Paulson.
The relative consistency of the axiom of choice — mechanized using Isabelle/ZF.
LMS Journal of Computation and Mathematics, 6:198–248, 2003.
http://www.lms.ac.uk/jcm/6/lms2003-001/.

65. L. C. Paulson.
A machine-assisted proof of Gödel's incompleteness theorems for the theory of hereditarily finite sets.
Review of Symbolic Logic, 7(3):484–498, Sept. 2014.

66. L. C. Paulson and J. C. Blanchette.
Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers.
In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

67. L. C. Paulson and K. Grabczewski.
Mechanizing set theory: Cardinal arithmetic and the axiom of choice.
Journal of Automated Reasoning, 17(3):291–323, Dec. 1996.

68. J. A. Robinson.
A machine-oriented logic based on the resolution principle.
J. ACM, 12:23–41, 1965.

69. G.-C. Rota.
Indiscrete Thoughts.
Springer, 2009.

70. P. Rudnicki, C. Schwarzweller, and A. Trybulec.
Commutative algebra in the Mizar system.
Journal of Symbolic Computation, 32(1):143–169, 2001.

71. A. M. Turing.
    On computable numbers, with an application to the Entscheidungsproblem.
    Proceedings of the London Mathematical Society, 2(42):230–265, 1936.
72. T. Univalent Foundations Program.
    Homotopy Type Theory: Univalent Foundations of Mathematics.
    https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
73. V. Voevodsky.
    The origins and motivations of univalent foundations.
    The Institute Letter, pages 8–9, Summer 2014.
    Online at https://www.ias.edu/ideas/2014/voevodsky-origins.
74. P. Wadler.
    Propositions as types.
    Commun. ACM, 58(12):75–84, Nov. 2015.
75. M. Wenzel.
    Type classes and overloading in higher-order logic.
    In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, LNCS
    1275, pages 307–322. Springer, 1997.
76. M. Wenzel.
    Isabelle/Isar — a generic framework for human-readable proof documents.
    Studies in Logic, Grammar, and Rhetoric, 10(23):277–297, 2007.
    From Insight to Proof — Festschrift in Honour of Andrzej Trybulec.
77. A. N. Whitehead and B. Russell.
    Principia Mathematica.
    Cambridge University Press, 1962.
    Paperback edition to *56, abridged from the 2nd edition (1927).
78. V. Woollaston.
    WannaCry ransomware: What is it and how to protect yourself, May 2017.
    Online at http://www.wired.co.uk/article/wannacry-ransomware-virus-patch.
79. B. Zhan.
    Formalization of the Fundamental Group in Untyped Set Theory Using Auto2, pages 514–530.
    In Ayala-Rincón and Muñoz [4], 2017.