Math. Struct. in Comp. Science (2009), *vol.* 19, *pp.* 877–896. © Cambridge University Press 2009
doi:10.1017/S0960129509990041 First published online 7 September 2009

Social processes, program verification and all that

ANDREA ASPERTI[†], HERMAN GEUVERS[‡] and
RAJA NATARAJAN[§]

[†]*Dept. of Comp. Sci., Univ. of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy*
Email: `asperti@cs.unibo.it`
[‡]*Dept. of Comp. Sci., Radboud Univ. Nijmegen and Tech. Univ. Eindhoven, The Netherlands*
Email: `herman@cs.ru.nl`
[§]*School of Tech. and Comp. Sci., Tata Institute of Fundamental Research, Mumbai 400 005, India*
Email: `raja@tifr.res.in`

Received 25 April 2009; revised 3 May 2009

In a controversial paper (De Millo *et al.* 1979) at the end of the 1970's, R. A. De Millo, R. J. Lipton and A. J. Perlis argued against formal verifications of programs, mostly motivating their position by an analogy with proofs in mathematics, and, in particular, with the impracticality of a strictly formalist approach to this discipline. The recent, impressive achievements in the field of interactive theorem proving provide an interesting ground for a critical revisiting of their theses. We believe that the social nature of proof and program development is uncontroversial and ineluctable, but formal verification is not antithetical to it. Formal verification should strive not only to cope with, but to ease and enhance the collaborative, organic nature of this process, eventually helping us to master the growing complexity of scientific knowledge.

1. Introduction

Heavier than air flying machines are impossible.
S. P. Langley (Langley 1891)

Formal verification of programs, no matter how obtained, will not play the same role in the development of computer science and software engineering as proofs do in mathematics.
R. A. De Millo, R. J. Lipton and A. J. Perlis (De Millo *et al.* 1979)

Samuel Pierpont Langley was a professor of astronomy and physics, and a world-expert in aerodynamics during the late nineteenth and early twentieth century. The esteem with which he is held can be seen from the fact that one of NASA's research centres was named after Langley. At the height of his research career, Samuel Langley published a result (Langley 1891), which came to be known as 'Langley's Law'. According to this erroneous law – *the higher the speed, the lower the drag* – more power was required in order to make an aircraft fly slower, and indeed if this were true, heavier than air flying machines would certainly have been an impossibility. Fortunately, the Wright Brothers had not read Langley's book, and they went on to develop the first manned aircrafts that could be controlled in-flight from the aircraft itself.

In a famous, influential paper at the end of 1970's, R. A. De Millo, R. J. Lipton and A. J. Perlis (De Millo *et al.* 1979) advanced various criticisms of the very idea of the formal verification of programs. The impressive advances in this area seem by themselves to belie their gloomy predictions. Formal verification has currently reached such a level of maturity as to allow correctness proofs of sophisticated hardware components (Harrison 2007), complex programs such as optimising compilers (see, for example, Leroy (2006) and Tristan and Leroy (2008)), and parts of modern operating systems (see, for example, Alkassar *et al.* (2009) and Klein (2009)).

It is precisely in view of these achievements, however, that we can look back at De Millo *et al.* (1979) with a less passionate and more objective spirit, making a more stringent analysis of their thesis and arguments, without focusing on the polemic frame intentionally chosen by its authors, *viz.*, in Lamport's words[†], *as a debate between a reasonable engineering approach that completely ignores verification and a completely unrealistic view of verification advocated only by its most naive proponents.*

In fact, some of the thesis advocated by De Millo, Lipton and Perlis is sharable, very pertinent and still relevant; on the other hand, most of their arguments, following thirty years of research, sound obsolete and a bit trite, and are asking for a critical reappraisal. Here is a quick summary of our critique, before we lay it out in detail:

— De Millo *et al.* state (quoting the logician Rosser) that *...intuition is the 'final authority*:
Intuitions and analogies may help in the explanation and the assimilation of a statement, but when it comes to verification of a statement, proof is the authority. Intuition sometimes just fails.

— De Millo *et al.* state that *...we will continue to argue that programming is like mathematics, and that the same social processes that work in mathematical proofs doom verifications*:
We argue that mathematics will become more and more like programming and that the future of both mathematics and programming lies in the fruitful *combination* of formal verification and the usual social processes that are already working in both scientific disciplines.

2. Proofs and programs

Russell did succeed in showing that ordinary working proofs can be reduced to formal, symbolic deductions. But he failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic. He showed what can be done in principle and what cannot be done in practice.
R. A. De Millo, R. J. Lipton and A. J. Perlis (De Millo *et al.* 1979)

In drawing a parallel between program verification and proofs of theorems, the key arguments adduced by De Millo *et al.* against the formal approach is the essential impracticality of a strictly logistic approach to mathematics due to the *nearly inconceivable length of a deduction from 'first principles.* The argument is repeated several times: the quotation above is from their paper (page 272); on the next page they go on to say

A formal demonstration of one of Ramanujan's conjectures assuming set theory and elementary analysis would take about two thousand pages;

[†] See `http://research.microsoft.com/en-us/um/people/lamport`.

and, on page 275, Poincaré is quoted in support of their claim,

If it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?

Of course, the argument is not new; even Bourbaki, who is traditionally enlisted in the ranks of the formalist school (Lee 2002), found the project of formalising mathematics *absolutely un-realisable* (Bourbaki 1968):

the tiniest proof at the beginning of the Theory of Sets would already require several hundreds of signs for its complete formalisation.

The argument is reminiscent of the general disbelief at the beginning of the fifties concerning the potentialities of computers and the possibility of writing long programs, which was just due to the inability of conceiving of high-level languages and a process of automatic translation to a machine-understandable code. When Grace Hopper wrote the first compiler in 1952, opening the way to software development, nobody seemed interested: 'I had a running compiler and nobody would touch it', she said. 'They told me computers could only do arithmetic.' (Schieber 1987). The analogy with compilation has already been made by Maurer back in 1979 (Maurer 1979):

We can make an analogy here with compiling a higher level language program into a machine language. Originally this was done by hand […], then compilers came along and started to do the job automatically. […] nobody is ever going to read the object code produced by a compiler; one simply trusts the compiler. What we hope for in verifiers is that we will at least be able to trust them to show program correctness.

Harrison (Harrison 2008) has recently restated the concept in the following terms:

the arrival of the computer changes the situation dramatically. […] checking conformance to formal rules is one of the things computers are very good at. […] the Bourbaki claim that the transition to a completely formal text is routine seems almost an open invitation to give the task to computers.

In fact, automation of formal reasoning has recently gone far beyond *the elementary facts of arithmetic*, permitting the formalisation and automatic verification of complex results such as the asymptotic distribution of prime numbers (Avigad *et al.* 2007), the four colour theorem (Gonthier 2007; Gonthier 2008) and the Jordan curve theorem (Hales 2007). All these developments are significant in size (spanning from 30 to 75 thousand lines of code), but their complexity is still negligible when compared with, say, the size of a modern operating system.

The formal proof of the Jordan curve theorem is due to Thomas Hales, a famous mathematician who is particularly known for his proof of the Kepler conjecture (the most compact way of packing congruent spheres in three dimensional Euclidean space), and for the events related to its publication. Briefly, his proof of the Kepler conjecture involved a large amount of computer verification, and after three years of work, the reviewers of the *Annals of Mathematics* concluded that, although they believed the proof was correct, they were unable to check it thoroughly due to many 'low-level components' that lacked a more general intuition, especially given the degree of computation involved. In the end, the *Annals of Mathematics* published a short version of the proof (Hales 2005), and made the code/data for the proof available un-reviewed on its website. A revised, full version of the article was finally published by *Discrete and Computational Geometry*. Since then, Thomas Hales has started a new project, called Flyspeck (see Hales (2008)), to check the correctness of his proof formally with the help of interactive theorem provers.

The Kepler conjecture and the four colour theorem are good examples of a large number of mathematical proofs based on a direct and substantial use of a computer. Other examples mentioned by Hales in Hales (2008) are the non-existence of a projective plane of order 10, the proof that the Lorentz equations have a strange attractor, the double-bubble problem for minimising soap bubbles enclosing two equal volumes, the optimality of the Leech lattice among 24-dimensional lattice packings, and hyperbolic 3-manifolds. In all these cases, the computer is used to manage the complexity, usually by automatically checking a finite, albeit large number of 'atomic' configurations (a kind of computation that would not be possible, or extremely laborious for a human). Mathematics is entering a new era of results requiring proofs of a complexity and dimension that defy human comprehension, leaving a miasma of doubt about their effective correctness. For instance, the theorem of classification of finite groups is the result of the collective work of about a hundred authors, composed of over 10000 pages of results, spread across 500 journal articles. One of the key results, the Feit–Thomson (or odd-order) theorem (Feit and Thompson 1963) itself takes 255 pages. A formalisation of the odd-order theorem has recently been started by the INRIA–Microsoft research project on 'mathematical components' lead by G. Gonthier (Gonthier *et al.* 2007). As another example, the preprint of F. Almgren's masterpiece in geometric measure theory, familiarly referred to as the 'Big Paper', is 1728 pages long.

Perhaps for the first time in the history of their discipline, mathematicians are now forced to accept the simple fact that many theorems, even if admitting simple and elegant statements, may not admit equally simple and elegant proofs[†]. As observed in Geuvers (2009), it can be formally proved that, in any given logical system, there is no upper bound to the relationship between the size of a statement and the size of its shortest proof, and there is no reason to believe that things should be better if we restrict our attention to 'interesting' theorems. Even De Millo *et al.* themselves admit that this is the case.

For even the most trivial mathematical theories, there are simple statements whose formal demonstrations would be impossibly long.

But they do not want to accept the consequence that a computer may be needed to help the human in verifying the formal demonstrations.

Not always can one find 'a truly marvelous demonstration' that, alas, is just a bit too long to 'fit in the margin of a book': a proof can just be so complicated, no matter what kind of rethinking you might try. Does this mean that such a proof just has to be dismissed and the validity of the statement rejected, possibly renouncing *any* proof, all sacrificed on the mystic altar of elegance and simplicity? May it not possibly mean that we just have to look for the right tools to help us cope with its complexity, and that we have to learn to appreciate a different and less archaic kind of beauty?

[†] According to Lakatos (Lakatos 1976), simplicity is the eighteenth-century idea of mathematical rigour.

Mathematics	*Programming*
theorem	program
proof	verification

Table 1. *The verifier's analogy, according to De Millo, Lipton and Perlis.*

Mathematics	*Programming*
theorem	specification
proof	program
imaginary formal proof	verification

Table 2. *De Millo, Lipton and Perlis' analogy.*

3. Theorems, proofs and specifications

By far the most common way in which we deal with something new is by trying to relate the novelty to what is familiar from past experience: we think in terms of analogies and metaphors. (Even the 5th Edition of the Concise Oxford Dictionary still defines a typewriter as a 'machine for printing characters on paper as substitute for handwriting'!) As long as history evolves along smooth lines, we get away with that technique, but that technique breaks down whenever we are suddenly faced with something so radically different from what we have experienced before that all analogies, being intrinsically too shallow, are more confusing than helpful.

Edsger Dijkstra (Dijkstra 1986)

De Millo, Lipton and Perlis describe the 'verifier's analogy' between mathematics and programming (see Table 1), and they contrast it with their own analogy (Table 2). Their observation is that the verifiers are mistaken by wanting to identify the notion of 'proof' (from mathematics) with 'formal verification' in computer science.

De Millo, Lipton and Perlis do not give any source for the analogy attributed to 'verifier's'; in fact, it is hard to imagine that anybody working in the area of program verification would feel at ease with such an analogy. If somebody working on formal methods really put it forward, their intent was probably to emphasise a simple but crucial fact, which, at the end of seventies was still hard to grasp, namely that programs themselves could become, like mathematical theorems, the object of a scientific investigation.

In fact, the analogy that De Millo, Lipton and Perlis are so happy with, to the extent that they believe they invented it themselves, was precisely the leading theme that, in a slightly more sophisticated form, was at that very time driving research in the field of computer-assisted verification. The 'formal' counterpart of the analogy is called the Curry–Howard correspondence (Howard 1980), and it just differs from the description in Table 2 by the substitution of 'type' for 'specification': a simple twist, which by itself opens up an entirely new universe of possibilities[†]. The analogy can then be made more precise (see Table 3). Proof verification is nothing other than type-checking, and, furthermore,

[†] Howard's paper was printed in 1980, but the first draft was circulated in 1969. As observed by Howard himself, however, the main ideas should be ascribed to Curry, back in the fifties.

Mathematics	*Programming*
theorem	type
proof	program
correctness verification	type checking
cut elimination	computation

Table 3. *Curry–Howard Correspondence.*

Cut Elimination

A cut is a particular logical rule that permits the factorization of a complex reasoning step into a sequence of simpler steps. In order to prove A, we may temporarily assume B, provided we prove it later. In principle, one expects to be able to avoid the use of this rule entirely by simply unfolding the proof of B inside the original proof of A. In practice, the cut-elimination proof, also known as Gentzen's Hauptsatz, is far from trivial, and is not satisfied by all logical systems; but when it holds, it is rich in consequences:

Consistency: It is usually easy to verify that a system does not admit *cut-free* proofs of the absurd. In such a case, if the system enjoys the cut elimination property, it is immediately consistent.

Subformula property: This is an important property in several approaches to proof-theoretic semantics and automated theorem proving. In essence, it says that in order to prove a given statement A, you only need to use sub-formulae of A.

Fig. 1. Normalisation of proofs.

computation has a proof-theoretical counterpart in the form of cut-elimination, which is a process of normalisation of proofs consisting essentially of removing 'detours' (lemmas) by inlining them (see Figure 1).

The analogy with type checking also helps in clarifying a common misconception about automatic verification. When we type-check a piece of code, a mathematical expression, say, we do not have to compute it in order to check that it is properly typed: type checking is an entirely static operation. Similarly, when we check the proof of a mathematical theorem – the famous Ramanujan conjecture, say – we are not supposed to first normalise the proof to a mere application of axioms from *set theory and elementary analysis*: the proof can make use of any sort of theorems and lemmas, and we check it statically and compositionally.

Starting from their analogy (see Table 2), De Millo, Lipton and Perlis also raise the following *fundamental logical objection to verification*:

Since the requirement for a program is informal and the program is formal, there must be a transition, and the transition itself must necessarily be informal.

This criticism has already been answered by Maurer (Maurer 1979), distinguishing between *program correctness* (the fact that a program meets a specification), and *specification correctness* (the fact that the specification meets the user expectations):

A proof of correctness consists of two steps, one formal, the other informal; and neither of the two is valid without the other one.

The idea of the Curry–Howard correspondence is that it is not only specifications that can be given in a completely formal way, but that, by suitably enriching the system of types, and, in particular, by adding *dependent types*, they can be altogether assimilated to types. A dependent type is a type that depends on a term. A typical example is the type of n-dimensional vectors in some space $\mathscr{A}$, whose type clearly depends on the value n. More generally, given a specification $R(x, y)$ expressing the expected relation between the input $x : A$ and the output $y : B$ of a program (where A and B can be seen as 'traditional' types), we can build the following dependent type $\forall x : A.\{y : B | R(x, y)\}$ and try to check if our program inhabits it. Of course, in order to perform the verification, the programmer may be forced to provide substantial help to the type-checker in the form of suitable type annotations of inner expressions (this is not too far from pre- and post-conditions in an axiomatic setting (Hoare 1969)). Quoting Altenkirch *et al.* (2005):

While conventional type systems allow us to validate our programs with respect to a fixed set of criteria, dependent types are much more flexible, they realise a continuum of precision from the basic assertions we are used to expect from types up to a complete specification of the program's behavior. [...] While the price for formally certified software may be high, it is good to know that we may pay it in instalments and that we are free to decide how far we want to go. Dependent types reduce certification to type checking, hence they provide a means to convince others that the assertions we make about our programs are correct. Dependently typed programs are, by their nature, proof carrying code (Necula and Lee 1996; Hamid et al. 2003).

The Curry–Howard correspondence also opens up a completely innovative perspective on program verification, not consisting of trying to match a program against a specification, but merely consisting of proving that the specification can be inhabited. If the user can supply a *constructive* proof of this fact, then it is possible to automatically extract from the proof its algorithmic content, that is, a program satisfying the specification. Program extraction was exploited for the first time in the Nuprl proof development system (Constable *et al.* 1986). Having proved a property t of the form $\forall x : A, \exists y : B, R(x, y)$, the term `term_of(t)` extracts a function mapping any a of type A into a pair consisting of an element b of type B and a proof p that such a b verifies the property $R(a, b)$. By selecting the first component of this pair, we have a function f from A to B such that $R(x, f(x))$ for all x in A. The extraction technique can also be extended to some extent to classical proofs (Parigot 1992; Barbanera and Berardi 1996).

The Curry–Howard correspondence may also help in gaining an understanding of some of the reasons for the slow recognition of a computer-aided, strictly formal approach in the mathematical community and, conversely, of the moderate interest of computer scientists for its application in this field. The point is that formal proofs, whatever effort you make to write them in a natural, declarative style (see, for example, Wenzel (1999)), still look like programs, and the vast majority of mathematicians dislike programs altogether. On the other side, computer scientists (usually) like them, but all the fun is in creating something *executable* while, of course, you never eliminate cuts from mathematical proofs (Boolos 1984). There is, alas, nothing so irremediably static, somberly boring as a (formal) proof of a mathematical statement, once it has been completed. Still, of course, there is the possibility of *extracting* a program from a (constructive) proof. Unfortunately, you have no chance of extracting a *good* algorithm from a *good* proof, simply because the criteria used to evaluate proofs (elegance, conciseness) and algorithms (performance, above all) are completely different, while the proof and the program realising it are essentially isomorphic. For instance, if you try to prove that any list of objects can be ordered with respect to a given ordering relation, you will most likely end up with a proof corresponding to an insertion algorithm; to extract a quicksort, you have to entirely rethink the proof according to the expected output (which is not methodologically very far removed from first writing the algorithm and then proving its correctness).

4. Proofs and refutations

We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem – and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail.

R. A. De Millo, R. J. Lipton and A. J. Perlis (De Millo *et al.* 1979)

A theorem either can or cannot be derived from a set of axioms. I don't believe that the correctness of a theorem is to be decided by a general election.

Leslie Lamport (Lamport 1979)

There has always been an interesting debate about the actual role of proofs in mathematics. De Millo *et al.* firmly deny any deductive validity to proofs. This position is very common among mathematicians, and had eminent supporters. For instance, G. H. Hardy, who is traditionally credited with reforming British mathematics by bringing rigour into it, described the notion of mathematical proof *as we working mathematicians are familiar with* in the following terms (Hardy 1928):

There is strictly speaking no such thing as a mathematical proof; we can, in the last analysis, do nothing but point; [...] proofs are what Littlewood and I call gas*, rhetorical flourishes designed to affect psychology, pictures on the board in the lecture, devices to stimulate the imagination of pupils.*

This view is traditionally contrasted with the logistic (neopositivist) school, which has been re-invigorated by the recent results in the field of automation of formal reasoning:

The history of mathematics has stories about false results that went undetected for long periods of time. However, it is generally believed that if a published mathematical argument is not valid, it will be eventually detected as such. While the process of finding a proof may require creative insight, the activity of checking a given mathematical argument is an objective activity; mathematical correctness should not be decided by a social process (Coquand 2008).

Harrison (Harrison 2008) explicitly mentions that one of the goals of computer-aided verification should be

supplementing or even partly replacing the process of peer review for mainstream mathematical papers with an objective and mechanizable criterion for the correctness of proofs.

As suggested by MacKenzie (MacKenzie 2005) among others, the two positions can be reconciled if we accept the idea that proof assistants are going to change the 'whole concept of proof', splitting the two roles of *message* and *certificate*:

Ever since Euclid, mathematical proofs have served a dual purpose: certifying that a statement is true, and explaining why it is true. Now those two epistemological functions may be divorced. In the future *the computer assistant may take care of the certification and leave the mathematicians to look for an explanation that humans can understand.*

A proof serves two purposes:

1 to be able to verify the validity of a statement; and
2 to explain – by providing intuitions – why the statement is true.

These roles are traditionally interwoven: a proof contains intuitions, sometimes some remarks on why a certain method does not work, motivating examples, plus a line of reasoning that builds up the precise argument. With the advent of tools like proof assistants, it becomes realistic for us to leave the first incarnation of a proof (the verification) to a computer and the second (the explanation) to the human.

In the optimal scenario, we can maintain a close connection between the two incarnations of proof. Or even better, we would be able to generate automatically a machine checkable certificate from a human readable message. The problem is that, since the translation has to be done automatically, the message itself must already be written in a machine understandable language, and it is extremely difficult to define a 'high-level' language suitable for this kind of human–machine communication.

However, there is another problem. Suppose we have Harrison's tool (and we are still very far from such a goal). Then, of course, the fact that proofs are validated by the tool is not a sufficient reason for accepting tham as a scientific contribution; Harrison does not suggest that, but seems to suggest the converse, which looks equally problematic.

The historical relevance of wrong proofs in the development of mathematics is easily documented (see, for example, Lecat (1939)). To take a recent example, which is well known in the programming language community, without the publication of Martin-Löf's mistaken proof of termination for system U, we would probably never have had Girard's system F, that is, the polymorphic lambda calculus.

False proofs, or proofs becoming refutable under a suitable concept-stretching of the relevant notions, are an essential component of the quasi-empirical nature of mathematics and are extensively discussed in Lakatos (Lakatos 1976):

'Certainty' is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge.

The risk envisaged by Lakatos (though a similar criticism of logical positivism was raised by Popper (Popper 1963)) is to

construct formalised languages in which artificially congealed states of science are expressed. [...] Science teaches us not to respect any given conceptual-linguistic framework lest it should turn into a conceptual prison.

However, Lakatos seems to be more concerned with the declarative, descriptive level of theories and definitions, than with the foundational, logical layer, and a few pages later he observes:

Nineteenth-century mathematical criticism stretched more and more concepts, and shifted the meaning-load of more and more terms onto the logical form *of the propositions and onto the meaning of the few (as yet) unstretched terms. In the 1930's this process seemed to slow down and the demarcation line between unstretchable (logical) terms and stretchable (descriptive) terms seemed to become stable.*

A list, containing a small number of logical terms came to be widely agreed upon, so that a general definition of logical truth became possible. [...] The most interesting result in this direction was Popper (Popper 1948) from which it follows that one cannot give up further logical constants without giving up some basic principles of rational discussion.

The only actual philosophical danger of a strictly formal approach to mathematics – though not to be underestimated – may be to induce the fallacy of deductionism, suggesting that the *path of discovery* is from axioms and/or definitions to proofs and theorems (a risk that Lakatos considers just a bit more dangerous for mathematics, than inductivism). This point has been clearly stated by Paul Halmos in his 'Automathography' (Halmos 1985):

Mathematics is not a deductive science. When you try to [solve a problem] ... what you do is trial and error, experimentation, guesswork. You want to find out what the facts are, and what you do is in that respect similar to what a laboratory technician does, but it is different in its degree of precision and information.

The systematic use of an automatic checker to rule out wrong theorems, simply negating their right to existence and with them any form of refutation (and hence of criticism), would of course be a dramatic step towards a strictly conservative deductionist attitude, negating the possibility and importance of what Lakatos calls naive guessing and concept stretching. However, this is not its intended use. In exactly the same way as during program compilation, the process of type checking is not meant to simply discriminate good programs from bad ones. The type checker is an important driver during the program development phase, and a major tool for the deployment of lightweight, adaptive software methodologies, requiring frequent modifications and refactoring. This interactive use of the type-checker is likely to increase in the near future; the situation is so described in Altenkirch *et al.* (2005):

Programming is a complex task which can be made easier for people to do with the help of computers. The conventional cycle of programming with a text editor then compiling in 'batch mode' is a welcome shortening of the feedback loop since the days of punched cards, but it clearly under utilises the technology available today. Any typed programming language can benefit from the capacity – but not necessarily the compulsion – to invoke the type-checker incrementally on incomplete subprograms whilst they are under development. The more powerful the type system, the more pressing this need becomes – it just gets harder to do in your head, especially when types contain computations, *for which computers are inherently useful.*

It is this interactive use of the computer that is precisely the most exciting prospect in computer-assisted reasoning, and the crucial point where modern interactive proof assistants differ from their first generation precursors. The goal (which is still extremely distant, though clearly identifiable) would be to assist mathematicians not in the act of checking the 'correctness' of their reasoning, but in the *process of mathematical discovery*, that is, during design, analysis and elaboration: for instance, in the investigation of the impact of small, local modifications on the meaning of entities on the logical correctness of the proof. This can be more easily understood by a simple extension of the Curry–Howard correspondence, by distinguishing between raw (pre-typed/untyped) programs and well-typed ones. A mathematical proof, in Lakatos' sense, is a raw program: a *crystal clear thought experiment, or construction.* From this perspective, 'proofs' prove nothing:

Mathematics	*Programming*
proof analysis	type inference/refinement
proof/thought experiment	untyped program

Table 4. *An extended analogy.*

they are just tests, experiments, but not necessarily leading to the expected results[†]. It is the process of proof analysis that adds a deductive layer to the proof, *inferring* proof-generated lemmas and concepts. Proof analysis is then akin to type inference. In the realm of interactive theorem provers, the module in charge of type inference, synthesising or constraining information omitted by the user, inserting coercions, imposing suitable views and so on, is traditionally called the *refiner*. The refiner (and not the *kernel* in charge of *proof checking*) is the real heart of these systems, and the primary source of their 'intelligence'. The constant improvements in the functionalities of this component is one of the main research trends in the field of interactive theorem proving. In particular, most of the studies aim to attain a tighter integration between the refiner and the modules in charge of proof automation, with the attempt to add limited deduction abilities to the former, using an interesting and synergistic analogy with similar studies on type systems for programming languages (see, for example, the recent, parallel investigations of type classes (Wadler and Blott 1989; Hall *et al.* 1996; Wenzel 1997; Sozeau and Oury 2008)).

5. Mathematics and computer science

The only feasible way of coming to grips with really radical novelty is orthogonal to the common way of understanding: it consists in consciously trying not *to relate the phenomenon to what is familiar from one's accidental past, but to approach it with a blank mind and to appreciate it for its internal structure.*

Edsger Dijkstra (Dijkstra 1986)

As observed by Van Den Bos (Bos 1979), the real novelty of the De Millo *et al.* paper was that

for the first time a paper on the philosophy of computer science, in this case the methodology of program verification, has been published in Communications of ACM.

Here, De Millo, Lipton and Perlis seem to have lost a great opportunity by failing to exploit the most interesting aspects of the analogy they had put forth and entrenching themselves behind a strictly sociological position and making use of old slogans like 'dullness of rigour', 'artificiality versus beauty', and similar things.

Actually, there are at least two major novelties introduced by the advent of computer science in the epistemological debate: the first is related to the intrinsic nature of computer science, which strongly differentiates it from mathematics (and all other scientific disciplines), while the second concerns the altered conditions induced by an extensive use of information technology in scientific practice.

[†] As Lakatos says, after Columbus, one should not be surprised if one does not solve the problem one has set out to solve.

Concerning the first point, if we look back at the Curry–Howard–De Millo correspondence, there is a striking difference between mathematics and computer science that should be evident at first glance. The point is that while programs (not algorithms, but programs!) are a major object of study, analysis and elaboration for computer science, the *mathematical* investigation of proofs is absolutely marginal, essentially confined to a minor subfield of logic known as 'proof theory', see, for example, Prawitz (1965) and Girard *et al.* (1989).

This is not surprising since computer science is about *information*, its automatic processing, transformation and communication, and the main vehicle for managing information are *programs* (again, programs, not algorithms!). Luckily, programs are informative entities, and not only can they be the object of a metamathematical investigation, but they can also be processed, transformed and communicated as any other kind of information. In fact, computer science starts to become really interesting when it is applied to itself. It is precisely this circularity, this auto-referentiality of computer science that makes it entirely peculiar among all scientific disciplines: meta-information is still another form of information; the techniques and methodologies of computer science are an essential part of its domain of investigation.

The difference with mathematics is striking, since the mathematical method has never been the object of a serious *mathematical* investigation (apart, possibly, from the timid, limited attempts of neopositivism). We should then acknowledge, following Popper, that since it cannot be made an object of validation experiments, and cannot possibly be refuted, the celebrated 'mathematical rigour' is a purely ideological claim, a mere illusion, or, if you prefer, the result of a refined liturgy.

A program is written in a strictly formal language, and the possibility of writing long programs itself testifies to the possibility of writing long *formal* proofs. Moreover, since programs (almost always) work, there must be a way to govern the *pedantic complexity of formal languages*. Here, we are completely reversing the traditional position: the point is not that verification is *important because programs crash*, but that verification *must be possible because, most of the time, they do not crash.*

An external observer might believe that this is due to particularly favorable, peculiar conditions of computer science, but this is not the case. The multilingual foundational miasma is a reality we have learned to live with (and, to some extent, to appreciate) at the descriptive/functional level. Change is a rule and adaptability a bare necessity (Fowler 2000):

There's a refrain I've heard on every problem project I've run into. The developers come to me and say 'the problem with this project is that the requirements are always changing'. The thing I find surprising about this situation is that anyone is surprised by it. In building business software requirements changes are the norm, the question is what we do about it.

Hence, admitting that long, sensible formal programs/proofs can be written, the really interesting question is if some of the methodologies, not only of static analysis but more generally of software development, can be applied to the realm of mathematics. In this way, the original verification/type-checking problem is put in its correct perspective, namely as one of the tools contributing to a comfortable development environment for the 'working mathematician'.

Here we also directly arrive at our second point, namely the exploitation of the possibilities offered by the new information technologies, and their impact on scientific practice. According to Popper, scientific rigour does not depend on the objectivity or critical attitude of individuals, but on the *methodology* employed (here, Kuhn expresses similar ideas (Hutcheon 1995), but in terms of standards and values of science dictated by the 'paradigm' adopted by the scientific community). Hence, if the advent of technology does materially affect the methodology, the notion of rigour will change accordingly, and with it the essence of the entire scientific discipline. The point is particularly important in the case of mathematics, where *changes in the criterion of 'rigour of the proof' engender major revolutions* (Lakatos 1976).

From many indications, we are approaching one of Kuhn's pre-revolutionary crises in the realm of mathematics. The big novelty of this crisis is due to the introduction of the use of the computer to master the growing complexity of mathematical proofs. Standard methods seem to have hit a ceiling, though not from intrinsic deficiencies of the theoretical framework, but from a human deficiency in coping with complex computations/encodings. As observed by Sarnak (Economist 2005), one of the editors of the *Annals of Mathematics*, they expect to receive a growing number of papers involving computer code in the next 20–50 years. So mathematics may become *a bit like experimental physics* – as foreseen by Sarnak – *where certain results are taken on trust, and independent duplication of experiments replaces examination* – or, as attempted by Hales in his Flyspeck project, we may try to use the computer as a remedy also. In both cases the notion of 'mathematical rigour' will be deeply affected.

6. Content and semantics

One of the major goals of verification is to provide a new dimension in the way we do mathematics, as well as in the way we do computer science.

W. D. Maurer (Maurer 1979)

The idea that a proof assistant should not just support the process of mathematical verification, but that of mathematical discovery has already been clearly outlined by Constable *et al.* in their description of the Nuprl system (Constable *et al.* 1986):

For our intention is to provide a medium for doing mathematics different from that provided by paper and blackboard. Eventually such a medium may support a variety of input devices and may provide communication with other users and systems; the essential point, however, is that this new medium is active, whereas paper, for example, is not.

In the nineties this goal was somewhat blurred by the imposing pronouncement of the QED manifesto[†], which, with its taxing goal *to provide a cultural monument to 'the fundamental reality of truth'* shifted the focus back onto formal verification.

QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques. The QED system will conform to the highest standards of mathematical rigor, including the use of strict formality in the internal representation of

[†] See `http://www-unix.mcs.anl.gov/qed`.

knowledge and the use of mechanical methods to check proofs of the correctness of all entries in the system.

The manifesto describes the ambitious goals of the project and discusses questions and doubts, and the answers to them. There were two workshops on QED, in 1994 and 1995, but none since then. Is the QED manifesto too ambitious? In this respect it is instructive to read what the authors of the QED manifesto thought was needed to be done. First, a group of enthusiastic scientists should get together to determine which parts of mathematics are needed to be formalised, in what order and with which cross-connections. The authors assume that this phase may take a few years and it may even involve a rearrangement of the mathematics itself, before the actual formalisation work can start. Other points in this ‘to-do-list’ are of a similar top-down nature.

However, this is a rather old fashioned approach to the problems, focusing solely on formal correctness. Developments like Wikipedia show that a more ‘bottom up’ distributed approach may work better, using a simple lightweight basic technology. One could claim that for mathematics – where the end goal is to get a library of verified reliable results – such an approach could never work, but for Wikipedia the same doubts were raised at first: Wikipedia is typically something that works in practice but not in theory. (See Wiedijk (2007) for a present day evaluation of the QED manifesto.)

The goal of developing innovative, semantic based functionalities transcending the mere operation of formal checking, and focusing on problems related to the management of the repository of (formal) mathematical theorems, such as archiving, indexing, searching, communication and publishing was strongly advocated in Asperti *et al.* (2000). The emerging XML technology seemed to provide the natural infrastructure for the development of the new systems. In particular, the idea was to use XML as a main, platform independent language for long-term representation and exchange of the naturally structured, formal mathematical knowledge, exploiting to their full extent all kinds of XML technologies: MathML and XHTML for rendering; XSLT for the application of notational transformations; Xpath and XQuery for complex, content based queries; and RDF for indexing and efficient document retrieval. It is a pity that, since then, most of the expectations for XML technologies have not been fulfilled due to intrinsic deficiencies in their design and implementation: MathML failed to be adopted by major browsers; XSLT is just too prolix for simple operations and too weak for more complex ‘content sensitive’ operations; XQuery is too slow for large, highly structured data bases; and RDF never really went beyond the project phase.

An alternative attempt to employ XML for encoding mathematical content was made by the OpenMath project (Dewar 2000). In Dewar’s words, OpenMath is a standard for representing mathematical data in *as unambiguous a way as possible*. Essentially, an OpenMath object is a labelled tree describing the abstract syntax tree of the mathematical entity, whose leaves are the basic OpenMath data structures, such as IEEE double precision floats; Unicode strings; byte arrays, variables or symbols. Symbols consist of a name and a reference to a ‘definition’ in an external document called a content dictionary (CD). The definition itself is given in natural language, while CD’s are essentially meant as background references for the implementers of *phrasebooks*, that is, of the actual software tools able to internalise the OpenMath object inside specific applications.

A ferocious critique of OpenMath is contained in Fateman (2001). Although we share most of Fateman's opinions and, in particular, the concern about the lack of any serious *proof of concept*, there are a few points that probably deserve a deeper discussion. In particular, Fateman affirms that

All protestations to the contrary, it [OpenMath] simply does not have any mandate outside the rather simple application of denoting what could be trivially done in any programming language capable of representing attributed trees.

This is true, but the point is not to just use abstract syntax trees for representing the information, but *to agree* on their syntax, that is, to propose a standard. Sharing a common grammar seems to be a minimal pre-requisite for any possible kind of communication between automatic devices. Then, the use of XML is indeed not essential, but quite natural. The tremendous step forward consisting of passing from an unstructured representation of the information to a structured (standard) one, simply cannot be ignored. Of course, what makes a standard is not a self-proclamation, but its actual adoption, and OpenMath clearly failed in its mission. However, this does not imply that the objective was basically wrong.

The second point is more delicate. Fateman says:

We learn that each corresponding program X must have a phrasebook which converts its internal form Y to an OpenMath form which is, one hopes, the universal semantic notion of Y But it seems that except in trivial matters, its semantics may have to be encoded as 'the meaning of Y to the program X'. Thus the ideal of having n programs communicating using n phrasebooks to/from OpenMath has been lost.

Of course, it is not the 'n^2 versus n' point that matters, but the fact that we would entirely lose the real *sense* of having an intermediate structure. If we cannot give any *intended* semantic interpretation to our syntax, what is the actual point in having it? The critique seems to undermine irreparably and at its very roots the quest for a universal 'intermediate' language.

In fact it does not. Suppose we ask an application to compute a solution for a given problem P. We have no way to be sure that it really understood our problem. After a given time it returns a solution 'a'. We have no way to be sure we correctly interpreted the solution, either. But *who cares?* We *check* if *our* interpretation of a is a solution to *our* interpretation of P, and if it is, we are happy. The point is that the intermediate information is merely a *witness*, a *trace* that we have to interpret and check after internalisation. The interest is, as usual, that checking is enormously less expensive than finding. The general picture is even more clear when we add proofs. Suppose there is in our 'universal' language a proof p of some property A. We may define translations p_x and A_x to our internal language and check if indeed p_x is (under our interpretation) a proof of A_x. If it is, we have a proof, *without caring whether the translation was 'correct'* (and surely it could not be, since we have no semantics for the intermediate language).

Having understood that we may have an interesting intermediate language *with no semantics*, the actual points are:

1 Can we define a 'trace' language for proofs that are of suitable interest to multiple applications?

2 Can we apply static analysis techniques for this language (for example, a weak, possibly logically inconsistent type system)?
3 Can we promote a direct collaborative development of this layer?

Regarding the first point, as we have already observed, the process of formalisation of a mathematical statement is often compared to the translation of a piece of code from a high-level programming language to some assembly language, corresponding to a foundational dialect in some logical system. Inter-operability at a foundational level is as hopeless as trying to send instructions from one microprocessor to another. On the other side, the most expensive part of the formalisation process consists of a preliminary conceptual phase of transformation of the proof into a form *suitable to be understood by a machine* that is largely independent of the specific idiosyncrasies of each particular foundation. Compilation is not an atomic process, and what is currently clearly lacking is a good intermediate language – which is precisely the trace language we are talking about.

Is such an objective feasible? If we overcome the foundational impasse of fixing a formal semantics to quantifiers, and the diatribe about the role and nature of functions, it is usually acknowledged that we could probably agree on a common *syntax* for mathematical *formulae*. A minimal trace language, which is extremely poor but not completely deprived of interest, would consist of a graph of dependencies relating a result to the main auxiliary facts required for its proof (possibly admitting multiple paths). The interest of such an approach is that it can be refined to a more or less arbitrary degree of detail, and to the point where a software system can automatically fill in the missing steps. Moreover, the system itself could assist the user in this refinement activity. The other point is that the approach does not vastly differ from the natural *top-down* methodology already in use in wiki-like systems (where you typically *first* create the link and *then* the page you are linking). If we merely ask a typical mathematical user of such a system to type mathematical statements using a suitable set of content-oriented LaTeX-like macros (possibly to be agreed according to the same policies that usually govern these systems) we really see no reason why he should not consent (especially since he could also have some presentational benefits). Quoting Hales again (Hales 2008):

To undertake the formalisation of just 100,000 pages of core mathematics would be one of the most ambitious collaborative projects ever undertaken in pure mathematics, the sequencing of a mathematical genome. One might imagine a massive wiki collaboration that settles the text of the most significant theorems in contemporary mathematics from Poincaré to Sato-Tate.

The point is that everybody must be able to contribute, independent of whether they are using a proof assistant or not. Interactive theorem-prover users can be in charge of refining the proofs to more elementary components, possibly automatically populating the library of basic results (which are mostly meant to be inspected by automatic devices only). At the same time, the refinement process can provide an essential feedback to higher-levels, possibly requiring some revisiting of already codified notions and results.

Preliminary developments towards a 'MathWiki' system that supports the distributed development of a library of mathematics on various levels of formality, ranging from the mathematics informally described on existing web pages to the formalised mathematics that we encounter in proof assistants, have been described and advocated in Corbineau and Kaliszyk (2007) and Corbineau *et al.* (2008). Such a system should provide a lightweight cooperative framework for developing and discussing mathematics, on various levels of formality. It should also provide a place where various existing repositories of (formal) mathematics come together.

7. Conclusion

The lack at this late date of even a single verification of a working system has sometimes been attributed to the youth of the field. ...there has never been a verification of say, a Cobol system that prints real checks; lacking even one makes it seem doubtful that there could at some time in the future be many.

R. A. De Millo, R. J. Lipton and A. J. Perlis (De Millo *et al.* 1979)

As we have already observed, the gloomy predictions of De Millo *et al.* have been largely refuted. Formal verification is at present a concrete reality, permitting correctness proofs of complex software applications. For instance, in the framework of the Verifix Project a compiler from a subset of Common Lisp to Transputer code was formally checked in PVS (see Dold and Vialard (2001)). Strecker (Strecker 1998) and Klein (Klein 2005) certified bytecode compilers from a subset of Java to a subset of the Java Virtual Machine in Isabelle. In the same system, Leinenbach (Leinenbach *et al.* 2005) formally verified a compiler from a subset of C to a DLX assembly code. The Compcert project, headed by Xavier Leroy, has recently produced a verified optimising compiler from C to PowerPC assembly code, based on the use of the Coq proof assistant both for programming the compiler and proving its correctness (Leroy 2006; Tristan and Leroy 2008). Similar achievements have been obtained in other fields of computer science, spanning the range from hardware (Harrison 2007) to operating systems (Alkassar *et al.* 2009; Klein 2009).

However, the parallel drawn by De Millo, Lipton and Perlis between computer science and mathematics is still relevant, and possibly even more so in view of the recent proliferation of mathematical proofs involving the use of computers. In particular, in this paper we have argued that mathematics is destined to assimilate some practices of software development, and that the future of both mathematics and programming lies in the fruitful *combination* of formal verification and the usual social processes that are already working in both scientific disciplines. Quoting Hales (Hales 2008):

The hope is the system [the proof assistant] will eventually become sufficiently user-friendly to become a familiar part of the mathematical workplace, much as email, TEX, computer algebra systems and Web browsers are today.

At present, we are still a very long way from this dream; the current cost of transcribing a printed page of textbook mathematics into machine-checkable code is estimated as a week's labour in Hales (2008), and, more pessimistically, as 1.5 hours per line in Asperti and Ricciotti (2009). In Wiedijk (2001) the cost of formalising the standard bachelor's curriculum of mathematics is estimated at 140 man years. The point is not only to reduce this cost, but also to improve the benefits coming from the representation of the information in a 'machine understandable' richly structured format that is suitable for elaboration by a machine. This means developing innovative, content-based functionalities, eventually overcoming the reductive operational perspective of *verification*. The research directions were clearly traced in Constable *et al.* (1986) more than 20 years ago:

The natural growth path for a system like Nuprl tends toward increased 'intelligence'. [...] For example, it is helpful if the system is aware of what is in the library and what users are doing with it. It is good if the user knows when to involve certain tactics, but once we see a pattern to this activity, it is easy and natural to inform the system about it. Hence there is an impetus to give the system more knowledge about itself

Unfortunately, progress in this direction is extremely slow, and the following question asked by Dijkstra still remains, for the moment, unanswered:

In the relation between mathematics and computing science, the latter has been for many years at the receiving end, and I have often asked myself if, when, and how computing would ever be able to repay its debt.

Edsger Dijkstra (Dijkstra 1986)

References

Alkassar, E., Bogan, S. and Paul, W. J. (2009) Proving the correctness of client/server software. *Sadhana* **34** (1) 145–191.

Altenkirch, T., McBride, C. and McKinna, J. (2005) Why dependent types matter. (Available at `http://sneezy.cs.nott.ac.uk/epigram/`.)

Asperti, A., Padovani, L., Sacerdoti Coen, C. and Schena, I. (2000) Content-centric logical environments. Short Presentation at the Fifteenth IEEE Symposium on Logic in Computer Science.

Asperti, A. and Ricciotti, W. (2009) About the formalization of some results by Chebyshev in number theory. In: Proc. of TYPES'08. *Springer-Verlag Lecture Notes in Computer Science* **5497** 19–31.

Avigad, J., Donnelly, K., Gray, D. and Raff, P. (2007) A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.* **9** (1).

Barbanera, F. and Berardi, S. (1996) A symmetric lambda calculus for classical program extraction. *Information and Computation* **125** (2) 103–117.

Boolos, G. (1984) Don't eliminate cut. *Journal of Philosophical Logic* **13** 373–378.

Bos, J. V. D. (1979) Letter to the editor. *Communications of the ACM* **22** 623.

Bourbaki, N. (1968) *Theory of Sets*, Elements of mathematics, Addison Wesley.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. and Smith, S. F. (1986) *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall.

Coquand, T. (2008) Draft of the Formath Project.

Corbineau, P., Geuvers, H., Kaliszyk, C., McKinna, J. and Wiedijk, F. (2008) A real semantic web for mathematics deserves a real semantics. In: Lange, C., Schaffert, S., Skaf-Molli, H. and Völkel, M. (eds.) SemWiki. *CEUR Workshop Proceedings* **360**.

Corbineau, P. and Kaliszyk, C. (2007) Cooperative repositories for formal proofs – a wiki-based solution. In: Kauers, M., Kerber, M., Miner, R. and Windsteiger, W. (eds.) Towards Mechanized Mathematical Assistants. *Springer-Verlag Lecture Notes in Computer Science* **4573** 221–234.

De Millo, R. A., Lipton, R. J. and Perlis, A. J. (1979) Social processes and proofs of theorems and programs. *Commun. ACM* **22** (5) 271–280.

Dewar, M. (2000) Special issue on OpenMath. *ACM SIGSAM Bulletin* **34**.
Dijkstra, E. W. (1986) On a cultural gap. *Mathematical Intelligencer* **8** (1) 48–52.
Dold, A. and Vialard, V. (2001) A mechanically verified compiling specification for a lisp compiler. In: Proc. of FSTTCS 2001. *Springer-Verlag Lecture Notes in Computer Science* **2245** 144–155.
The Economist (2005) Proof and beauty. *The Economist*, 31st March 2005.
Fateman, R. (2001) A critique of OpenMath and thoughts on encoding mathematics. (Available at `http://www.eecs.berkeley.edu/~fateman/papers/openmathcrit.pdf`.)
Feit, W. and Thompson, J. G. (1963) Solvability of groups of odd order. *Pacific Journal of Mathematics* **13** 775–1029.
Fowler, M. (2000) The New Methodology. (Available at `http://www.martinfowler.com/articles/newMethodology.html`.)
Geuvers, H. (2009) Proof Assistants: history, ideas and future. *Sadhana* **34** (1) 3–25.
Girard, J.-Y., Lafont, Y. and Taylor, P. (1989) *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.
Gonthier, G. (2007) The four colour theorem: Engineering of a formal proof. In: Proc. of ASCM 2007. *Springer-Verlag Lecture Notes in Computer Science* **5081**.
Gonthier, G. (2008) Formal proof – the four color theorem. *Notices of the American Mathematical Society* **55** 1382–1394.
Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E. and Thery, L. (2007) A modular formalisation of finite group theory. In: The 20th International Conference on Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **4732** 86–101.
Hales, T. C. (2005) A proof of the Kepler conjecture. *Ann. Math.* **162** 1065–1185.
Hales, T. C. (2007) The Jordan curve theorem, formally and informally. *The American Mathematical Monthly* **114** 882–894.
Hales, T. C. (2008) Formal proof. *Notices of the American Mathematical Society* **55** 1370–1381.
Hall, C., Hammond, K., Jones, S. P. and Wadler, P. (1996) Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* **18** 241–256.
Halmos, P. (1985) *I want to be a Mathematician: An Automathography*, Springer-Verlag.
Hamid, N. A., Shao, Z., Trifonov, V., Monnier, S. and Ni, Z. (2003) A syntactic approach to foundational proof-carrying code. *J. Autom. Reasoning* **31** (3-4) 191–229.
Hardy, G. H. (1928) Mathematical proof. *Mind* **38** 1–25.
Harrison, J. (2007) Floating-point verification. *J. UCS* **13** (5) 629–638.
Harrison, J. (2008) Formal proof – theory and practice. *Notices of the American Mathematical Society* **55** 1395–1406.
Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Commun. ACM* **12** (10) 576–580.
Howard, W. A. (1980) The formulae-as-types notion of construction. In: Seldin, J. P. and Hindley, J. R. (eds.) *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press 479–490.
Hutcheon, P. D. (1995) Popper and Kuhn on the evolution of science. *Brock Review* **4** (1/2) 28–37.
Klein, G. (2005) Verified Java bytecode verification. *Information Technology* **47** (2) 107–110.
Klein, G. (2009) Operating system verification – an overview. *Sadhana* **34** (1) 27–69.
Lakatos, I. (1976) *Proofs and Refutations: The Logic of Mathematical Discovery*, Cambridge University Press.
Lamport, L. (1979) Letter to the editor. *Communications of the ACM* **22** 624.
Langley, S. P. (1891) *Experiments in Aerodynamics*, Kessinger Publishing.
Lecat, M. (1939) *Erreurs de mathématiciens: des origines à nos jours*, Ancienne Librairie Castaigne, Brussels.

Lee, J. K. (2002) Philosophical perspectives on proof in mathematics education. *Philosophy of Mathematics Education Journal* **16**.

Leinenbach, D., Paul, W. J. and Petrova, E. (2005) Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, IEEE Computer Society 2–12.

Leroy, X. (2006) Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA* 42–54.

MacKenzie, D. (2005) What in the name of Euclid is going on here? *Science* **207** (5714) 1402–1403.

Maurer, W. D. (1979) Letter to the editor. *Communications of the ACM* **22** 625–629.

Necula, G. C. and Lee, P. (1996) Proof-carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University.

Parigot, M. (1992) Lambda-mu calculus: An algorithmic interpretation of classical natural deduction. In: Proc. of the Logic Programming and Automated Reasoning International Conference LPAR'92. *Springer-Verlag Lecture Notes in Computer Science* **624** 190–201.

Popper, K. (1948) Logic without assumptions. *Aristotelian society proceedings* **47** 251–292.

Popper, K. (1963) *Conjectures and Refutations. The Growth of Scientific Knowledge*, Routledge.

Prawitz, D. (1965) *Natural Deduction: a proof theoretical study*, Almqvist and Wiksell.

Schieber, P. (1987) The wit and wisdom of Grace Hopper. *OCLC Newsletter* **167**.

Sozeau, M. and Oury, N. (2008) First-class type classes. In: *TPHOLs* 278–293.

Strecker, M. (1998) *Construction and Deduction in Type Theories*, Ph.D. thesis, Universität Ulm.

Tristan, J.-B. and Leroy, X. (2008) Formal verification of translation validators: a case study on instruction scheduling optimizations. In: *Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008* 17–27.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. In: *POPL '89: Proc. of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM 60–76.

Wenzel, M. (1997) Type classes and overloading in higher-order logic. In: *TPHOLs* 307–322.

Wenzel, M. (1999) Isar – a generic interpretative approach to readable formal proof documents. In: Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **1690** 167–184.

Wiedijk, F. (2001) Estimating the cost of a standard library for a mathematical proof checker. (Available at `http://www.cs.ru.nl/~freek/notes/mathstdlib2.pdf`.)

Wiedijk, F. (2007) The Qed manifesto revisited. In: Matuszwski, R. and Zalewska, A. (eds.) From Insight to Proof, Festschrift in Honour of Andrzej Trybulec. *Studies in Logic, Grammar and Rhetoric*, University of Białystok **10** (23) 121–133.