Proofs of Correctness in Mathematics and Industry

Henk Barendregt
Radboud University
Nijmegen, The Netherlands

July 11, 2007

Abstract

Quality of a product needs verification. If the product is complex, this verification cannot be done "by hand", but one needs tools. How then are those tools being checked? Eventually quality comes from a careful specification and design methodology. Warranty is based on a mathematical proof that the design meets its specification. As mathematical proofs become long and complex themselves, we also need a tool to verify proofs. In order to prevent an infinite regress, this last tool must have a basic simplicity. Indeed, mathematical assistants that help users to develop and verify proofs are build on the current foundational logical systems that can be described in a couple of pages.

It is expected that within a couple of decades, the use of reliable mathematical assistants will be widespread and will help the human user to learn, develop, teach, communicate, referee, and apply mathematics. Computer-verified correctness will probably become one of the most important applications of mathematics and computer science.

Keywords: **quality, design, specification, mathematical assistant, proof-object, computer-verification**

1. The quality problem

Buying a product from a craftsman requires some care. For example, in the Stone Age an arrow, used for hunting and hence for survival, needed to be inspected for its sharpness and proper fixation of the stone head to the wood. Complex products of more modern times cannot be checked in such a simple way and the idea of warranty was born: a nonsatisfactory product will be repaired or replaced, or else you get your money back. This puts the responsibility for quality on the shoulders of the manufacturer, who has to test the product before selling. In contemporary IT products, however, testing for proper functioning in general becomes impossible. If we have an array of 17×17 switches in a device, the number of possible positions is $2^{17^2} = 2^{289} \sim 10^{87}$, more than the estimated number of elementary particles in the universe. Modern chips have billions of switches on them, hence a state space of a size that is truly dwarfing astronomical numbers. Therefore, in most cases, simple-minded testing is out

of the question because the required time would surpass by far the lifetime expectancy of the universe. As these chips are used in strategic applications, like airplanes, medical equipment, and banking systems, there is a problem with how to warrant correct functioning.

Therefore, the need for special attention to the quality of complex products is obvious, both from a user's point of view and that of a producer. This concern is not just academic. In 1994 the computational number theorist T. R. Nicely discovered by chance a bug[1] in a widely distributed Pentium chip. After an initial denial, the manufacturer eventually had to publicly announce a recall, replacement and destruction of the flawed chip with a budgeted cost of US $475 Million.

Fortunately, mathematics has found a way to handle within a finite amount of time a supra-astronomical number of cases, in fact an infinity of them. The notion of proof provides a way to handle all possible cases with certainty. The notion of mathematical induction is one proof method that can deal with an infinity of cases: If a property P is valid for the first natural number 0 (or if you prefer 1) and if validity of P for n implies that for $n+1$, then P is valid for all natural numbers. For example, for all n one has

$$\sum_{k=0}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1). \qquad P(n)$$

This can be proved by showing it for $n = 0$; and then showing that if $P(n)$ holds, then also $P(n+1)$. Indeed $P(0)$ holds: $\sum_{k=0}^{0} k^2 = 0$. If $P(n)$ holds, then

$$\begin{aligned} \sum_{k=0}^{n+1} k^2 &= \Big(\sum_{k=0}^{n} k^2\Big) + (n+1)^2 \\ &= \frac{1}{6} n(n+1)(2n+1) + (n+1)^2 \\ &= \frac{1}{6}(n+1)(n+2)(2n+3), \end{aligned}$$

hence $P(n+1)$. Therefore $P(n)$ holds for all natural numbers n.

Another method to prove statements valid for an infinite number of instances is to use symbolic rewriting: From the usual properties of addition and multiplication over the natural numbers (proved by induction), one can derive equationally that $(x+1)(x-1) = x^2 - 1$, for all instances of x.

Proofs have been for more than two millennia the essence of mathematics. For more than two decades, proofs have become essential for warranting quality of complex IT products. Moreover, by the end of the twentieth century, proofs in mathematics have become highly complex. Three results deserve mention: the Four Color Theorem, the Classification of the Finite Simple Groups and the correctness of the Kepler Conjecture (about optimal packing of equal three-dimensional spheres). Part of the complexity of these proofs is that they rely

[1] It took Dr. Nicely several months to realize that the inconsistency he noted in some of his output was not due to his algorithms, but caused by the (microcode on the) chip. See [Ede97] for a description of the mathematics behind the error.

on large computations by a computer (involving up to a billion cases). A new technology for showing correctness has emerged: automated verification of large proofs.

Two methodological problems arise. (1) How do proofs in mathematics relate to the physical world of processors and other products? (2) How can we be sure that complex proofs are correct? The first question will be addressed in the next section, and the second in the following section. Finally the technology is predicted to have a major impact on the way mathematics will be done in the future.

2. Specification, design, and proofs of correctness

The Rationality Square

The ideas in this section come from [Wup98] and make explicit what is known intuitively by designers of systems that use proofs. The first thing to realize is that if we want quality of a product, then we need to specify what we want as its behavior. Both the product and its (desired) behavior are in “reality”, whereas the specification is written in some precise language. Then we make a design with the intention to realize it as the intended product. Also the design is a formal (mathematical) object. If one can prove that the designed object satisfies the formal specification, then it is expected that the realization has the desired behavior, see Fig. 1. For this it is necessary that the informal (desired) behavior and the specification are close to each other and can be inspected in a clearly understandable way. The same holds for the design and realization. Then the role of proofs is in its place: They do not apply to an object and desired behavior in reality but to a mathematical descriptions of these.

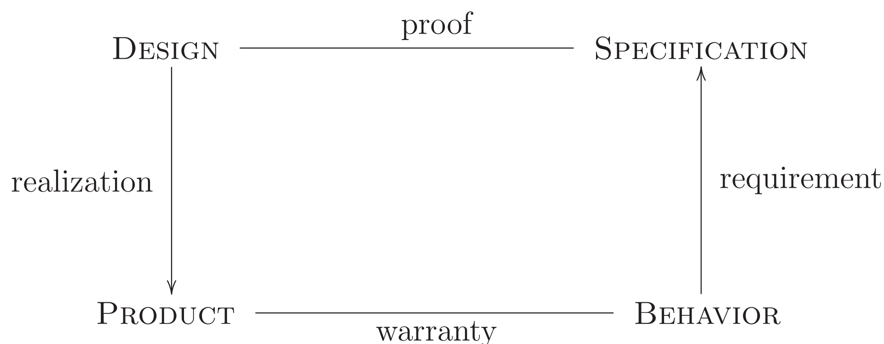

Figure 1: Wupper’s rationality square

In this setup, the specification language should be close enough to the informal specification of the desired behavior. Similarly, the technology of realization should also be reliable. The latter again may depend on tools that are constructed component wise and realize some design (e.g., silicon compilers that take as input the design of a chip and have as output the instructions to realize them). Hence, the rationality square may have to be used in an earlier phase.

This raises, however, two questions. Proofs should be based on some axioms. Which ones? Moreover, how do we know that provability of a formal (mathematical) property implies that we get what we want? The answers to these questions come together. The proofs are based on some axioms that hold for the objects of which the product is composed. Based on the empirical facts that the axioms hold, the quality of the realized product will follow.

Products as Chinese Boxes of Components

Now we need to enter some of the details of how the languages for the design and specification of the products should look. The intuitive idea is that a complex product consists of components $b_1, \ldots, b_k$ put together in a specific way yielding $F^{(k)}(b_1, \ldots, b_k)$. The superscript '$(k)$' indicates the number of arguments that F needs. The components are constructed in a similar way, until one hits the basic components $O_0, O_1, \ldots$ that no longer are composed. Think of a playing music installation B. It consists of a CD, CD-player, amplifier, boxes, wires and an electric outlet, all put together in the right way. So

$$B = F^{(6)}(\text{CD,CD-player,amplifyer,boxes,wires,outlet}),$$

where $F^{(6)}$ is the action that makes the right connections. Similarly the amplifier and other components can be described as a composition of their parts. A convenient way to depict this idea in general is the so-called *Chinese box*, see Fig. 2. This is a box with a lid. After opening the lid one finds a (finite) set of "neatly arranged" boxes that either are open and contain a basic object or are again other Chinese boxes (with lid). Eventually one will find something in a decreasing chain of boxes. This corresponds to the component-wise construction of anything, in particular of hardware, but also of software[2]. It is easy to construct a grammar for expressions denoting these Chinese boxes. The basic objects are denoted by $o_0, o_1, \ldots$. Then there are "constructors", that turn expressions into new expressions. Each constructor has an '"rity", that indicates how many arguments it has. There may be unary,

[2]In order that the sketched design method works well for software, it is preferable to have *declarative* software, i.e., in the *functional* or *logic* programming style, in particular without side effects.

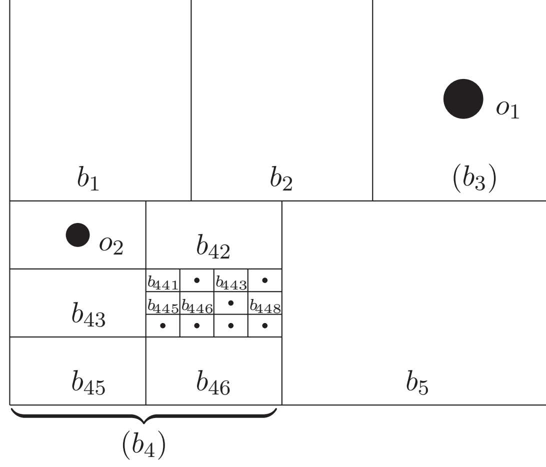

Figure 2: Partially opened Chinese box

binary, ternary, and so on constructors. Such constructors are denoted by

$$f_0^{(k)}, f_1^{(k)}, \ldots,$$

where k denotes the arity of the constructor. If $b_1, \ldots, b_k$ are expressions and $f_i^{(k)}$ is a constructor of arity k, then

$$f_i^{(k)}(b_1, \ldots, b_k)$$

is an expression. A precise grammar for such expressions is as follows.

2.1. Definition. (i) Consider the following alphabet

$$\Sigma = \{o_i \mid i \in \mathbb{N}\} \cup \{f_i^{(k)} \mid i, k \in \mathbb{N}\} \cup \{,, (,)\}.$$

(ii) Expressions $\mathcal{E}$ form the smallest set of words over Σ satisfying

$$\begin{array}{rcl} & & o_i \in \mathcal{E}; \\ b_1, \ldots, b_k \in \mathcal{E} & \Rightarrow & f_i^{(k)}(b_1, \ldots, b_k) \in \mathcal{E}. \end{array}$$

An example of a fully specified expression is

$$f_1^{(2)}(o_0, f_3^{(1)}(o_1, o_2, o_0)).$$

The partially opened Chinese box in Fig. 2 can be denoted by

$$f_1^{(5)}(b_1, b_2, o_1, b_4, b_5),$$

where

$$\begin{array}{rcl} b_4 & = & f_2^{(6)}(o_2, b_{4,2}, b_{4,3}, b_{4,4}, b_{4,5}, b_{4,6}), \\ b_{4,4} & = & f_4^{(12)}(b_{4,4,1}, o_3, b_{4,4,3}, o_4, b_{4,3,5}, b_{4,4,6}, o_5, \\ & & b_{4,4,8}, o_6, o_7, o_8, o_9), \end{array}$$

and the other b_k still have to be specified.

2.2. Definition. A *design* is an expression $b \in \mathcal{E}$.

Specification and Correctness of Design

Following the rationality square one now can explain the role of mathematical proofs in industrial design.

Some mathematical language is needed to state in a precise way the requirements of the products. We suppose that we have such a specification language $\mathcal{L}$, in which the expressions in $\mathcal{E}$ are terms. We will not enter into the details of such a language, but we will mention that for IT products it often is convenient to be able to express relationships between the states before and after the execution of a command or to express temporal relationships. Temporal statements include "eventually the machine halts" or "there will be always a later moment in which the system receives input". See [MP92], [AO97], [HJ98] and [BPS01] for possible specification languages, notably for reactive systems, and [Mor01] for a general introduction to the syntax and semantics of logical languages used in computer science.

2.3. Definition. A *specification* is a unary formula[3] $S(\cdot)$ in $\mathcal{L}$.

Suppose we have the specification S and a candidate design b as given. The task is to prove in a mathematical way $S(b)$, i.e., that S holds of b. We did not yet discuss any axioms, or a way to warrant that the proved property is relevant. For this we need the following.

2.4. Definition. A *valid interpretation* for $\mathcal{L}$ consists of the following.

(i) For basic component expressions o there is an interpretation O in the "reality" of products.

(ii) For constructors $f^{(k)}$, there is a way to put together k products $p_1, \ldots, p_k$ to form $F^{(k)}(p_1, \ldots, p_k)$.

(iii) By (i) and (ii) all designs have a realization. For example the design $f_1^{(2)}(o_0, f_3^{(1)}(o_1, o_2, o_0))$ is interpreted as $F_1^{(2)}(O_0, F_3^{(1)}(O_1, O_2, O_0))$.

(iv) There are axioms of the form

$$P(c)$$
$$\forall x_1 \ldots x_k\,[Q(x_1, \ldots, x_k) \;\Rightarrow\; R(f^{(k)}(x_1, \ldots, x_k))].$$

Here P, Q, and R are formulas (formal statements) about designs: P and R about one design and Q about k designs.

(v) The formulas of $\mathcal{L}$ have a physical interpretation.

(vi) By the laws of physics, it is known that the interpretation given by (v) of the axioms holds for the interpretation described in the basic components and constructors. The soundness of logic then implies that statements proved from the axioms will also hold after interpretation.

This all may sound a bit complex, but the idea is simple and can be found in any book on predicate logic and its semantics, see [Mor01], [Hod97]. Proving starts from the axioms using logical steps; validity of the axioms and soundness of logic implies that the proved formulas are also valid.

[3] Better: A formula $S = S(x) = S(\cdot)$ with one free variable x in S.

The industrial task of constructing a product with a desired behavior can be fulfilled as follows.

2.5. Design method (I). (i) *Find a language $\mathcal{L}$ with a valid interpretation.*

(ii) *Formulate a specification S, such that the desired behavior becomes the interpretation of S.*

(iii) *Construct an expression b, intended to solve the task.*

(iv) *Prove $S(b)$ from the axioms of the interpretation mentioned in (i).*

(v) *The realization of b is the required product.*

Of course the last step of realizing designs may be nontrivial. For example transforming a chip design to an actual chip is an industry by itself. But that is not the concern now. Moreover, such a realization process can be performed by a tool that is the outcome of a similar specification-design-proof procedure.

The needed proofs have to be given from the axioms in the interpretation. Design method I builds up products from "scratch". In order not to reinvent the wheel all the time, one can base new products on previously designed ones.

2.6. Design method (II). *Suppose one wants to construct b satisfying S.*

(i) *Find subspecifications $S_1, \ldots, S_k$ and a constructor $f^{(k)}$ such that*

$$S_1(x_1) \;\&\; \ldots \;\&\; S_k(x_k) \;\Rightarrow\; S(f^{(k)}(x_1, \ldots, x_k)).$$

(ii) *Find (on-the-shelf) designs $b_1, \ldots, b_k$ such that for $1 \leq i \leq k$, one has*

$$S_i(b_i).$$

(iii) *Then the design $b = f^{(k)}(b_1, \ldots, b_k)$ solves the task.*

Again this is done in a context of a language $\mathcal{L}$ with a valid interpretation and the proofs are from the axioms in the interpretation.

After having explained proofs of correctness, the correctness of proofs becomes an issue. In an actual nontrivial industrial design, a software system controlling metro-trains in Paris without a driver, one needed to prove about 25,000 propositions in order to get reliability. These proofs were provided by a theorem prover. Derivation rules were added to enhance the proving power of the system. It turned out that if no care was taken, 2% to 5% of these added derivation rules were flawed and led to incorrect statements, see [Abr98]. Next section deals with the problem of getting proofs right.

3. Correctness of proofs

Methodology

Both in computer science and in mathematics proofs can become large. In computer science this is the case because the proofs that products satisfy certain specifications, as explained earlier, may depend on a large number of cases that need to be analyzed. In mathematics, large proofs occur as well, in this case because of the depth of the subject. The example of the Four Color Theorem

in which billions of cases need to be checked is well known. Then there is the proof of the classification theorem for simple finite groups needing thousands of pages (in the usual style of informal rigor).

That there are long proofs of short statements is not an accident, but a consequence of a famous undecidability result.

3.1. Theorem (Turing). *Provability in predicate logic is undecidable.*

Proof. See, for example, [Dav04]. ■

3.2. Corollary. *For predicate logic, there is a number n and a theorem of length n, with the smallest proof of length at least $n^{n!}$.*

Proof. Suppose that for every n theorems of length at least n, a proof of length $<n^{n!}$ exists. Then checking all possible proofs of such length provides a decision method for theoremhood, contradicting the undecidablity result. ■

Of course this does not imply that there are *interesting* theorems with essentially long proofs. The question now arises, how one can verify long proofs and large numbers of shorter ones? This question is both of importance for pure mathematics and for the industrial applications mentioned before.

The answer is that the state of the foundations of mathematics is such that proofs can be written in full detail, making it possible for a computer to check their correctness. Currently it still requires considerable effort to make such "formalizations" of proofs, but there is good hope that in the future this will become easier. Anyway, industrial design, as explained earlier, already has proved the viability and value of formal proofs. For example, the Itanium, a successor of the Pentium chip, has a provably correct arithmetical unit; see [GHH$^+$02].

Still one may wonder how one can assure the correctness of mathematical proofs via machine verification, if such proofs need to assure the correctness of machines. It seems that there is here a vicious circle of the type chicken-egg. The principal founder of machine verification of formalized proofs is the Dutch mathematician N. G. de Bruijn[4]; see [dB70]. He emphasized the following criterion for reliable automated proof-checkers: Their programs must be small, so small that a human can (easily) verify the code by hand. In the next subsection, we will explain why it is possible to satisfy this so-called de Bruijn criterion.

Foundations of Mathematics

The reason that fully formalized proofs are possible is that for all mathematical activities, there is a solid foundation that has been laid in a precise formal system. The reason that automated proof-checkers exist that satisfy the de Bruijn criterion is that these formal systems are simple enough, allowing a logician to write them down from memory in a couple of pages.

[4]McCarthy described machine proof-checking some years earlier, see [McC62], but did not come up with a formal system that had a sufficiently powerful and convenient implementation.

Mathematics is created by three mental activities: structuring, computing and reasoning. It is an art and craftsmanship "with a power, precision and certainty, that is unequalled elsewhere in life[5]". The three activities, respectively, provide definitions and structures, algorithms and computations, proofs and theorems. These activities are taken as a subject of study by themselves, yielding ontology (consisting either of set, type, or category theory), computability theory and logic.

Activity	Tools	Results	Meta study
Structuring	Axioms Definitions	Structures	Ontology
Computing	Algorithms	Answers	Computability[6]
Reasoning	Proofs	Theorems	Logic

Figure 3: Mathematical activity: tools, results, and meta study

During the history of mathematics these activities enjoyed attention in different degrees. Mathematics started with the structures of the numbers and planar geometry. Babylonian—Chinese—Egyptian mathematics was mainly occupied with computing. In ancient Greek mathematics, reasoning was introduced. These two activities came together in the work of Archimedes, al-Kwarizmi, and Newton. For a long time only occasional extensions of the number systems was all that was done as structuring activity. The art of defining more and more structures started in the nineteenth century with the introduction of groups by Galois and non-Euclidean spaces by Lobachevsky and Bolyai. Then mathematics flourished as never before.

Logic

The quest for finding a foundation for the three activities started with Aristotle. This search for "foundation" does not imply that one was uncertain how to prove theorems. Plato had already emphasized that any human being of normal intelligence had the capacity to reason that was required for mathematics. What Aristotle wanted was a survey and an understanding of that capacity. He started the quest for logic. At the same time Aristotle introduced the "synthetic way" of introducing new structures: the axiomatic method. Mathematics consists of concepts and of valid statements. Concepts can be defined from other concepts. Valid statements can be proved from other such statements. To prevent an infinite regress, one had to start somewhere. For concepts one starts with the primitive notions and for valid statements with the axioms. Not long after this description, Euclid described geometry using the axiomatic method in a way that was only improved by Hilbert, more than 2000 years later. Also Hilbert gave the right view on the axiomatic method: The axioms form an implicit definition of the primitive notions.

[5]From: *The man without qualities*, R. Musil, Rohwolt.

[6]Formerly called "Recursion Theory".

Frege completed the quest of Aristotle by giving a precise description of predicate logic. Gödel proved that his system was complete, i.e., sufficiently strong to derive all valid statements within a given axiomatic system. Brouwer and Heyting refined predicate logic into the so-called *intuitionistic* version. In their system, one can make a distinction between weak existence ("there exists a solution, but it is not clear how to find it") and constructive one ("there exists a solution and from the proof of this fact one can construct it"), see [vD04].

Ontology

An early contribution to ontology came from Descartes, who introduced what is now called Cartesian products (pairs or more generally tuples of entities), thereby relating geometrical structures to arithmetical (in the sense of algebraic) ones. When in the nineteenth century, there was a need for systematic ontology, Cantor introduced set theory in which sets are the fundamental building-blocks of mathematics. His system turned out to be inconsistent, but Zermelo and Fraenkel removed the inconsistency and improved the theory so that it could act as an ontological foundation for large parts of mathematics, see [Hal74].

Computability

As soon as the set of consequences of an axiom system had become a precise mathematical object, results about this collection started to appear. From the work of Gödel, it followed that the axioms of arithmetic are essentially incomplete (for any consistent extension of arithmetic, there is an independent statement A, that is neither provable nor refutable). An important part of the reasoning of Gödel was that the notion "p is a proof of A" is after coding a computable relation. Turing showed that predicate logic is undecidable (it cannot be predicted by machine whether a given statement can be derived or not). To prove undecidability results, the notion of computation needed to be formalized. To this end Church came with a system of lambda-calculus, see [Bar92], later leading to the notion of functional programming with languages such as Lisp, ML, and Haskell. Turing came with the notion of the Turing machine, later leading to imperative programming with languages such as Fortran and C, and showed that it gave the same notion of computability as Church's. If we assume the so-called Church−Turing thesis that humans and machines can compute the same class of mathematical functions, something that most logicians and computer scientists are willing to do, then it follows that provability in predicate logic is also undecidable by humans.

Mechanical Proof Verification

As soon as logic was fully described, one started to formalize mathematics. In this endeavor Frege was unfortunate enough to base mathematics on the inconsistent version of Cantorian set theory. Then Russell and Whitehead came with an alternative ontology, type theory, and started to formalize very elementary parts of mathematics. In type theory, that currently exists in various forms, functions are the basic elements of mathematics and the types form a way to

classify these. The formal development of mathematics, initiated by Russell and Whitehead, lay at the basis of the theoretical results of Gödel and Turing. On the other hand, for practical applications, the formal proofs become so elaborate, that it is almost undoable for a human to produce them, let alone to check that they are correct.

It was realized by J. McCarthy and independently by N.G. de Bruijn that this verification should not be done by humans but by machines. The formal systems describing logic, ontology, and computability have an amazingly small number of axioms and rules. This makes it possible to construct relatively small mathematical assistants. These computer systems help the mathematician to verify whether the definitions and proofs provided by the human are well founded and correct.

Based on an extended form of type theory de Bruijn introduced the system AUTOMATH, see [NGdV94], in which this idea was first realized, although somewhat painfulof thely, because of the level of detail in which the proofs needed to be presented. Nevertheless, proof-checking by mathematical assistants based on type theory is feasible and promising. For some modern versions of type theory and assistants based on these, see [ML84], [NGdV94], [Con97], [BG01] and [BC04].

Soon after the introduction of AUTOMATH, other mathematical assistants were developed, based on different foundational systems. There is the system MIZAR based on set theory; the system HOL(-light) based on higher order logic; and ACL_2 based on the computational model "primitive recursive arithmetic". See [BW05] for an introduction and references and [Bar06] for resulting differences of views in the philosophy of mathematics. To obtain a feel of the different styles of formalization, see [Wie06].

In [Gon05] an impressive full development of the Four Color Theorem is described. Tom Hales of the University of Pittsburgh, assisted by a group of computer scientists specializing in formalized proof-verification, is well on his way to verify his proof of the Kepler conjecture [Hal05], see [Hal]. The Anals of Mathematics published that proof and considered adding (but finally did not do it) a proviso, that the referees became exhausted (after 5 years) of checking all details by hand therefore the full correctness depends on a (perhaps not so reliable) computer computation. If Hales and his group succeed in formalizing and verifying the entire proof, then that will be of a reliability higher than most mathematical proofs, one third of which is estimated to contain real errors, not just typos[7].

The possibility of formalizing mathematics is not in contradiction with Gödel's theorem, which only states the limitations of the axiomatic method, informal or formal alike. The proof of Gödel's incompleteness theorem does in fact heavily rely on the fact that proof-checking is decidable and uses this by reflecting over the notion of provability (the Gödel sentence states: "This sentence is not provable").

One particular technology to verify that statements are valid is the use

[7] It is interesting to note that, although informal mathematics often contains bugs, the intuition of mathematicians is strong enough that most of these bugs usually can be repaired.

of model-checking. In IT applications the request “statement A can be proved from assumptions Γ (the ‘situation’)” often boils down to “A is valid in a model $\mathcal{A} = \mathcal{A}_\Gamma$ depending on Γ”. (In logical notation

$$\Gamma \vdash A \;\Leftrightarrow\; \mathcal{A}_\Gamma \models A.$$

This is so because of the completeness theorem of logic and because of the fact that the IT situation is related to models of digital hardware that are finite by its nature.) Now, despite the usual huge size of the model, using some cleverness the validity in several models in some industrially relevant cases is decidable within a feasible amount of time. One of these methods uses the so-called *binary decision diagrams* (BDD). Another ingredient is that universal properties are checked via some rewriting rules, like $(x + 1)(x - 1) = x^2 - 1$.

For an introduction to model-checkers, see [CJGP99]. For successful applications, see e.g. [Hol03]. The method of model-checking is often somewhat ad hoc, but nevertheless important. Using “automated abstraction”, that works in many cases, see [BGL$^+$00] and [Vaa06], the method becomes more streamlined.

4. Scaling-up through reflection

As to the question of whether fully formalized proofs are practically possible, the opinions have been divided. Indeed, it seems too much work to work out intuitive steps in full detail. Because of industrial pressure, however, full developments have been given for correctness of hardware and frequently used protocols. Formalizations of substantial parts of mathematics have been lagging behind.

There is a method that helps in tackling larger proofs. Suppose we want to prove statement A. Then it helps if we can write $A \leftrightarrow B(f(t))$, where t belongs to some collection X of objects, and we also can see that the truth of this is independent of t; i.e., one has a proof $\forall x \in X.B(f(x))$. Then $B(f(t))$, hence A. An easy example of this was conveyed to me by A. Mostowski in 1968. Consider the following formula as proof obligation in propositional logic:

$$A = p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow p)))))))))).$$

Then $A \leftrightarrow B(12)$, with $B(1) = p$, $B(n+1) = (p \leftrightarrow B(n))$. By induction on n one can show that for all natural numbers $n \geq 1$, one has $B(2 * n)$. Therefore, $B(12)$ and hence A, because $2 * 6 = 12$. A direct proof from the axioms of propositonal logic would be long. Much more sophisticated examples exist, but this is the essence of the method of reflection. It needs some form of computational reasoning inside proofs. Therefore, the modern mathematical assistants contain a model of computation for which equalities like $2 * 6 = 12$ and much more complex ones become provable. There are two ways to do this. One possibility is that there is a deduction rule of the form

$$\frac{A(s)}{A(t)} \quad s \twoheadrightarrow_R t.$$

This so-called *Poincaré Principle* should be interpreted as follows: From the assumption $A(s)$ and the side condition that s computationally reduces in several steps to t according to the rewrite system R, it follows that $A(t)$. The alternative is that the transition from $A(s)$ to $A(t)$ is only allowed if $s = t$ has been proved first. These two ways of dealing with proving computational statements can be compared with the styles of, respectively, functional and logical programming. In the first style, one obtains proofs that can be recorded as *proof-objects*. In the second style, these full proofs become too large to record as one object, because computations may take giga steps. Nevertheless the proof exists, but spread line by line over time, and one speaks about an *ephemeral* proof-object.

In the technology of proof-verification, general statements are about mathematical objects *and* algorithms, proofs show the correctness of statements *and* computations, and computations are dealing with objects *and* proofs.

Results

The state-of-the-art of computer-verified proofs is as follows. To formalize one page of informal mathematics, one needs four pages in a fully formalized style and it takes about five working days to produce these four pages, see [BW05]. It is expected that both numbers will go down. There have been formalized several nontrivial statements, like the fundamental theorem of algebra (also in a constructive fashion; it states that every non-constant polynomial over the complex numbers has a root), the prime number theorem (giving an asymptotic estimation of the number of primes below a given number), and the Jordan curve theorem (every closed curve divides the plane in two regions that cannot be reached without crossing this curve; on the torus surface this is not true). One of the great success stories is the full formalization of the Four Color Theorem by Gonthier, see [Gon05]. The original proof of this result was not completely trustable for its correctness, as a large number of cases needed to be examined by computer. Gonthier's proof still needs a computer-aided computation, but all steps have been formally verified by an assistant satisfying the de Bruijn principle.

References

[Abr98] J.-R. Abrial. On B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method: Second International B Conference Montpellier*, volume 1393 of *LNCS*, pages 1–8. Springer, 1998.

[AO97] K. R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1997. Second edition.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. 2*, Oxford Sci. Publ., pages 117–309. Oxford Univ. Press, New York, 1992.

[BG01] H.P. Barendregt and H. Geuvers. Proof-assistants Using Dependent Type Systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier Science Publishers B.V., 2001.

[BW05] H. P. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philos. Trans. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci.*, 363(1835):2351–2375, 2005.

[Bar06] H. P. Barendregt. Foundations of Mathematics from the Perspective of Computer Verification. In *Mathematics, Computer Science, Logic - A Never Ending Story*. Springer Verlag, 2006. To appear. Available at ⟨`www.cs.ru.nl/~henk/papers.html`⟩.

[BGL+00] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Mu noz, S. Owre, H. Rue, J. Rushby, V. Rusu, H. Sadi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

[BPS01] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of process algebra*. North-Holland Publishing Co., Amsterdam, 2001.

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[CJGP99] E. M.. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[Con97] Robert L. Constable. The structure of Nuprl's type theory. In *Logic of computation (Marktoberdorf, 1995)*, volume 157 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 123–155. Springer, Berlin, 1997.

[vD04] D. van Dalen. *Logic and structure*. Universitext. Springer-Verlag, Berlin, fourth edition, 2004.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration (Versailles, 1968)*, pages 29–61. Lecture Notes in Mathematics, Vol. 125. Springer, Berlin, 1970.

[Dav04] M. Davis, editor. *The undecidable*. Dover Publications Inc., Mineola, NY, 2004. Basic papers on undecidable propositions, unsolvable problems and computable functions, Corrected reprint of the 1965 original [Raven Press, Hewlett, NY].

[Ede97] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 37:54–67, 1997.

[Gon05] G. Gonthier. A computer checked proof of the Four Colour Theorem. Unpublished. Available from URL: ⟨`research.microsoft.com/~gonthier/4colproof.pdf`⟩, 2005.

[GHH+02] B. Greer, J. Harrison, G. Henry, W. Li, and P. Tang. Scientific computing on the itanium^r processor. *Scientific Programming*, 10(4):329–337, 2002.

[Hal05] T. C. Hales. A proof of the Kepler conjecture. *Ann. of Math. (2)*, 162(3):1065–1185, 2005.

[Hal] T. C. Hales. The flyspeck project fact sheet. URL: <`www.math.pitt.edu/~thales/flyspeck/index.html`>.

[Hal74] P. R. Halmos. *Naive set theory.* Springer-Verlag, New York, 1974. Reprint of the 1960 edition, Undergraduate Texts in Mathematics.

[HJ98] C.A.P Hoare and H. Jifeng. *Unifying theories of programming.* Prenrice Hall, 1998.

[Hod97] Wilfrid Hodges. *A shorter model theory.* Cambridge University Press, Cambridge, 1997.

[Hol03] G. J. Holzmann. *The SPIN model checker, primer and reference manual.* Addison-Wesley, 2003.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[McC62] J. McCarthy. Computer programs for checking the correctness of mathematical proofs. In *Proceedings of a Symposium in Pure Methematics, vol. V.*, pages 219–227. American Mathematical Society, Providence, RI, 1962.

[ML84] P. Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[Mor01] B.-A. Mordechai. *Mathematical Logic for Computer Science.* Springer, 2001.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. Twenty-five years of Automath research. In *Selected papers on Automath*, volume 133 of *Stud. Logic Found. Math.*, pages 3–54. North-Holland, Amsterdam, 1994.

[Vaa06] F.W. Vaandrager. Does it pay off? model-based verification and validation of embedded systems! In F.A. Karelse, editor, *PROGRESS White papers 2006*. STW, the Netherlands, 2006. URL: ⟨`www.cs.ru.nl/ita/publications/papers/fvaan/whitepaper`⟩.

[Wie06] F. Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006.

[Wup98] H. Wupper. Design as the discovery of a mathematical theorem - What designers should know about the art of mathematics. In Ertas et al., editor, *Proc. Third Biennial World Conf. on Integrated Design and Process Technology (IDPT)*, pages 86–94. Soc. Des. & Proc. Sc., 1998.