The Challenge of Computer Mathematics

By Henk Barendregt and Freek Wiedijk

Radboud University Nijmegen
The Netherlands

Progress in the foundations of mathematics has made it possible to formulate all thinkable mathematical concepts, algorithms and proofs in one language and in an impeccable way. This is not in spite of, but partially based on the famous results of Gödel and Turing. In this way statements are about mathematical objects *and* algorithms, proofs show the correctness of statements *and* computations, and computations are dealing with objects *and* proofs. Interactive computer systems for a full integration of defining, computing and proving are based on this. The human defines concepts, constructs algorithms and provides proofs, while the machine checks that the definitions are well-formed and the proofs and computations are correct. Results formalised so far demonstrate the feasibility of this 'Computer Mathematics'. Also there are very good applications. The challenge is to make the systems more mathematician-friendly, by building libraries and tools. The eventual goal is to help humans to learn, develop, communicate, referee and apply mathematics.

Keywords: Computer Mathematics, Formalised Proofs, Proof Checking.

Acknowledgements. The authors thank Mark van Atten, Wieb Bosma, Femke van Raamsdonk and Bas Spitters for useful input.

1. The Nature of Mathematical Proof

Proofs in mathematics have come to us from the ancient Greeks. The notion of proof is said to have been invented by Thales (ca. 624-547 BC). For example he demonstrated that the angles at the bases of any isosceles triangle are equal. His student Pythagoras (ca. 569-475 BC) went on proving among other things the theorem bearing his name. Pythagoras started a society of followers, half religious, half scientific, that continued after he passed away. One member of the society was Theodorus (465-398 BC), who taught the philosopher Plato (428-347 BC) the irrationality of $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{6}$ etcetera up to $\sqrt{17}$. Plato emphasised to his students the importance of mathematics, with its proofs that show non-obvious facts with a clarity such that everyone can understand them. In Plato's dialogue Meno a slave was requested by Socrates (469-399 BC) to listen and answer, and together, using the maieutic method, they came to the insight that the size of the long side of an isosceles rectangular triangle is, in modern terminology, $\sqrt{2}$ times the size of the shorter side. Not much later the subject of mathematics was evolved to a sufficient degree that Plato's student Aristotle (384-322 BC) could reflect about this discipline. He described the *axiomatic method* as follows. Mathematics consists of objects and of valid statements. Objects are defined from previously defined objects; in order to be able to get started one has the *primitive objects*. Valid statements are *proved* from other such statements; in order to get started one has the *axioms*.

Euclid (ca. 325-265 BC) wrote just a few decades later his monumental *Elements* describing geometry in this axiomatic fashion. Besides that, the *Elements* contain the first important results in number theory (theory of divisibility, prime numbers, factorisation) and even Eudoxos' (408-355 BC) account of treating ratios (that was later the inspiration for Dedekind (1831-1916) to give a completely rigorous description of the reals as cuts of rational numbers).

During the course of history of mathematics proofs increased in complexity. In particular in the 19-th century some proofs could no longer be followed easily by just any other capable mathematician: one had to be a specialist. This started what has been called the sociological validation of proofs. In disciplines other than mathematics the notion of *peer review* is quite common. Mathematics for the Greeks had the 'democratic virtue' that anyone (even a slave) could follow a proof. This somewhat changed after the complex proofs appeared in the 19-th century that could only be checked by specialists. Nevertheless mathematics kept developing and having enough stamina one could decide to become a specialist in some area. Moreover, one did believe in the review by peers, although occasionally a mistake remained undiscovered for many years. This was the case e.g. with the erroneous proof of the Four Colour Conjecture by Kempe [1879].

Verifiable by	Theorems
Lay person/ Student	$\sqrt{2}$ is irrational There are infinitely many primes
Competent mathematician	Fundamental Theorem of Algebra
Specialist	Fermat's Last Theorem
Group of specialists	Classification of the finite simple groups
Computer	Four Colour Theorem Kepler's Conjecture

Figure 1. Theorems and their verification

In the 20-th century this development went to an extreme. There is the complex proof of Fermat's Last Theorem by Wiles. At first the proof contained an error, discovered by Wiles himself, and later his new proof was checked by a team of twelve specialist referees†. Most mathematicians have not followed in detail the proof of Wiles, but feel confident because of the sociological verification. Then there is the proof of the Classification of the Finite Simple Groups. This proof was announced in 1979 by a group of mathematicians lead by Gorenstein. The proof consisted of a collection of connected results written down in various places, totalling 10,000 pages. In the proof one relied also on 'well-known' results and it turned out that not all of these were valid. Work towards improving the situation has been performed,

† One of these referees told us the following. "If an ordinary non-trivial mathematical paper contains an interesting idea and its consequences and obtains 'measure 1', then Wiles' proof can be rated as having measure 156."

and in Aschbacher [2004] it is announced that at least this author believes in the validity of the theorem. Finally there are the proofs of the Four Colour Theorem, Appel and Haken [1977a], [1977b] and Robertson et al. [1996], and of Kepler's Conjecture, Hales [to appear]. All these proofs use a long computation performed on a computer. (Actually Aschbacher [2004] believes that at present also the proof of the Classification of the Finite Simple Groups relies on computer performed computation.) The situation is summed up in table 1.

A very different development, defended by Zeilberger and others, consists of admitting proofs where the result is not 100% certain, but say 99.9999999999. Examples of these proofs concern the primality of large numbers, see Miller [1976], Rabin [1980].

In this situation the question arises whether there has been a necessary devaluation of proofs. One may fear that the quote of Benjamin Peirce (1809-1880) "Mathematics is the science which draws necessary conclusions" may not any longer hold. Scientific American even ventured an article called 'The Death of Proof', see Horgan [1993]. The Royal Society Symposium 'The Nature of Mathematical Proof' (London, October 18-19, 2004) had a more open-minded attitude and genuinely wanted to address the question. We will argue below that proofs remain alive and kicking and at the heart of mathematics. There is a sound methodology to ensure the full correctness of theorems with large proofs, even if they depend on complex computations, like the Four Colour Theorem, or on a sociological verification, like the Classification of the Finite Simple Groups.

Phenomenology

From where does the confidence come that is provided by a proof in mathematics? When a student asks the teacher: "Sir, am I allowed to do this step?", the answer we often give is "When it is convincing, both for you and me!". Mathematics is rightly considered as the most exact science. It is not too widely known to outsiders that this certainty eventually relies on a mental judgement. It is indeed the case that proofs and computations are a warranty for the exactness of mathematics. But both proofs and computations need a judgement that the performed steps are correct and applicable. This judgement is based on a trained form of our intuition. For this reason Husserl [1901], and also Gödel [1995], and notably Bernays in Hao Wang [1997], p.337, 10.2.7, emphasise the phenomenological character of the act of doing mathematics.

Computation vs. Intuition

In Buddhist psychology one distinguishes discursive versus intuitive knowledge. In order to explain this a contemporary example may be useful. Knowing physics one can calculate the range of angles a bike rider may use while making a right turn. This is discursive knowledge; it does not enable someone to ride a bike. On the other hand a person who knows how to ride a bike 'feels' the correct angles by intuition, but may not be able to compute them. Both forms of knowledge are useful and probably use different parts of our brain.

For the mental act of doing mathematics one may need some support. In fact, before the Greek tradition of proofs, there was the Egyptian-Chinese-Babylonian

tradition of mathematics as the art of computing. Being able to use computational procedures can be seen as discursive knowledge. This aspect is often called the 'algebraic' side of mathematics. On the other hand proofs often rely on our intuition. One speaks loosely about the intuitive 'geometric' side of mathematics.

A computation like $13338 \times 3145727 = 41957706726$ needs do be done on paper or by some kind of computer (unless we are an *idiot savant*; this computation is related to the famous 'Pentium bug' appearing in 1994). Symbolic manipulations, like multiplying numbers or polynomials, performing symbolic integrations and arbitrary other 'algebraic' computations may not be accompanied by intuition. Some mathematicians like to use their intuition, while others prefer algebraic operations. Of course knowing both styles is best. In the era of Greek mathematics at first the invention of proofs with its compelling exactness drew attention away from computations. Later in the work of Archimedes (287-212 BC) both computations and intuition did excel.

The story repeated itself some two millennia later. The way in which Newton (1643-1727) introduced calculus was based on the solid grounds of Euclidean geometry. On the other hand Leibniz (1646-1716) based his calculus on infinitesimals that had some dubious ontological status (do they really exist?). But Leibniz' algebraic approach did lead to many fruitful computations and new mathematics, as witnessed by the treasure of results by Euler (1707-1783). Infinitesimals did lead to contradictions. But Euler was clever enough to avoid these. It was only after the foundational work of Cauchy (1789-1857) and Weierstrass (1815-1897) that full rigour could be given to the computational way of calculus. That was in the 19-th century and mathematics bloomed as never before, as witnessed by the work of Gauss (1777-1855), Jacobi (1804-1851), Riemann (1826-1866) and many others.

During the last third of the 20-th century the 'schism' between computing and proving reoccurred. Systems of computer algebra, being good at symbolic computations, were at first introduced for applications of mathematics in physics: a pioneering system is Schoonschip, see Veltman [1967], which helped win a Nobel prize in physics. Soon they became useful tools for pure mathematics. Their drawback is that the systems contain bugs and cannot state logically necessary side-conditions for the validity of the computations. On the other hand systems for proof-checking on a computer have been introduced, the pioneer being Automath of de Bruijn, see Nederpelt et al. [1994]. These systems are able to express logic and hence necessary side conditions, but at first they were not good at making computations. The situation is changing now as will be seen below.

Computer Science Proofs

Programs are elements of a formal (i.e. precisely defined) language and thereby they become mathematical objects. It was pointed out by Turing [1949] that one needs a proof to show that a program satisfies some desired properties. This method was refined and perfected by Floyd [1967] and Hoare [1969]. Not all software has been specified, leave alone proven correct, as it is often hard to know what one exactly wants from it. But for parts of programs and for some complete programs that are small but vital (like protocols) proofs of correctness have been given. The methodology of (partially) specifying software and proving that the required property holds for the program is called 'Formal Methods'.

Proofs for the correctness of software are often long and boring, relying on nested case distinctions, contrasting proofs in mathematics that are usually more deep. Therefore the formal methods ideal seemed to fail: who would want to verify the correctness proofs, if they were longer than the program itself and utterly uninspiring. Below we will see that also this situation has been changed.

2. Foundations of Mathematics

A foundation for mathematics asks for a formal language in which one can express mathematical statements and a system of derivation rules using which one can prove some of these statements. In order to classify the many objects that mathematicians have considered an 'ontology', describing ways in which collections of interest can be defined, comes in handy. This will be provided by set theory or type theory. Finally one also needs to provide a model of computation in which algorithms performed by humans can be represented in one way or another. In other words, one needs Logic, Ontology and Computability.

Logic

Elimination rules	Introduction rules
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \to B}{\Gamma \vdash B}$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$
$\dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A} \quad \dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$
$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$
$\dfrac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x := t]}\ t \text{ is free in } A$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ x \notin \Gamma$
$\dfrac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ x \notin C$	$\dfrac{\Gamma \vdash A[x := t]}{\Gamma \vdash \exists x.A}$
$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}$	$\dfrac{}{\Gamma \vdash \top}$
Start rule	**Double-negation rule**
$\dfrac{}{\Gamma \vdash A}\ A \in \Gamma$	$\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$

Figure 2. Predicate Logic Natural Deduction Style

Not only did Aristotle describe the axiomatic method, he also started the quest for logic. This is the endeavour to chart the logical steps needed in mathematical

reasoning. He started a calculus for deductions. The system was primitive: not all connectives (and, or, implies, not, for all, exists) were treated, only monadic predicates, like $P(n)$ 'n is a prime number', and not binary ones, like $R(n,m)$ '$n < m$', were considered. Nevertheless, the attempt to find rules sufficient for reasoning was quite daring.

The quest for logic, as needed for mathematical reasoning, was finished 2300 years later in Frege [1971]. Indeed, Gödel [1930] showed that Frege's system was complete (mathematically this was done already by Skolem [1922], but the result itself was not mentioned there). This means that from a set of hypotheses Γ a statement A can be derived iff in all structures $\mathfrak{A}$ in which the hypotheses Γ hold, also the conclusion A holds.

A particular nice version of logic was given by Gentzen in 1934 (see his collected papers [1969]). This system is presented in Figure 2. Some explanations are in order. The signs $\rightarrow, \&, \vee, \top, \forall, \exists$ stand for 'implies', 'and', 'or', 'true', 'for all' and 'exists', respectively. $\neg A$ stands for 'not A' and is defined as $A \rightarrow \bot$, where $\bot$ stands for a ('the') false statement (like 0=1). Γ stands for a set of statements and $\Gamma \vdash A$ stands for 'from the set Γ the statement A is derivable'. A rule like

$$\frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}$$

has to be read as follows: 'If $A \,\&\, B$ is derivable from Γ, then so is A.'

First-, second- and higher-order logic

The logic presented is first-order logic. It speaks about the element of a structure $\mathfrak{A}$ (a set with some given operations and relations on it) and can quantify over the elements of this structure. In second-order logic one can quantify over subsets of the structure $\mathfrak{A}$, i.e. over $\mathcal{P}(\mathfrak{A})$. Then there is higher-order logic, that can quantify over each $\mathcal{P}^n(\mathfrak{A})$. In first-order logic one can distinguish the difference between continuity and uniform continuity of a given function (say on $\mathbb{R}$).

$$\forall x{\in}\mathbb{R}\, \forall \epsilon{>}0\, \exists \delta{>}0\, \forall y{\in}\mathbb{R}\,.|x-y|{<}\delta \;\Rightarrow\; |f(x)-f(y)|{<}\epsilon$$

versus

$$\forall \epsilon{>}0\, \exists \delta{>}0\, \forall x, y{\in}\mathbb{R}\,.|x-y|{<}\delta \;\Rightarrow\; |f(x)-f(y)|{<}\epsilon.$$

Here $\forall \epsilon{>}0. \ldots$ has to be translated to $\forall \epsilon.[\epsilon{>}0 \;\Rightarrow\; \ldots]$.

In second-order logic one may express that an element x of a group G has torsion (a power of x is the unit element e) without having the notion of natural number:

$$\forall X{\in}\mathcal{P}(G).x{\in}X \;\&\; [\forall y{\in}X.(x \cdot y){\in}X] \;\Rightarrow\; e{\in}X.$$

This states that e belongs to the intersection of all subsets of G that contain x and that are closed under left-multiplication by x.

In higher-order logic one may state that there exists a non-trivial topology on $\mathbb{R}$ that makes a given function f continuous:

$$\exists \mathcal{O}{\in}\mathcal{P}^2(\mathbb{R}).\mathcal{O} \text{ is a non-trivial topology } \&\; \forall O{\in}\mathcal{O}.f^{-1}(O){\in}\mathcal{O}.$$

Here $\mathcal{O}$ is a non-trivial topology stands for

$$\begin{array}{l}\mathcal{O} \neq \mathcal{P}(\mathbb{R}) \;\&\; \mathbb{R}{\in}\mathcal{O} \;\&\\ \forall \mathcal{X}{\in}\mathcal{P}(\mathcal{O}).[\emptyset \neq \mathcal{X} \rightarrow \bigcup \mathcal{X}{\in}\mathcal{O}] \;\&\\ \forall X, Y{\in}\mathcal{O}.X \cap Y{\in}\mathcal{O}.\end{array}$$

Intuitionistic logic

Not long after the first complete formalisation of (first-order) logic was given by Frege, Brouwer criticised this system of 'classical logic'. It may promise an element when a statement like

$$\exists k.P(k)$$

has been proved, but nevertheless it may not be the case that a *witness* is found, i.e. one may not know how to prove any of

$$P(0),\ P(1),\ P(2),\ P(3),\ \ldots\ .$$

For example, this is the case for the statement

$$P(x) := (x = 0 \;\&\; \mathsf{RH}) \vee (x = 1 \;\&\; \neg\mathsf{RH}),$$

where RH stands for the Riemann Hypothesis that can formulated in (Peano) Arithmetic. The only possible witnesses are 0 and 1. By classical logic $\mathsf{RH} \vee \neg\mathsf{RH}$ holds. In the first case one can take $x = 0$ and in the second case $x = 1$. Therefore one can prove $\exists x.P(x)$. One, however, cannot provide a witness, as $P(0)$ can be proved only if the RH is proved and $P(1)$ can be proved only if the RH is refuted. At present neither is the case. One may object that 'tomorrow' the RH may be settled. But then one can take another open problem instead of the RH, or an independent statement, for example a Gödel sentence G stating that

$$\text{'}G\text{' is not provable}$$

or the Continuum Hypothesis $2^{\aleph_0} = \aleph_1$ (if we are in set theory). A similar criticism can be addressed to provable statements of the form $A \vee B$. These can be provable, while neither A nor B can be proved.

Brouwer analysed that the law of excluded middle, $A \vee \neg A$ is the cause of this unsatisfactory situation. He proposed to do mathematics without this 'unreliable' logical principle. In Heyting [1930] an alternative logic was formulated. For this logic one can show that

$$\vdash A \vee B \;\Leftrightarrow\; \vdash A \text{ or } \vdash B,$$

and similarly

$$\vdash \exists x.P(x) \;\Leftrightarrow\; \vdash P(t), \text{ for some expression } t.$$

Gentzen provided a convenient axiomatisation of both classical and intuitionistic logic. In Figure 2 the system of classical logic is given; if one leaves out the rule of double negation one obtains the system of intuitionistic logic.

Ontology

Ontology is the philosophical theory of 'existence'. Kant remarked that existence is not a predicate. He probably meant that in order to state that something exists we already must have it. Nevertheless we can state that there exists a triple (x, y, z) of positive integers such that $x^2 + y^2 = z^2$ (as Pythagoras knew), but not such that $x^3 + y^3 = z^3$ (as Euler knew). Ontology in the foundations of mathematics focuses on collections of objects O, so that one may quantify over it (i.e. stating $\forall x{\in}O.P(x)$, or $\exists x{\in}O.P(x)$). Traditional mathematics only needed a few of these collections: number systems and geometric figures. From the 19-th century on a wealth of new spaces was needed and ample time was devoted to constructing these. Cantor (1845-1918) introduced set theory that has the virtue of bringing together all possible spaces within one framework. Actually this theory is rather strong and not all postulated principles are needed for the development of mathematics. An interesting alternative is type theory in which the notion of function is a first class object.

Set Theory

Postulated are the following axioms of 'set existence':

		$\mathbb{N}$	$(infinity)$
a, b	$\mapsto$	$\{a, b\}$	(pair)
a	$\mapsto$	$\cup a = \cup_{x\in a} x$	(union)
a	$\mapsto$	$\{x \mid x \subseteq a\} = \mathcal{P}(a)$	(powerset)
a	$\mapsto$	$\{x{\in}a \mid P(a)\}$	(comprehension)
a	$\mapsto$	$\{F(x) \mid x{\in}a\}$	(replacement)

These axioms have as intuitive interpretation the following. $\mathbb{N}$ is a set; if a, b are sets, then $\{a, b\}$ is a set; ... ; if P is a property over sets, then $\{x{\in}a \mid P(x)\}$ is a set; if for every x there is given a unique $F(x)$ in some way or another, then $\{F(x) \mid x{\in}a\}$ is a set. We will not spell out the way the above axioms have to be formulated and how P and F are given, but refer the reader to a textbook on axiomatic set theory, see e.g. Kunen [1983]. Also there are the axioms of 'set properties':

$$a = b \;\Leftrightarrow\; \forall x.[x{\in}a \;\Leftrightarrow\; x{\in}b] \qquad \text{(extensionality)}$$
$$\forall a.[[\exists x.x{\in}a] \;\Rightarrow\; \exists x.[x{\in}a \;\&\; \neg\exists y.y{\in}x \;\&\; y{\in}a]] \qquad \text{(foundation)}$$

The axiom of extensionality states that a set is completely determined by its elements. The axiom of foundation is equivalent with the statement that every predicate P on sets is well-founded: if there is a witness x such that $P(x)$ holds, then there is a minimal witness x. This means that $P(x)$ but for no $y{\in}x$ one has $P(y)$. Another way to state foundation: $\forall a\neg\exists f{\in}(\mathbb{N}{\to}a)\forall n{\in}\mathbb{N}.f(n+1){\in}f(n)$.

Type Theory

Type Theory, coming in several variants, forms an alternative to set theory. Postulated are inductively defined data types with their recursively defined functions. Moreover types are closed under function spaces and products. A type may be

thought of as a set and that an element `a` belongs to type `A` is denoted by `a : A`. The difference with set theory is that in type theory an element has a unique type. Inductive types are given in the following examples (boolean, natural numbers, lists of elements of `A`, binary trees with natural numbers at the leafs).

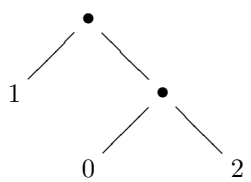
```
bool    :=  true:bool | false:bool
nat     :=  0:nat | S:nat->nat
list_A  :=  nil:list_A | cons:A->list_A->list_A
tree    :=  leaf:nat->tree | branch:tree->tree->tree
A×B     :=  pair:A->B->A×B
```

These definitions should be read as follows. The only elements of `bool` are `true, false`. The elements of `nat` are freely generated from `0` and the unary 'constructor' `S`, obtaining $\mathtt{0}, \mathtt{S}(\mathtt{0}), \mathtt{S}(\mathtt{S}(\mathtt{0})), \ldots$. One writes for elements of `nat` 1=`S(0)`, 2=`S(1)`, ... A typical element of `list_nat` is

$$\langle 1, 0, 2\rangle = \mathtt{cons}(\mathtt{1}, \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{2}, \mathtt{nil}))).$$

A typical tree is

$$\mathtt{branch}(\mathtt{leaf}(\mathtt{1}), \mathtt{branch}(\mathtt{leaf}(\mathtt{0}), \mathtt{leaf}(\mathtt{2}))) =$$



A typical element of $\mathtt{A} \times \mathtt{B}$ is $\langle \mathtt{a}, \mathtt{b}\rangle$ =`pair(a,b)`, where `a:A, b:B`. Given types `A, B`, one may form the 'function-space' type `A->B`. There is the primitive operation of application: if $\mathtt{f} : \mathtt{A{-}{>}B}$ and $\mathtt{a} : \mathtt{A}$, then $\mathtt{f}(\mathtt{a}) : \mathtt{B}$. Conversely there is abstraction: if $\mathtt{M} : \mathtt{B}$ 'depends on' an $\mathtt{a} : \mathtt{A}$ (like $\mathtt{a}^2 + \mathtt{a} + \mathtt{1}$ depends on $\mathtt{a} : \mathtt{nat}$) one may form the function $\mathtt{f} := (\mathtt{a} \mapsto \mathtt{M}) : (\mathtt{A{-}{>}B})$. This function is denoted by $\lambda \mathtt{a} : \mathtt{A}.\mathtt{M}$ (function abstraction). For example this can be used to define composition: if $\mathtt{f} : \mathtt{A{-}{>}B}, \mathtt{g} : \mathtt{B{-}{>}C}$, then $\mathtt{g} \circ \mathtt{f} := \lambda \mathtt{a}{:}\mathtt{A}.\mathtt{g}(\mathtt{f}(\mathtt{a})) : \mathtt{A{-}{>}C}$. Next to the formation of function space types there is the *dependent cartesian product*. If `B` is a type that depends on an $\mathtt{a} : \mathtt{A}$, then one may form $\Pi \mathtt{a}{:}\mathtt{A}.\mathtt{B}$. One has (here $\mathtt{B}[\mathtt{a} := \mathtt{t}]$ denotes the result of substitution of `t` in `B` for `a`)

$$\mathtt{f} : (\Pi \mathtt{a}{:}\mathtt{A}.\mathtt{B}),\ \mathtt{t} : \mathtt{A} \ \Rightarrow\ \mathtt{f}(\mathtt{t}) : \mathtt{B}[\mathtt{a} := \mathtt{t}].$$

A typical example is $\mathtt{B} = \mathtt{A}^{\mathtt{n}}$ for $\mathtt{n}{:}\mathtt{nat}$. If $\mathtt{f}{:}(\Pi \mathtt{n}{:}\mathtt{nat}.\mathtt{A}^{\mathtt{n}})$, then $\mathtt{f}(\mathtt{2n}) : \mathtt{A}^{\mathtt{2n}}$. Type theories are particularly convenient to express intuitionistic mathematics. Type theories differ as to what dependent cartesian products and what inductive types are allowed, whether or not they are predicative†, have 'powersets', the axiom of choice. See Martin-Löf [1984], Aczel and Rathjen [2001], Barendregt and Geuvers [2001] and Moerdijk and Palmgren [2002]. In Feferman [1998], Ch. 14, a type-free system (which can be seen as a system as a type system with just one type) is presented for predicative mathematics.

† In predicative sytems a subset of an infinite set X can only be defined if one does not refer to the class of all subsets of X. For example $\{n{\in}\mathbb{N} \mid n \text{ is even}\}$ is allowed, but not

$$\{n{\in}\mathbb{N} \mid \forall X \subseteq \mathbb{N}.P(X, n)\}.$$

Computability

Mathematical algorithms are much older than mathematical proofs. They have been introduced in Egyptian-Babylonian-Chinese mathematics a long time before the notion of proofs. In spite of that, reflection over the notion of computability through algorithms has appeared much later, only about 80 years ago. The necessity came when Hilbert announced in 1900 his famous list of open problems. His 10-th problem was the following.

"*Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers*‡."

By a number a steps over a time interval of nearly 50 years, the final one by Matijasevic using the Fibonacci numbers, this problem was shown to be undecidable, see Davis [1973]. In order to be able to state such a result one needed to reflect over the notion of algorithmic computability.

Steps towards the formalisation of the notion of computability were done by Skolem, Hilbert, Gödel, Church and Turing. At first Hilbert [1926] (based on work by Grassmann, Dedekind, Peano and Skolem) introduced the primitive recursive functions over $\mathbb{N}$ by the following schemata‡.

$$\begin{array}{rcl} Z(x) & = & 0; \\ S(x) & = & x+1; \\ P^n_k(x_1,\ldots,x_n) & = & x_k; \\ f(\vec{x},0) & = & g(\vec{x}); \\ f(\vec{x},y+1) & = & h(\vec{x},y,f(\vec{x},y)). \end{array}$$

Figure 3. The primitive recursive functions

It was shown by Sudan [1927] and Ackermann [1928] that not all computable functions were primitive recursive. Then Gödel (based on a suggestion of Herbrand) introduced the notion of totally defined computable functions¶, based on what is called nowadays *Term Rewrite Systems*, see Terese [2003]. This class of total computable functions can also be obtained by adding to the primitive recursive schemata the scheme of minimalisation ('$\mu y \ldots$' stands for 'the least y such that $\ldots$'), Kleene [1936].

$$f(\vec{x}) \;=\; \mu y.[g(\vec{x},y)=0], \quad \text{provided that } \; \forall \vec{x} \exists y. g(\vec{x},y) = 0.$$

‡ By 'rational integers' Hilbert just meant the set of integers $\mathbb{Z}$. This problem is equivalent to the problem over $\mathbb{N}$. The solvability of Diophantine equations over $\mathbb{Q}$ is still open.

‡ This definition scheme was generalised by Scott [1970] to inductive types. For example over the binary trees introduced above one can define a primitive recursive function mirror as follows.

$$\begin{array}{rcl} \texttt{mirror}(\texttt{leaf}(n)) & = & \texttt{leaf}(n) \\ \texttt{mirror}(\texttt{branch}(t_1,t_2)) & = & \texttt{branch}(\texttt{mirror}(t_2),\texttt{mirror}(t_1)) \end{array}$$

It mirrors the tree displayed above.

¶ Previously called *(total) recursive functions.*

Finally it was realised that it is more natural to formalise computable partial functions. This was done by Church [1936] using lambda calculus, and Turing [1936] using what are now called Turing machines. The formalised computational models of Turing and Church later gave rise to the so-called imperative and functional programming styles. The first is more easy to be implemented, the second more easy to use and to show the correctness of the programs.

Both the computational models of Church and Turing have a description about as simple as that of the first-order predicate calculus. More simple is the computational model given in Schönfinkel [1924] that is also capturing all partial computable functions. It is a very simple example of a Term Rewrite System.

$$\begin{array}{lcl} \mathsf{S}xyz & = & xz(yz) \\ \mathsf{K}xy & = & x \end{array}$$

Figure 4. CL Combinatory Logic

The system is based on terms built up from the constants K, S under a binary operation (application). Various forms of data (natural numbers, trees, etcetera) can be represented as K, S expressions. Operations on this represented data can be performed by other such expressions.

The Compactness of the Foundations

The study of the foundations of mathematics has achieved the following. The triple activity of defining, computing and reasoning can be described in each case by a small set of rules. This implies that it is decidable whether a (formalised) putative proof p (from a certain mathematical context) is indeed a proof of a given statement A (in that context). This is the basis of the technology of Computer Mathematics. For more on the relation between the foundational studies and Computer Mathematics, see Barendregt [2005].

3. Computer Mathematics

In systems for Computer Algebra one can deal with mathematical objects like $\sqrt{2}$ with full precision. The idea is that this number is represented as a symbol, say α, and that with this symbol one computes symbolically. One has $\alpha^2 - 2 = 0$ but $\alpha + 1$ cannot be simplified. This can be done, since the computational rules for $\sqrt{2}$ are known. In a vague sense $\sqrt{2}$ is a 'computable object'. There are many other computable objects like expressions dealing with transcendental functions $(e^x, \log x)$ and integration and differentiation.

In systems for Computer Mathematics, also called Mathematical Assistants, one can even represent non-computable objects. For example the set S of parameters for which a Diophantine equation is solvable. Also these can be represented on a computer. Again the non-computable object is represented by a symbol. This time one cannot simply compute whether a given number, say 7, belongs to S. Nevertheless one can state that it does and in some cases one may prove this. If one provides a proof of this fact, then that proof can be checked and one can add $7 \in S$ to the database of known results and use it in subsequent reasoning. In short, although provability is undecidable, being a proof of a given statement is decidable

and this is the basis of systems for Computer Mathematics. It has been the basis for informal mathematics as well.

One may wonder whether proofs verified by a computer are at all reliable. Indeed, many computer programs are faulty. It was emphasised by de Bruijn that in case of verification of formal proofs, there is an essential gain in reliability. Indeed a verifying program only needs to see whether in the putative proof the small number of logical rules are always observed. Although the proof may have the size of several Megabytes, the verifying program can be small. This program then can be inspected in the usual way by a mathematician or logician. If someone does not believe the statement that a proof has been verified, one can do independent checking by a trusted proof-checking program. In order to do this one does need formal proofs of the statements. A Mathematical Assistant satisfying the possibility of independent checking by a small program is said to satisfy the *de Bruijn* criterion.

Of particular interest are proofs that essentially contain computations. This happens on all levels of complexity. In order to show that a linear transformation A on a finite dimensional vector space has a real eigenvalue one computes

$$\det(A - \lambda I) = p(\lambda)$$

and determines whether $p(\lambda)$ has a real root. In order to show that a polynomial function F vanishes identically on some variety V, one computes a Groebner basis to determine whether F is contained in the ideal generated by the equations defining V, see Buchberger and Winkler [1998].

Although it is shown that provability in general is undecidable, for interesting particular cases the provability of statements may be reduced to computing. These form the decidable cases of the decision problem. This will help Computer Mathematics considerably. Tarski [1951] showed that the theory of real closed fields (and hence elementary geometry) is decidable. An essential improvement was given by Collins [1975]. In Buchberger [1965] a method to decide membership of finitely generated ideals in certain polynomial rings was developed. For polynomials over $\mathbb{R}$ this can be done also by the Tarski-Collins method, but much less efficiently so. Moreover, 'Buchberger's algorithm' was optimised by e.g. Bachmair and Ganzinger [1994].

In order to show that the Four Colour Theorem holds one checks 633 configurations are 'reducible', involving millions of cases, see Robertson et al. [1996]. How can such computations be verified? All these cases can be stylistically rendered as $f(a) = b$ that needs to be verified. In order to do this one first needs to represent f in the formal system. One way to do this is to introduce a predicate $P_f(x, y)$ such that for all a, b (say natural numbers) one has

$$f(a) = b \;\Leftrightarrow\; \vdash P_f(\underline{a}, \underline{b}).$$

Here '$\vdash$' stands for provability. If e.g. $a = 2$, then $\underline{a} = S(S(0))$, a representation of the object 2 in the formal system†. In these languages algorithms are represented as so called 'logical programs', as happened also in Gödel [1931]. In other formal

† For substantial computations one needs to introduce decimal (or binary) notation for numbers and prove that the operations on them are correctly defined. In the history of mathematics it was al-Khowârizmî (780-850) who did not introduce algorithms as the name suggests, but proved that the well-known basic operations on the decimal numbers are correctly taught.

theories, notably those based on type theory, the language itself contains expressions for functions and the representing predicate has a particularly natural form

$$P_f(x, y) \;:=\; (F(x) = y).$$

This is the representation of the algorithm in the style of functional programming. Of course this all is not enough. One also needs to prove that the computation is relevant. For example in the case of linear transformations one needs a formal proof of

$$P_f(\underline{A}, \underline{0}) \;\leftrightarrow\; A \text{ has an eigenvalue.}$$

But once this proof is given and verified one only needs to check instances of $P_f(\underline{a}, \underline{b})$ for establishing $f(a) = b$.

There are two ways of doing this. In the first one the computation trace is produced and annotated by steps in the logical program P_f (respectively functional program F). This produces a very long proof (in the order of the length of computation of $f(a) = b$) that can be verified step by step. Since the resulting proofs become long, they are usually not stored, but only the local steps to be verified ('Does this follow from that and that?'). One therefore can refer to these as *ephemeral proofs*.

On the other hand there are systems in which proofs are fully stored for later use (like extraction of programs from them). These may be called *petrified proofs*. In systems with such proofs one often has adopted the Poincaré Principle. This principle states that for a certain class of equations $t = s$ no proofs are needed, provided that their validity can be checked by an algorithm. This puts somewhat of a strain on the de Bruijn criterion requiring that the verifying program be simple. But since the basic steps in a universal computational model are simple, this is justifiable.

4. The Nature of the Challenge

State of the Art: Effort and Space

Currently there are not many people who formalise mathematics with the computer, but that does not mean that the field of Computer Mathematics is not yet mature. The full formalisation of all of undergraduate university mathematics is within reach of current technology. Formalising on that level will be labour-intensive, but it will not need any advances in proof assistant technology.

To give an indication of how much work is needed for formalisation, we estimate that it takes approximately *one work-week* (five work-days of eight work-hours) to formalise one page from an undergraduate mathematics textbook. This measure for some people is surprisingly low, while for others it is surprisingly high. Some people think it is impossible to formalise a non-trivial theorem in full detail all the way down to the axioms, and this measure shows that they are wrong. On the other hand it takes much longer to formalise a proof than it takes to write a good informal version of it (this takes about half a workday per page: which is a factor of ten smaller).

One can also compare the formal version of a mathematical proof with the corresponding informal—'traditional'—way of presenting that proof. We investigated in Wiedijk [2000] that a file containing a full formalisation of a mathematical theory is

approximately *four* times as long as the LaTeX source of the informal presentation. We call this factor the *de Bruijn factor*, as de Bruijn claimed that this ratio is a constant, which does not change when one proceeds in formalising a mathematical theory. Some researchers actually believe that the factor decreases as the theory grows.

State of the Art: Systems

In Fig. 5 some contemporary systems for Computer Mathematics that are especially suited for the formalisation of mathematical proof are presented. On the left in this diagram there are the four 'prehistoric' systems that started the subject in the early seventies (three of those systems are no longer actively being used and have their names in parentheses). These systems differed in the amount of automated help that they gave to their users when doing proofs. At one extreme there was the Automath system of de Bruijn, that had no automation whatsoever: all the details of the proofs had to be provided by the user of the system himself (it is surprising how far one still can get in such a system). At the other extreme there was the nqthm system—also known as the Boyer-Moore prover—which fully automatically tries to prove the lemmas that the user of the system puts to it. In between these two extremes there was the LCF system, which implemented an interesting compromise. The user of this system was in control of the proof, but could make use of the automation of so-called *tactics* which tried to do part of the proof automatically. As will be apparent from this diagram the LCF system was quite influential.

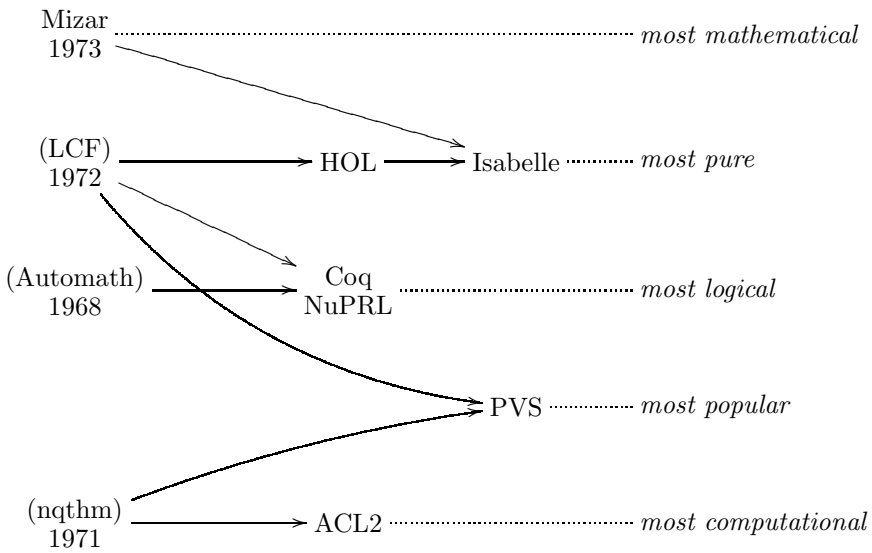


Figure 5. Some systems for Computer Mathematics

The seven systems in this diagram are those contemporary Proof Assistants in which a significant body of mathematics has been formalised. To give some impression of the 'flavour' of those systems we have put a superficial characterisation in the right margin. See ⟨`http://www.cs.ru.nl/~freek/digimath/`⟩ for the web-addresses with information on these systems. The ontologies on which these systems

are based are stated in Fig. 6. All of these systems (with the exception of Automath and Mizar) were primarily motivated by computer science applications. Being able to prove algorithms and systems correct is at the moment the main driving force for the development of Proof Assistants. This is an extremely exciting area of research. Currently, people who have experience with programming claim to 'know' that serious programs without bugs are impossible. However, we think that eventually the technology of Computer Mathematics will evolve into a methodology that will change this perception. Then a bug free program will be as normal as a 'bug free' formalised proof is for us who do formal mathematics.

Systems	Basis
Mizar	Set Theory
Coq, NuPRL	Intuitionistic Type Theory
HOL, Isabelle	Higher Order Logic
PVS	Higher Order Logic with predicate subtypes
ACL2	Primitive Recursive Arithmetic

Figure 6. Foundational bases for Systems of Computer Mathematics

When one starts applying the technique to mathematics, one may be struck when finishing a formalisation. Usually one needs to go over a proof when it is finished, to make sure one really has understood everything and made no mistakes. But with a formalisation that phase is not needed anymore. One can even finish a proof before one has fully understood it! The feeling in that case is not unlike trying to take another step on a staircase which turns out not to be there.

On the other hand when one returns from formalisation to 'normal' programming, it feels as if a safety net has been removed. One can then write down incorrect things again, without it being noticed by the system!

Theorem	System
Hahn-Banach Theorem	Mizar, ALF†, Isabelle
Law of Quadratic Reciprocity	nqthm, Isabelle
Gödel's First Incompleteness Theorem	nqthm, Coq
Correctness of Buchberger's Algorithm	ACL2, Agda†, Coq
Fundamental Theorem of Galois theory	Lego†
Fundamental Theorem of Calculus	*many systems*
Fundamental Theorem of Algebra	Mizar, HOL, Coq
Bertrand's Postulate	HOL, Coq
Prime Number Theorem	Isabelle
Four Colour Theorem	Coq
Jordan Curve Theorem	HOL
Textbook on Continuous Lattices	Mizar

Figure 7. Formalised mathematics

A currently less successful application of Proof Assistants, but one which in the long run will turn out to be even more important than verification in computer science, is the application of Proof Assistants to mathematics. The QED manifesto, see

† The ALF and Lego systems are Proof Assistants from the Automath/Coq/NuPRL tradition that are no longer in use. Agda is the successor of ALF: it is related to Automath but not to LCF.

Boyer et al. [1994], gives a lucid description of how this might develop. We believe that when later generations look back at the development of mathematics one will recognise four important steps: (1) the Egyptian-Babylonian-Chinese phase, in which correct computations were made, without proofs; (2) the ancient Greeks with the development of 'proof'; (3) the end of the nineteenth century when mathematics became 'rigorous'; (4) the present, when mathematics (supported by computer) finally becomes fully precise and fully transparent.

To show what current technology is able to do, we list some theorems that have been formalised already in Fig. 7. Clearly the technology has not yet reached 'the research frontier', but the theorems that can be formalised are not exactly trivial either.

The formalisations that are listed in this table are much like computer programs. To give an indication of the size of these formalisations: the Isabelle formalisation of the Prime Number Theorem by Avigad and others consists of 44 files that together take 998 kilobytes in almost thirty thousand lines of 'code'.

What is needed?

Today no mathematician uses a Proof Assistant for checking or developing new work. We believe that in the coming decennia this will change (although we do not know exactly how long it will take). We now will list some properties that a system for Computer Mathematics should have before this will happen.

Mathematical style

In the current proof assistants the mathematics does not resemble traditional mathematics very much. This holds both for the statements as well as for the proofs. As an example consider the following statement:

$$\lim_{x \to x_0} f(x) + g(x) = \lim_{x \to x_0} f(x) + \lim_{x \to x_0} g(x)$$

In the HOL system this statement is called `LIM_ADD`, and there it reads†

```
!f g l m. (f --> l)(x0) /\ (g --> m)(x0) ==>
   ((\x. f(x) + g(x)) --> (l + m))(x0)
```

This does not match the LaTeX version of the statement. (The technical reason for this is that as HOL does not support partial functions, the limit operator is represented as a relation instead of as a function.)

In the Mizar library the statement is called `LIMFUNC3:37`, and there it reads (where for clarity we replaced the condition of the statement, which states that the limits actually exist, by an ellipsis):

```
... implies
lim(f + g, x0) = lim(f, x0) + lim(g, x0)
```

Again this does not resemble the informal version of the statement. (Here the reason is that Mizar does not support binders, and therefore the limit operator cannot bind

† Here '`!`' stands for '$\forall$' and '`\`' stands for 'λ'.

the limit variable. Therefore the functions f and g have to be added instead of the function values $f(x)$ and $g(x)$.)

Clearly unless a system can accept this statement written similar to

```
... ==> lim(x->x0, f(x) + g(x)) =
        lim(x->x0, f(x)) + lim(x->x0, g(x))
```

mathematicians will not be very much inclined to use it†.

While in most current systems the statements themselves do not look much like their informal counterparts, for the proofs it is even worse. The main exceptions to this are the Mizar language, and the Isar language for the Isabelle system. We call these two proof languages *mathematical modes*. As an example, the following is what a proof looks like in the actual Coq system‡.

```
Lemma limit_plus :
 forall (f g:R -> R) (D:R -> Prop) (l l' x0:R),
   limit1_in f D l x0 ->
   limit1_in g D l' x0 -> limit1_in (fun x:R => f x + g x) D (l + l') x0.
Proof.
intros; unfold limit1_in; unfold limit_in; simpl;
 intros; elim (H (eps * / 2) (eps2_Rgt_R0 eps H1));
 elim (H0 (eps * / 2) (eps2_Rgt_R0 eps H1)); simpl;
 clear H H0; intros; elim H; elim H0; clear H H0; intros;
 split with (Rmin x1 x); split.
exact (Rmin_Rgt_r x1 x 0 (conj H H2)).
intros; elim H4; clear H4; intros;
 cut (R_dist (f x2) l + R_dist (g x2) l' < eps).
 cut (R_dist (f x2 + g x2) (l + l') <= R_dist (f x2) l + R_dist (g x2) l').
exact (Rle_lt_trans _ _ _).
exact (R_dist_plus _ _ _ _).
elim (Rmin_Rgt_l x1 x (R_dist x2 x0) H5); clear H5; intros.
generalize (H3 x2 (conj H4 H6)); generalize (H0 x2 (conj H4 H5)); intros;
 replace eps with (eps * / 2 + eps * / 2).
exact (Rplus_lt_compat _ _ _ _ H7 H8).
exact (eps2 eps).
Qed.
```

Not even a Coq specialist will be able to understand what is going on in this proof without studying it closely with the aid of a computer. It will be clear why we think that having a mathematical mode is essential for a system to be attractive to working mathematicians.

As an example of what a mathematical mode looks like, in Fig. 8 there is the Coq proof rephrased using the Mizar proof language¶.

† Structurally this last version is what one would like to write, but typographically it still is not ideal. Perhaps it will be one day be possible to use the mathematical style that opens this section – or, at least, the LaTeX source for it.

‡ In this proof '`limit_in1 f D l x0`' is the Coq notation for $\lim_{x \to x_0} f(x) = l$ where x ranges over the set $D \subseteq \mathbb{R}$.

¶ Actually, this is a mixture of Mizar and Coq. The proof language is Mizar, but the statements are written in Coq syntax. We do this to be able to follow the Coq script very closely.

```
proof
  let f,g be R -> R and D be R -> Prop and l,l',x0 be R such that
A1: limit1_in f D l x0 and
A2: limit1_in g D l' x0;
  now let eps be R such that
A3:   eps > 0;
    consider del being R such that
A4:   del > 0 and
A5:   forall x : R, D x /\ R_dist x x0 < del -> R_dist (f x) l < eps * / 2
        by A1,A3;
    consider del' being R such that
A6:   del' > 0 and
A7:   forall x : R, D x /\ R_dist x x0 < del' -> R_dist (g x) l' < eps * / 2
        by A2,A3;
    take del'' = Rmin del del';
    thus del'' > 0 by Rmin_Rgt_r,A4,A6;
    let x be R;
    assume that
A8:   D x and
A9:   R_dist x x0 < del'';
    thus R_dist (f x + g x) (l + l') < eps
    proof
A10:   R_dist (f x + g x) (l + l') <= R_dist (f x) l + R_dist (g x) l'
        by R_dist_plus;
A11:  del > R_dist x x0 by Rmin_Rgt_l,A9;
A12:  del' > R_dist x x0 by Rmin_Rgt_l,A9;
A13:  R_dist (f x) l < eps * / 2 by A5,A8,A11;
A14:  R_dist (g x) l' < eps * / 2 by A7,A8,A12;
      R_dist (f x) l + R_dist (g x) l' < eps * / 2 + eps * / 2
        by Rplus_lt_compat,A13,A14;
      then R_dist (f x) l + R_dist (g x) l' < eps by eps2;
      hence thesis by Rle_lt_trans,A10;
    end;
  end;
  then limit_in R_met R_met (fun x : R => f x + g x) D x (l + l');
  hence limit1_in (fun x:R => f x + g x) D (l + l') x0;
end;
```

Figure 8. A proof in *mathematical mode*

Library

The most important part of a proof assistant is its library of pre-proved lemmas. If one looks which systems are useful for doing formal mathematics, then those are exactly the systems with a good library. Using an average system with a good library is painful but doable. Using an excellent system without a library is not. The bigger the library, the more mathematics one can deal with in a reasonable time.

As an example, in Nijmegen we formalised a proof of the Fundamental Theorem of Algebra (see Geuvers et al. [2001]) and it took a team of three people two years. At the same time Harrison formalised the same theorem all by himself (as described

in Harrison [2001]) and it only took him a few days. The main difference which explains this huge difference in effort needed, is that he already had an extensive library while in Nijmegen we had not†.

Decision procedures

One might imagine that the computer can help mathematicians find proofs. However *automated theorem proving* is surprisingly weak when it comes to finding proofs that are interesting to human mathematics. Worse, if one takes an existing informal textbook proof, and considers the gaps between the steps in that proof as 'proof obligations', then a general purpose theorem prover often will not even be able to find proofs for those. For this reason Shankar, whose group is developing PVS, emphasised that rather than the use of general automated theorem provers the *decision procedures*, which can only solve one very specific task, are important as they will always be able to solve problems in a short time. In fact Shankar claims that the big success of PVS is mainly due to the fact that it has the best decision procedures of all the systems, and combines those well.

Our view on automating Computer Mathematics is that a proof is something like an iceberg. When considering all details of the proof, a human mathematician will not even be consciously aware of the majority of those, just like an iceberg is 90% under water. What is written in papers and communicated in lectures is only the 10% of the proof (or even less) which is present in the consciousness of the mathematician. We think that the automation of a system for Computer Mathematics should provide exactly those unconscious steps in the proof. (There is a risk of having the computer provide too many steps so that we will not understand anymore what it is doing, and then we will not be able to guide the proof any longer.)

One should make a distinction between unconscious steps and decidable ones. Some unconscious steps may be guided in undecidable areas by heuristic tactics. Also, some decision procedures have horrendous complexity, so it is not necessarily the case that they will 'solve problems in a short time'. However, the point we are making here is that the main function of automation in proof assistants should be taking care of unconscious steps, and that decision procedures are an important part of that.

Support for reasoning with gaps

The manner in which proof assistants are generally being used today is that the whole formalisation is completed all the way to the axioms of the system. This is for a good reason: it turns out that it is very difficult to write down fully correct formal statements without having the computer help 'debug' the statements by requiring to formalise the proofs. If one starts a formalisation by first writing down a global sketch of a theory, then when filling in the actual formal proofs, it often turns out that some of those statements are not provable after all!

However, if one just wants to use a Proof Assistant to order one's thoughts, or to communicate something to another mathematician, then fully working out all

† Another difference was that in Nijmegen we formalised an intuitionistic proof, while Harrison formalised a classical proof. But when analysing the formalisations, it turned out that this was *not* the main reason for the difference in work needed.

proofs is just not practical. In that case one would like to just give a *sketch* of the proof inside the formal system, as described in Wiedijk [2004]. Related to this is the technique called *proof planning*, see for instance Bundy [1991]. Still, the current Proof Assistants do not support this way of working very well.

In Lamport [1995] a proof style is described in which proofs are incrementally developed by *refining* steps in the proof into more detailed steps. Although that paper does not talk about proofs in the computer, and although we are not sure that the specific proof display format that is advocated in that paper is optimal, it is clear that this *style of working* should be supported by systems for Computer Mathematics, if they are to be acceptable to mathematicians.

5. Romantic vs. Cool Mathematics

After the initial proposals of the possibility of computer mathematics many mathematicians protested on emotional grounds. "Proofs should be survey-able in our mind", was and still is an often heard objection. We call this the *romantic* attitude towards mathematics. There is another style, *cool* mathematics, that is, verified by a computer. The situation may be compared to that in biology. In romantic biology, based on the human eye, one is concerned with flowers and butterflies. In cool biology, based on the microscope, an extension of our eyes, one is concerned with cells. There is even *super-cool* molecular biology, based on electron microscopes. By now we know very well that these latter forms of biology are vital and essential and have a romanticism of their own. Similarly, we expect that cool proofs in mathematics will eventually lead to romantic proofs based on these. In comparison with biology there is also super-cool mathematics, checked by a computer, with a program this time not checked by the human mind, but checked by a computer in the cool way. This kind of *boot-strap* has been used for a compiled (hence faster) version of Coq, see Barras [1996] and Barras [1999].

A fully formalised proof in Coq of the Four Colour Theorem has been verified, see Gonthier [2004]. Moreover a full proof in HOL of the Jordan Curve Theorem has been produced by Tom Hales as part of his work towards a full formalisation of his proof of the Kepler conjecture. Both informal proofs need a long computer verification. These kinds of theorems with a long proof seem exceptional, but they are not. From the undecidability of provability it follows trivially that there will be relatively short statements with arbitrarily long proofs†.

We foresee that in the future *cool* proofs will have *romantic* consequences and moreover that Computer Mathematics will have viable applications.

References

Ackermann, W. [1928]. Zum Hilbertschen Aufbau der reellen Zahlen, *Mathematische Annalen* **99**, pp. 118–133.

Aczel, P. and M. Rathjen [2001]. Notes on constructive set theory, *Technical report*, Institut Mittag-Leffler. URL: ⟨`http://www.ml.kva.se/preprints/meta/AczelMon_Sep_24_09_16_56.rdf.html`⟩.

† Indeed if every theorem of length n would have a proof of length 2^{2^n}, then theorem-hood would be decidable by checking all the possible candidate proofs.

Appel, K. and W. Haken [1977a]. Every planar map is four colorable. part I. Discharging, *Illinois J. Math.* **21**, pp. 429–490.

Appel, K. and W. Haken [1977b]. Every planar map is four colorable. part II. Reducibility, *Illinois J. Math.* **21**, pp. 491–567.

Aschbacher, M. [2004]. The Status of the Classification of the Finite Simple Groups, *Mathematical Monthly* **51**(7), pp. 736–740.

Bachmair, Leo and Harald Ganzinger [1994]. Buchberger's algorithm: a constraint-based completion procedure, *Constraints in computational logics (Munich, 1994)*, Lecture Notes in Comput. Sci. 845, Springer, Berlin, pp. 285–301.

Barendregt, H. [2005]. Foundations of Mathematics from the Perspective of Computer Verification, In: *Mathematics, Computer Science, Logic - A Never Ending Story*, Springer Verlag. ⟨`http://www.cs.ru.nl/~henk/papers.html`⟩.

Barendregt, Henk and Herman Geuvers [2001]. Proof-assistants Using Dependent Type Systems, *in:* Alan Robinson and Andrei Voronkov (eds.), *Handbook of Automated Reasoning*, Elsevier Science Publishers B.V., pp. 1149–1238.

Barras, B. [1996]. Verification of the interface of a small proof system in coq, *in:* E. Gimenez and C. Paulin-Mohring (eds.), *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, Springer-Verlag LNCS 1512, Aussois, France, pp. 28–45.

Barras, B. [1999]. *Auto-validation d'un système de preuves avec familles inductives*, Thèse de doctorat, Université Paris 7.

Boyer, R. et al. [1994]. The QED Manifesto, *in:* A. Bundy (ed.), *Automated Deduction – CADE 12*, LNAI 814, Springer-Verlag, pp. 238–251. ⟨`http://www.cs.ru.nl/~freek/qed/qed.ps.gz`⟩.

Buchberger, B. [1965]. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ring*, Dissertation, University of Innsbruck.

Buchberger, B. and F. Winkler [1998]. *Gröbner Bases and Applications*, Cambridge University Press.

Bundy, Alan [1991]. A science of reasoning, *in:* J.-L. Lassez and G. Plotkin (eds.), *Computational Logic: Essays in Honor of Alan Robinson*, MIT Press, pp. 178–198. Also available from Edinburgh as DAI Research Paper 445.

Church, A. [1936]. An unsolvable problem of elementary number theory, *American Journal of Mathematics* **58**, pp. 345–363.

Collins, G. E. [1975]. Quantifier elimination for real closed fields by cylindrical algebraic decomposition, *Automata theory and formal languages (Second GI Conf., Kaiserslautern, 1975)*, Springer, Berlin, pp. 134–183. Lecture Notes in Comput. Sci., Vol. 33.

Davis, Martin [1973]. Hilbert's tenth problem is unsolvable, *Amer. Math. Monthly* **80**, pp. 233–269.

Feferman, S. [1998]. *In the Light of Logic*, Oxford University Press, Oxford.

Floyd, Robert W. [1967]. Assigning meanings to programs, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, American Mathematical Society*, pp. 19–32.

Frege, Gottlob [1971]. *Begriffsschrift und andere Aufsätze*, Georg Olms Verlag, Hildesheim. Zweite Auflage. Mit E. Husserls und H. Scholz' Anmerkungen herausgegeben von Ignacio Angelelli, Nachdruck.

Gentzen, G. [1969]. *The collected papers of Gerhard Gentzen*, Edited by M. E. Szabo. Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Co., Amsterdam.

Geuvers, Herman, Freek Wiedijk and Jan Zwanenburg [2001]. A constructive proof of the Fundamental Theorem of Algebra without using the rationals, *in:* Paul Callaghan, Zhaohui Luo, James McKinna and Robert Pollack (eds.), *Types for Proofs and Programs, Proceedings of the International Workshop TYPES 2000*, LNCS 2277, Springer, pp. 96–111.

Gödel, K. [1930]. Die Vollständigkeit der Axiome des logischen Funktionalkalküls, *Monatshefte für Mathematik und Physik* **37**, pp. 349–360.

Gödel, K. [1931]. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, *Monatshefte für Mathematik und Physik* **38**, pp. 173–198. Translated and commented in Gödel [1986]. Another English version based on course notes by Kleene and Rosser is in Gödel [1965].

Gödel, K. [1995]. *Collected works III: Unpublished essays and lectures (S. Feferman et al., editors)*, Oxford University Press.

Gödel, Kurt [1965]. On undecidable propositions of formal mathematical systems, *in:* Martin Davis (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven Press, New York, pp. 41–74. From mimeographed notes on lectures given by Gödel in 1934.

Gödel, Kurt [1986]. *Collected works. Vol. I*, The Clarendon Press Oxford University Press, New York. Publications 1929–1936, Edited and with a preface by Solomon Feferman.

Gonthier, George [2004]. The Four Color Theorem in Coq, Talk at the TYPES 2004 conference, December 15–18, 2004, Campus Thalès, Jouy-en-Josas, France.

Hales, T. [to appear]. A Proof of the Kepler Conjecture, *Annals of Mathematics*. URL: ⟨`http://www.math.pitt.edu/~thales/kepler03/fullkepler.pdf`⟩.

Harrison, John [2001]. Complex quantifier elimination in HOL, *in:* Richard J. Boulton and Paul B. Jackson (eds.), *TPHOLs 2001: Supplemental Proceedings*, Division of Informatics, University of Edinburgh, pp. 159–174. Published as Informatics Report Series EDI-INF-RR-0046. URL: ⟨`http://www.informatics.ed.ac.uk/publications/report/0046.html`⟩.

Heyting, A. [1930]. Die formalen Regeln der intuitionistischen Logik, *Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse* pp. 42–56.

Hilbert, D. [1926]. Uber das Unendliche, *Mathematische Annalen* **95**, pp. 161–190.

Hoare, C.A.R. [1969]. An axiomatic basis for computer programming, *The Communication of ACM* **12**, pp. 576–583.

Horgan, John [1993]. The death of proof, *Sci. Amer.* **269**(4), pp. 92–103.

Husserl, E. [1901]. *Untersuchungen zur Phänomenologie und Theorie der Erkenntnis*, Max Niemeyer, Halle.

Kempe, A.B. [1879]. On the geographical problem of the four colors, *Amer. J. Math.* **2**, pp. 193–200.

Kleene, S. C. [1936]. Lambda-definability and recursiveness, *Duke Mathematical Journal* **2**, pp. 340–353.

Kunen, Kenneth [1983]. *Set theory*, Studies in Logic and the Foundations of Mathematics 102, North-Holland Publishing Co., Amsterdam. An introduction to independence proofs, Reprint of the 1980 original.

Lamport, Leslie [1995]. How to Write a Proof, *American Mathematical Monthly* **102**(7), pp. 600–608.

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

Miller, G. [1976]. Riemann's Hypothesis and Tests for Primality, *J. Comp. Syst. Sci.* **13**, pp. 300–317.

Moerdijk, I. and E. Palmgren [2002]. Type Theories, Toposes and Constructive Set Theory: Predicative Aspects of AST, *Annals of Pure and Applied Logic* **114**, pp. 155–201.

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

Rabin, M.O. [1980]. Probabilistic Algorithm for Testing Primality, *J. Number Th.* **12**, pp. 128–138.

Robertson, N., D.P. Sanders, P. Seymour and R. Thomas [1996]. A new proof of the four-colour theorem, *Electron. Res. Announc. Amer. Math. Soc.* **2**, pp. 17–25. URL: ⟨`http://www.ams.org/era/1996-02-01/S1079-6762-96-00003-0/home.html`⟩.

Schönfinkel, M. [1924]. Über die Bausteine der Mathematische Logik, *Math. Annalen* **92**, pp. 305–316.

Scott, D. [1970]. Constructive validity, *Symposium on Automatic Demonstration (Versailles, 1968)*, Lecture Notes in Mathematics, Vol. 125, Springer, Berlin, pp. 237–275.

Skolem, T. [1922]. Über ganzzahlige Lösungen einer Klasse unbestimmter Gleichungen, *Norsk Matematisk Forenings skrifter*.

Sudan, G. [1927]. Sur le nombre transfini ω^ω, *Bulletin mathématique de la Société Roumaine des Sciences* **30**, pp. 11–30.

Tarski, A. [1951]. *Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley.

Terese (ed.) [2003]. *Term Rewrite Systems*, Cambridge University Press.

Turing, A.M. [1936]. On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society, Series 2* **42**, pp. 230–265.

Turing, A.M. [1949]. Checking a Large Routine, *Report of a Conference on High Speed Automatic Calculating machines*, pp. 67–69. Paper for the EDSAC Inaugural Conference, 24 June 1949.

Veltman, M. [1967]. SCHOONSCHIP, A CDC 6600 program for Symbolic Evaluation of Algebraic Expressions, *Technical report*, CERN.

Wang, Hoa [1997]. *A Logical Journey, From Gödel to Philosophy*, Bradford Books.

Wiedijk, F. [2000]. The De Bruijn Factor, ⟨`http://www.cs.ru.nl/~freek/notes/factor.ps.gz`⟩.

Wiedijk, F. [2004]. Formal Proof Sketches, *in:* Stefano Berardi, Mario Coppo and Ferruccio Damiani (eds.), *Types for Proofs and Programs: Third International Workshop, TYPES 2003, Torino, Italy, April 30 – May 4, 2003, Revised Selected Papers*, LNCS 3085, pp. 378–393.