Mixing Computations and Proofs

MICHAEL BEESON
Professor Emeritus, San José State University

We examine the relationship between proof and computation in mathematics, especially in formalized mathematics. We compare the various approaches to proofs with a significant computational component, including (i) verifying the algorithms, (ii) verifying the results of the unverified algorithms, and (iii) trusting an external computation.

1. INTRODUCTION

Computers have been used in mathematics since the dawn of the Computer Age; Turing himself investigated the Riemann hypothesis by computations on the Manchester Mark I in 1950 [Boo06]. General purpose symbolic computation systems are now in routine use, performing integrations, algebraic and combinatorial computations, and number-theoretical computations. When a mathematician wants to perform or check a complicated calculation, he or she naturally turns to a computer program, and this is no longer a new development. But when a proof has to be checked for correctness, the mathematical community still relies on a social system rather than a computer system. Just in the last few years, some large proofs have been checked by computer. Headlines were made by Gonthier's verification of the four-color theorem [Gon08], especially since that same theorem made headlines forty years ago when a computer was used to make the calculations required by the proof. At the time, doubts were raised as to whether the program(s) might have bugs leading to missing a counterexample.

In 2012, Gonthier headed a team that completed another large formal proof [GAA$^+$13]. This proof used 170,000 lines, 15,000 definitions, and 4300 theorems to prove the "odd-order theorem" of Feit-Thompson (every group of odd order is solvable). In 2014 the proof of the Kepler Conjecture was checked, by an informal international collaboration led by Thomas Hales [HAB$^+$15]. That proof also involved extensive computations, originally performed by computer. The referees who recommended publication of Hales's original article would not certify the correctness of the proof, due to doubts about those computations. These doubts, like those about the four-color theorem and the odd-order theorem, have been laid to rest by formalization.

These examples show the importance of being able to formalize proofs that depend heavily on computation. They were done in two different systems and used different approaches. All three proofs required many person-months of effort to formalize; an amount of effort likely to be expended only on a result of great mathematical interest. Indeed, the last two examples were completed by teams numbering in the dozens; the list of authors looks more like a paper in particle physics than one in mathematics. All three required a great deal of *ad hoc* work, as well as the formalization of some underlying mathematics that was not specific to the problem at hand. Here we investigate why these matters are not routine, and what the possibilities may be to make them more routine.

The author gratefully acknowledges the many helpful suggestions of the referees.

2. WHAT'S HOLDING UP THE QED SINGULARITY?

The QED Singularity, mentioned (though not by that name) in [Wie08], is the future time when formal proofs will be the norm in mathematics. It is now only a gleam in the eye. Some mathematicians (most of those I know personally) take the view that formal mathematics is either not even useful, or is not worth the cost (the time and energy would be better spent proving new theorems informally). There are two notable exceptions among the famous: Thomas Hales and Vladimir Voevodsky. Each of these had a large and complicated proof and wanted certainty about its correctness; Hales because the referees had expressed doubt, and Voevodsky because a mistake had been found in another proof he had long believed correct. Both have become advocates for formalized proofs; but that view is still the view of a small minority. Here we investigate the reasons for that.

2.1 Why do we want proofs?

The reasons for the view that formal mathematics is "not worth the trouble" are not simple to summarize; they go back to the roots of mathematics. What is the reason for proving things at all? A brash student asked a professor friend of mine, "What do we need all this rigor for?" This was during a discussion of Euclid's "proof" of the side-angle-side congruence criterion, in which Euclid argues that one triangle can be moved rigidly atop the other, and the third vertex must also coincide. This argument was found "convincing" by most students, and of course it *is* convincing in some sense, but it *isn't* a proof from Euclid's axioms. Similarly, the numerical evidence for the Riemann hypothesis would satisfy any physicist as to its truth, but not the mathematicians. (There are hundreds of millions of confirming numerical computations.) There are two possible answers to the question why we want proofs:

—To *know* that the theorem is true.

—To know *why* the theorem is true.

It is the difference between these answers that accounts for the ambivalence about long computer-checked proofs. After Appel and Haken's proof by computation of the four-color theorem, the referees commissioned an independent program to check the computations; that provided some evidence that there was no program bug. Assuming there was no bug, one then had to believe *that* the theorem was true. But simply to say that all the many cases had been checked did not give people the feeling of understanding *why*. Gonthier's formalized proof served to remove all doubt about a possible bug; and it may have come closer to explaining why, since he did improve the analysis: distinguishing between the topological and combinatorial aspects of "planar", he eliminated the use of the Jordan curve theorem and introduced some helpful new concepts, as described in [Gon08]. These things helped us understand better the part of the proof that reduces an infinite number of cases to a finite number. But still, a large finite number of cases need to be computed, and the proof is the result of a long computation. There just isn't any map requiring five colors. We don't know "why"; it's just how the world is.

Similarly, there's no better way than Kepler found to pack oranges; that's just how the world is. The "reason" is many thousands of lines of proof and computation; if it can be shortened to digestible size, that remains for a future generation.

The resulting ambivalence accounts for the fact that the mathematical community has not made a Manhattan Project out of the need to develop easily usable proof-checkers and deploy them. That task has been left to a handful of developers and users, who have not succeeded in convincing many mathematicians to follow them. That is evidently because the cost-benefit analysis does not look promising to the mathematicians. The cost is many hours, first to learn how to use a formal proof-checker, then to fill in the formal details of their proofs. (Often these two steps are separated by the necessity to first develop a library of needed theorems, before even arriving at the proof whose formalization is of interest.) The benefit is the certainty that the proof is correct in every detail. But in general that does not earn a promotion or a salary increase, or an increase in prestige or the respect of colleagues; and if it must be balanced against the benefit of writing three additional unverified papers in the time it would take to formalize one, the choice is usually the former.[1]

2.2 Structured proofs

Contrast the far-from-universal use of proof-checking with the near-universal adoption of TEX. Evidently the cost-benefit analysis for TEX came out much better: beautiful mathematics, easily revised almost instantly, versus the cost of learning a few typesetting commands.[2] Leslie Lamport, the author of LATEX, has long advocated a semiformal approach to writing "structured proofs" [Lam12, Lam95]. He says

> [M]athematical notation has changed considerably in the last few centuries. Mathematicians no longer write formulas as prose, but use symbolic notation such as $e^{i\pi} + 1 = 0$. On the other hand, proofs are still written in prose pretty much the way they were in the 17th century. The proofs in Newton's *Principia* seem quite modern. This has two consequences: proofs are unnecessarily hard to understand, and they encourage sloppiness that leads to errors.

Lamport maintains that writing structured proofs will make them both easier to read, and more rigorous (hence more likely to be correct). But he says, "Learning both a new way to write proofs and how to be more precise and rigorous was too high a barrier for most mathematicians." In the more recent of his two articles, he toned down the goal:

> [T]he important goal is to stop writing 17th century prose proofs in the 21st century.

[1]A referee pointed out that in some parts of computer science, formal proofs are becoming the norm. But these developments do not seem to have accelerated the pace of formalization of mathematics.

[2]No doubt the introduction of LATEXhelped reduce the cost and increase the benefits. In some sense LATEXis a user interface to TEX, and it illustrates the point that a good user interface helps.

But this goal does not seem to have attracted much of a following, in spite of Lamport's persuasive articles. Lamport has also provided a LaTeX package to support writing structured proofs; even that did not produce a flood of structured proofs. If the structured proofs so typeset could be automatically checked for correctness, with errors generated like TeX errors, would that tip the balance in the cost-benefit analysis?[3]

2.3 Formal proof sketches

A similar idea, intended to address in another way the difficulty of having to write too many and too detailed steps, is the "formal proof sketches" introduced in [Wie04]. A formal proof sketch is a formal proof, without labels and references and with some details omitted. It is nevertheless "formal" because it is a precise question whether an alleged formal proof sketch can or cannot be completed to a formal proof. The cited paper compares the features of Lamport proofs and formal proof sketches, finding the latter more readable, and proposes that they be supported in a proof development environment. The crucial part of the proposal is "The same proof language should be used for the formal proof sketches and for the full formalizations."

2.4 Ways QED can be too difficult

The QED project was explicitly about building a database of formalized mathematics; but it was implicitly about making formalized proof more common and easier. (There is only one line in the QED manifesto mentioning "interface.") Of course, the hope is to set up a positive feedback: the larger the formalized database, the more the tools are used, the more they are improved, the easier it is to used them. It is this feedback loop that should, according to Wiedijk, result in the QED singularity. Since this result has not yet occurred, twenty years after the original QED workshop, we should analyze the (at least perceived) difficulties. Why is proof-checking so difficult?

—I have to write too many steps in too much detail.
—I have to write the proofs in a vastly different way than I would naturally write informal proofs in TeX.
—The system does not know facts at the undergraduate level.
—Well-known theorems (meaning theorems so well-known that I would not have to cite a reference to them in a published paper) should be known to the system.
—Referencing such well-known theorems should be easy, e.g., it should not cost a lot of time to look them up.
—It is too hard to become sufficiently expert with a proof-checker (e.g. to learn its interface, libraries, and workflow).

All these are problems with all of today's systems. The first two are already problems with using Lamport's structured (informal) proofs, but the lack of formality

[3] Even among the subset of the QED community that believes in writing declarative proofs, there is not complete agreement on how they should be written. Lamport emphasizes naming proof steps; by contrast the Isar language [Wen99], which conforms to Lamport's requirements in some respects, makes a point of *not* naming the steps of a proof.

means that the last three problems do not arise with structured proofs. Thus: Structured proofs require too many steps and a constrained style, and that is not (thought by many to be) worth the payoff in increased rigor and likelihood (but not certainty) of preventing errors. To get certainty of correctness we need to solve all four difficulties, and so far, that is not judged worth the price (by most mathematicians), except in high-stakes cases like the Kepler conjecture.

The first difficulty, "I have to write too many steps", leads naturally to the desire that the lowest-level steps in a proof-checker should be automated. That is, the line between proof-checking and proof-finding programs is not absolute; proof-checkers should *find* those steps that seem trivial to humans. Witness the following quotation from the HOL Light tutorial [Har11], p. 17:

> Typically a proof in HOL proceeds as follows. The user employs special insight into the problem to break it down into a series of relatively simple subproblems, and once the subproblems are simple enough or fall within a limited enough domain, they can be dealt with automatically by HOL. Generally speaking, the level of detail needed before HOL can fill in the gaps is greater than most people are used to. On the other hand, there are pleasant exceptions where one can replace a fairly long manual proof with a single automated HOL step.

The part about "fall within a limited enough domain" refers to the effort to include a number of different decision procedures for specific decidable theories. This has been the guiding principle of the proof-checker PVS developed at SRI [OSR92]. In [Sha09], there is a short list of some of the important decision procedures in PVS: SAT and SMT procedures, binary decision diagrams, symbolic model checking, predicate abstraction, and decision procedures for monadic second-order logic and Presburger arithmetic. The idea is to use interaction to reduce a goal until it lies in a decidable theory.

3. LOGIC AND COMPUTATION

The previous section surveyed the aims of the QED project and reviewed some of the difficulties. Now let us consider the causes of and the remedies for those difficulties. Two obvious causes are inadequate libraries (of theorems) and inadequate user interfaces. We shall not discuss the remediation of those problems (see [Geu09] for such a discussion.) What we *shall* discuss is another cause, perhaps more fundamental in nature: the interplay between logic and computation. That is important, because the connection between proof and supporting computations is a weak point of mathematics. People are often willing to trust unverified computations, especially if those computations are made by familiar software that has produced no obvious errors in the past. Everyone trusts their calculator (except professors of numerical analysis). Most mathematicians trust common symbolic computation software, even if they know that it can sometimes deliver wrong answers due to mistaken hidden assumptions. They think they can detect such situations and that if the answer appears reasonable it is correct. Yet, if there are many computations, or they are performed by special-purpose software, trust evaporates.

The QED software of the future must be able to smoothly integrate logical proof steps with computation, perform large computations efficiently, and leave no doubt as to the correctness either of the computations or their connection to the logical part of the proof. There is at present no universal consensus on how this is to be accomplished, and different systems work differently.

3.1 What is the difference between logic and computation?

Mathematics consists of logic and computation, interwoven in tapestries of proofs. "Computation" refers to chains of formulas progressing towards an "answer", such as one makes when evaluating an integral or solving an equation. Typically computational steps move "forwards" (from the known facts further facts are derived), and involve chains of inferences of equalities or inequalities.

Logical steps, on the other hand, often move "backwards" (from the goal towards the hypothesis, as in *it would suffice to prove.* Of course, logical steps can also move forwards, and many proofs work forwards from the assumptions and backwards from the goal until the inferences "meet in the middle." Hence the distinction between computation and proof is not the same as the forward–backward distinction. Sometimes the logic associated with a computation also runs backwards, as when solving algebraic equations.

As is well-known, propositional logic can be regarded as boolean equations, so a backwards-running proof converts to a forwards-running computation in which we try to rewrite the goal to the constant `true`. This we regard as an example of reducing proof to computation, and it further illustrates the point that computations consist of forward chaining of equalities or inequalities. In lucky cases, we can reduce an infeasible search for a proof to an efficient execution of some algorithm; that is when it pays to reduce proof to computation.

The mixture of logic and computation gives mathematics a rich structure that has not yet been captured, either in the formal systems of logic, or in computer programs. We have computer languages that are used to represent algorithms in a formal way; we have logical languages that are used to represent reasoning in a formal way. None of the programming languages used to implement the apps on your smartphone can represent logic (directly). None of the languages mentioned in logic textbooks can represent real, implementable algorithms.[4] The difficulty of merging proof and computation has yet to be properly solved even at a theoretical level. You will see this fundamental theoretical difficulty appearing again in the last sentence of this paper.

Expert readers will at this point be jumping to contradict me, pointing to certain functional programming languages, for example OCaml, in which HOL Light is written and which has supported John Harrison's pathbreaking book [Har09b]. There are also Agda and Idris, based on Martin-Löf's type theories. OK, I accept that point, and we will consider HOL Light further below. There may also still be some Prolog programmers who insist that computation reduces to logic; and there may be some LISP programmers and users of NQTHM who think that logic reduces to computation. But the fact remains that in the mainstream of logic, and in the mainstream of programming, logic and computation remain separate.

[4]A few logical languages, like Gödel's theory of functionals of finite type, do have terms that correspond to number-theoretic algorithms, but nobody actually writes programs in those languages.

3.2 Kinds of Mathematical Reasoning

In order to elucidate the connections between mathematical reasoning and computation, we are going to make a short survey of some different kinds of mathematics. Librarians and journal editors are accustomed to classifying mathematics by subject matter, but that is not what we have in mind. Instead, we classify mathematics by the *kind of proofs* that are used. In the following list, the items are not intended to be exclusive; items lower in the list represent more complex proofs and may include the kinds of arguments earlier in the list.

—Purely logical, as in checking that one line of a proof does really follow from the stated previous lines.

—Simple theory, as in geometry (one kind of object, few relations). Euclid's Books I to IV, for example.

—Equational. For example, the famous identity that the product of two sums of four squares is again a sum of four squares; or deriving the famous ten Knuth-Bendix identities of group theory.

—Simple algebraic calculations, but not just normalization. For example: Simplify the following expression:

$$\frac{a^2-(b-c)^2}{(a+c)^2-b^2}+\frac{(a-c)^2-b^2}{(a+b)^2-c^2}+\frac{(a-b)^2-c^2}{(b+c)^2-a^2}$$

A systematic normalization procedure will usually produce a much longer and more complicated proof than an experienced human, which is why this example does not come under the previous category. In case you would like to paste this into your favorite software, here it is in machine-readable form:[5]

```
(a^2-(b-c)^2)/((a+c)^2-b^2) + ((a-c)^2-b^2) /((a+b)^2-c^2)
+ ((a-b)^2-c^2)/ ((b+c)^2-a^2)
```

—Uses natural numbers and mathematical induction. For example, the formula for the sum of the first n squares.

—Uses definitions (perhaps lots of them). For example, point set topology, with definitions of open, closed, convergence, limit point, closure, isolated, perfect set, etc.

—Uses a little number theory and simple set theory. For example, the theorem that the order of a subgroup divides the order of a group (Lagrange's theorem). Abstract algebra up through Galois theory falls in this category.

—Uses calculus (limits, derivatives, integrals).

—Uses inequalities heavily. For example, the arithmetic mean is less than the geometric mean. The Flyspeck project to prove the Kepler conjecture relied heavily on the verification of algebraic inequalities. Here is an example (mentioned in a talk by Voevodsky):

$$-1.44 < x_6x_2^2 + x_5x_3^2 - x_1x_4^2 + x_4^2 - \frac{1}{2}x_1 + \frac{4}{3}x_4$$

[5]A referee said an answer would be helpful: $(a+b-3c)/(a+b+c)$. *MathXpert* gets that answer in auto mode, accounting for the "usually" in the text. Can you get that answer out of other software? Or prove the resulting identity in your favorite proof-checker?

given that $(x_1, x_2, x_3, x_4, x_5, x_6)$ belongs to

$$[(-1, -0.1, -0.1, -1, -0.1, -0.1), (0, 0.9, 0, 5, -0.1, -0.05, 0.03)]$$

There is a list of 100 theorems on which the proof-checking community has been testing its provers [Wie99]. It is an interesting exercise to classify the 100 theorems according to the kind of proof techniques involved. All but a few are easily classified. (Those few involve more set theory.)

Mathematicians divide proofs into "algebra" and "analysis". What distinguishes the two? I think it is the use of inequalities that divides analysis from algebra. At least, no serious analysis can be done without inequalities, and one rarely sees inequalities in algebra. A referee disagreed, saying that algebra is concerned with first-order structures, and analysis is concerned with functions. Algebra, however, is usually second-order, e.g., it deals from the outset with cosets, subgroups, homomorphisms and isomorphisms, etc., whereas "hard analysis" often deals only with specific functions. For example in [Tit51] no function variables meet the eye, but I do find many inequalities.

3.3 My experience

Over the years, a number of my projects have dealt with aspects of proof and computation. Here is a list of the projects and the corresponding issues:

—Symbolic computation with logical correctness in *MathXpert* [Bee89b, Bee89c, Bee89a, Bee97]

—Reducing logic to computation I: Using infinitesimals in limit computations in *MathXpert* [Bee95].

—Precise semantics of limits using filters [BW05]

—Reducing logic to computation II: convergence tests for infinite series in *MathXpert*, and asymptotic inequalities (2012, unpublished)

—Linking proof to computation, by calling *MathXpert* from theorem-prover Otter-λ, with resulting applications to proof by mathematical induction [Bee06].

—Theorem prover *Weierstrass* and the proof of irrationality of e [Bee01]. Computations from *MathXpert* combined with Gentzen-style inference rules. The right level of detail in a formal proof.

We will return to some of these examples below; but it is fair to set out at this point my basis of experience, on which my opinions and observations are partly based. Of course, I have tried to be aware of the main developments in formalized mathematics, but my own work gives me the lens through which I see those developments.

3.4 Obstacles

Here is a list of obstacles to the goal of smoothly integrating logic and computation:

—Computation software doesn't track assumptions, or doesn't track them completely, and can give erroneous results.

—Computation performed by logic is inefficient.

—Justifying computations requires putting in too many steps.

—Computation performed by unverified software may be unreliable.

—Verified software may be inefficient.

4. SEVERAL APPROACHES TO THE PROBLEM

There are several ways to combine proof and computation. Barendregt [BB02] mentions three of them, to which he gave the names *believer*, *skeptic*, and *autarkic*. I here split the skeptic category in two, and list four approaches to the problem.

—*Believer*: Use unverified software and just believe it. After all your proof-checker may have a bug too. This is the approach of the *Theorema* project [Buc06], which relies on *Mathematica*, although it guards against bugs resulting from mishandling of side conditions. Also the *Analytica* prover [BCZ96] falls in this category.

—*Witness*: Use unverified software, but check the result (not every step). E. g. if you find an indefinite integral, it doesn't matter how you got it, you can check it by differentiation. This is also done in HOL Light. A good example can be found in Thomas Hales's file `Jordan/num_ext_gcd.ml`, which contains the comment:

> Now compute gcd with CAML num calculations, then check the answer in HOL-light.

Another example is discussed in § 5.3 below.

—*Skeptic*: Verify *each computation* step by step, rather than the algorithm. HOL Light does this, see the HOL Light tutorial §3.4 ([Har11], top of p. 17). For example `ARITH_RULE` appears to verify $(x+y)^2 = x^2 + 2xy + y^2$ in one step, but in reality many steps are taken; there is nowhere a formal specification for the algebraic results that are produced by `ARITH_RULE`.

—*Autarkic*[6] : Verify the algorithm, coded in the same language the proof system uses. Then you don't need to verify each result. This technique is known as "reflection".

For a similar, but slightly different, classification of ways to connect computation and proofs, see [KW08], which also contains a list of examples of systems using different methods.

4.1 Witness *vs.* skeptic

What is the difference between "witness" and "skeptic"? It is this: in the witness approach, the witness to the truth of the result may not at all explain how it was found. For example, a witness to the non-primality of a large number N could be simply a divisor of N. A witness to the fact that c is the greatest common divisor of a and b may be two integers λ and μ such that $c = \lambda a + \mu b$. In [BC01], Barendregt and Cohen report on how they got GAP to produce witnesses to the primality of certain specific large numbers N, and then used those witnesses in Coq to prove formally that N is prime. On the other hand, in the "skeptic" approach, we just formally verify the computation. The important difference is that in the skeptic

[6] Autarky is "the quality of being self-sufficient." Usually the term is applied to political states or their economic systems. Autarky exists whenever an entity can survive or continue its activities without external assistance.

approach, the computation must be done, or re-done, in the proof-checking system itself, even if it was originally done elsewhere.

What the witness and skeptic method have in common is summarized in Ronald Reagan's famous saying from the Cold War era, "Trust, but verify."

4.2 Reflection

As explained above, "reflection" is short for: write a decision procedure (or other algorithm), prove it correct, and run it, all in the same system. For the history and theory of reflection, see [Har95b]. The proof-checker Coq makes use of this technique, and other modern proof-checkers, such as Isabelle and HOL Light, are also able to do so. According to the explanation just given, the algorithm should be expressed in the formal language of the proof checker, and run in that form. However, that is usually inefficient, and if we have many time-consuming calculations to run, we would like to run them instead in the implementation language of the prover (e.g. OCaml, which compiles to C). It is possible to automatically extract OCaml code from prover-language code. Similarly with Isabelle and its implementation language ML [HAB$^+$15]: "Isabelle/HOL supports a form of computational reflection (which is used in the Flyspeck project) that allows executable terms to be exported as ML and executed, with the results of the computation re-integrated in the proof assistant as theorems."

This means that the prover (Coq or Isabelle) is trusting the code that "exports" internal terms (programs) as executable (OCaml or ML) programs, as well as the interpreter or compiler and operating system that executes that code. Since not all that software has been proved correct, we are technically back to the "believer" paradigm, rather than the "autarkic" paradigm. But we are putting our faith not in implementations of particular mathematical algorithms, but in the relatively simple and well-tested "exporting" code and compilers and operating systems. This is made more explicit in §7 of [HAB$^+$15]:

> . . . [W]e rely on the ability of Isabelle/HOL to execute closed HOL formulas by translating them automatically into a functional programming language (in this case ML), running the program, and accepting the original formula as a theorem if the execution succeeds [12]. The programming language is merely used as a fast term rewriting engine.

To remedy this step backwards from autarky to faith would require a formal proof of the correctness of the translation, and of the execution system. A first step has been taken in [HN10], who formalized the *statement* of the translation from Isabelle/HOL to ML, using an intermediate language "mini-Haskell", and gave a "standard mathematical proof" (i.e., not formalized) of the correctness.

Reflection is not a new idea; it goes back at least to NuPrl in the 1980s [How88], was discussed in [Har95a], see also [Bou97] (for Coq) and [Ber02]. The prover ACL2 [BM79] should also be mentioned, because in that case the prover's language is a subset of Common LISP, and the translation from internal code to executable code is (almost) the identity. One of its first big successes was its use in the formalization of the Fundamental Theorem of Algebra; this required the extension of reflection to partial functions [GWZ00].

Reflection is an important idea, since it was central to the big successes with the odd-order theorem and Kepler's conjecture. It is therefore a minor scandal that it breaches the autarkic paradigm by requiring faith in the translation, compiler, and operating system.

4.3 The Poincaré principle

It is under the "autarkic' method that Barendregt introduces the "Poincaré principle". By this he means that if p is a proof of $A(t)$, and the term t is computationally equivalent to s, then p should also count as a proof of $A(s)$. Thus any proof of $A(4)$ counts as a proof of $A(2+2)$ as well. For this to be legitimate, according to Barendregt, "computationally equivalent" has to mean, shown equivalent by an algorithm that has itself been proved correct. The Poincaré principle is the justification for the autarkic method. Barendregt uses the phrase "autarkic computations", but it is really the *proofs* that are autarkic, in the sense that the steps of the computations are omitted (since they are certain to be correct, and only the result is of interest). The Poincaré principle is *a priori* stronger than reflection, but in practice they are almost synonymous, since if we are going to believe that $t = s$ because a proved-correct algorithm computes it, then we are going to have little choice but to count a proof of $A(t)$ as a proof of $A(s)$. We could, of course, tack on a note that $t = s$ had been proved by a certain algorithm.

The Poincaré principle has been applied in Coq, so that proof objects do not need to incorporate (what would amount to) traces of computations. On the other hand, then we can't access directly the fast and powerful algorithms in systems like GAP. The example given above of `ARITH_RULE` in HOL Light is similar to the Poincaré principle, in that the proof object does not include the trace, but is different in that there is no proof (or even statement) of the correctness of `ARITH_RULE`.

4.4 Witness examples

Here we mention more examples of the witness approach. First, symbolic integration. Differentiation is easier than integration, so once we have somehow obtained an indefinite integral, we can "just" differentiate it to check the result. If we can formally verify the steps of differentiation (or if we use a formally verified differentiation program) then we need not care how we got the integral. This approach was demonstrated successfully in [HT98], using HOL Light and Maple. It is not as straightforward as the naive might suppose, since after we differentiate, we must still "simplify" the derivative to the original integrand. In that paper, they asked for the integral of $\sin^3 x$, and upon differentiating the answer, needed to verify the equality of two trigonometric polynomials. They used a decision procedure for that problem and called Maple again.[7] There is also the problem of verifying the hypotheses of the fundamental theorem of algebra. For example we have (according to most computer algebra systems)

$$\frac{d}{dx}\ln|x| = \frac{1}{x}$$

[7] They could have used an easier decision procedure, namely write $\sin\theta$ and $\cos\theta$ respectively as $2t/(1+t^2)$ and $(1-t^2)/(1+t^2)$, the so-called Weierstrass substitution.

but we cannot correctly conclude that

$$\int_{-1}^{1} \frac{dx}{x} = \ln|1| - \ln|-1| = 0.$$

This illustrates the fact that in the "witness" method we need to do the verifying in the prover, not the algebra system. In HOL Light, the system generates the assumption $x \neq 0$:

```
?# DIFF_CONV \x. ln(x);;
val it : thm = |- !x. &0 < x ==> ((\x. ln x) diffl inv x * &1) x
```

A second example of the use of witnesses is in finding closed forms for finite sums. An example of the type of problem we have in mind is

$$\sum_k \binom{n}{k}^2 = \binom{2n}{k}$$

The Wilf-Zeilberger (WZ) algorithm sometimes enables one to sum such series, i.e. find the right side and the proof when given the left side. We refer the reader to [PWZ96] for a thorough treatment, or to the Wikipedia article on "Wilf-Zeilberger pair" for a brief introduction. The Wilf-Zeilberger (WZ) algorithm produces a rational function as a certificate from which the proof can be recovered, so it is a natural candidate for the use of the witness method. Such experiments have recently been performed by John Harrison, and they were not quite as straightforward as originally imagined (i.e., we "just need to connect maxima to HOL Light"). Some flaws in the original proofs were exposed, whose correction was not trivial; but in the end all was well. See [Harar] for the full story.

5. HOW MANY STEPS DO WE WANT TO SEE?

One reason why the QED singularity has not yet arrived is that a mathematician wants to create a *beautiful* proof. None of today's proof assistants create beautiful proofs. That statement is deliberately blunt, in order to provoke discussion; but I believe it is fundamentally true. The presentation of proofs in a manner that pleases both the creator's eye and sense of mathematical aesthetics is important. One of the reasons for the success of TEX is that it pleases the eye. The need to present proofs in a human-readable fashion (as opposed to merely human-decipherable fashion) was recognized at the dawn of proof-checking, in AUTOMATH [dB70, dB94]. We shall not attempt to discuss the many issues to which these considerations lead, but take up in detail just *one* of those issues, namely the *level of detail* of the proof.

5.1 Too many steps?

When writing formal proofs that are intended to be human-readable, we do not want to see low-level justifications of tiny steps of a calculation. To illustrate: do we want to see answer-only steps like the following?

```
sage: factor(t^119-1)
(t - 1) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
 * (t^16 + t^15 + t^14 + t^13 + t^12 + t^11
 + t^10 + t^9 + t^8 + t^7 + t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
```

```
* (t^96 - t^95 + t^89 - t^88 + t^82 - t^81 + t^79 - t^78 +
t^75 - t^74 + t^72 - t^71 + t^68 - t^67 + t^65 - t^64
+ t^62 - t^60 + t^58 - t^57 + t^55 - t^53 + t^51 - t^50
+ t^48 - t^46 + t^45 - t^43 + t^41 - t^39 + t^38 - t^36
+ t^34 - t^32 + t^31 - t^29 + t^28 - t^25 + t^24 - t^22
+ t^21 - t^18 + t^17 - t^15 + t^14 - t^8 + t^7 - t + 1)
```

We almost certainly don't want to see that result verified by multiplying out the factorization and justifying each step from the associative and commutative laws. That is avoided in both HOL Light and Coq, but by different methods. In HOL Light, `ARITH_TAC` can be used to verify the answer, which we can find in Sage and paste in (after a little syntax editing). But we have to find the answer first, using a different program than the program in which we check it. It's possible that we might consider the above output "too many steps" already. Perhaps we just want to prove that $t^{119} - 1$ is not irreducible, i.e., we want to know that there exist two non-constant polynomials f and g whose product is $t^{119} - 1$. We don't have any hope of proving that directly in HOL Light or Coq.

5.2 Not enough steps?

Perhaps we do not need to see the results, especially if the results are obtained by an algorithm whose correctness has been formally verified. Here is an example, from Gonthier's 4-color proof. The program `check_reducibility` is proved to meet its specification `cf_reducible`. After that, any particular run of the program produces a correct result. For example (quoting the paper [Gon], p. 12)

```
Lemma cfred232 : (cfreducible (Config 11 33 37
           H 2 H 13 Y 5 H 10 H 1 H 1 Y 3 H 11 Y 4 H
           9 H 1 Y 3 H 9 Y 6 Y 1 Y 1 Y 3 Y 1 Y Y 1 Y)).
```

> [is proved] in just *two* logical steps, by applying `check_reducible_is_valid` to the concrete configuration above ... even though ... a longhand demonstration would need to go over 20 million cases. Of course the complexity does not disappear altogether–Coq 7.3.1 needs an hour to check the validity of this trivial proof.

This is the opposite of seeing too many tiny steps. This does not help us understand why the theorem is true. We have to believe it, because the algorithm has been formally verified. On the other hand, seeing twenty million steps would not help us understand why, either.

5.3 Just right: the Goldilocks option

An interesting example is given in the HOL Light tutorial [Har11], p. 61. There is a conversion rule `PURE_SOS` that "will attempt to express a polynomial as a sum of squares", and attempts to prove non-negativity of a polynomial just by expressing it as a sum of squares. For example, we can ask it whether

$$x^4 + 2x^2z + x^2 - 2xyz + 2y^2z^2 + 2yz^2 + 2z^2 - 2x + 2yz + 1 \geq 0$$

or not. The system will give the "answer", i.e. the sum of squares, but not the way by which it was found or the verification that it is equal to the input. This is a case in which the answer can be easily verified, so it would suffice if the computation had been done externally (and verified directly once obtained). Indeed that is how it is done: this functionality is not in the HOL Light core, but is considered "experimental". A comment in the code says: "This needs an external SDP solver to assist with the proof."

What if the system had just returned `true`, indicating that it had found a sum of squares equal to the input, and verified the equality, without telling us the particular sum of squares? Would we like that brevity, or not? Does it matter whether the sum of squares is 200 characters long or twenty million?

5.4 All that information is in there somewhere

As demonstrated by Claudio Sacerdoti Coen in [Coe10], it is possible to extract a more human-readable proof with steps from a Matita proof script:

In the following example H labels the fact $(x+y)^2 = x^2 + 2xy + y^2$:

obtain H

$$\begin{aligned}(x+y)^2 &= (x+y)(x+y)\\ &= x(x+y)+y(x+y) \textbf{ by } \textit{distributivity}\\ &= x^2+xy+yx+y^2 \textbf{ by } \textit{distributivity}\\ &= x^2+2xy+y^2\end{aligned}$$

done

5.5 User's choice?

It is worth noting that the problem of how much detail to display comes up not only with computer calculations, but also with hand calculations, and also with very detailed proofs. An example is the omission of the detailed calculation of the second, third, and fourth variations of area in the 560-page book [Nit89], p. 95-97, where the author says "a direct but lengthy computation (which we omit owing to lack of space)." Lack of space cannot really have been the issue; it was a choice about how much detail readers would want, or how much the author wanted to check himself.

Lamport [Lam12] proposes hypertext as the solution to this problem. The details will be hidden until the reader clicks to open one more level of detail. Programmers are already used to being able to "fold" code in this way. Mathematics is far behind, but catching up. This journal allows electronic appendices, and it's quite likely that you are reading this in electronic form, so I could not get away with omitting details for "lack of space."

6. LOGICALLY CORRECT COMPUTATION

It is a shame that the widely used and very useful computation systems (for example *Mathematica*, Maple, and SAGE) cannot be integrated into proof-checking systems. But it is impossible, since they have been designed without a sound logical foundation. Many readers know all about this, but some may not, so permit me to review the situation.

—Each line of a computation represents the right-hand side of a sequent.

—The left-hand side, which is not written, lists the assumptions in force at the moment.

—Computation rules generate new lines from old, but they have *side conditions*; that is, hypotheses that must be satisfied to apply the rule.

A computation may appear as a sequence of equations, each obtained from the previous by some symbolic manipulation. But in reality, each line depends on some assumptions that are usually not written, except sometimes when new assumptions are made or old assumptions are eliminated ("discharged"). For example, if we divide both sides of an equation by c, we must assume $c \neq 0$. Thus each line of a computation is correctly interpreted as an implication: if the assumptions at that line hold, then the equation at that line holds. The systems mentioned above were not designed with an architecture that can keep track of the assumptions accurately. There is, for example, no command to display the current assumptions. It is easily possible to derive contradictions in those computation systems by giving a couple of commands that will generate contradictory assumptions. Mathematically, we have generated an implication that a contradiction derives some false equation, which is logically OK, but the system lost track of the assumptions and only keeps the false conclusion. A number of examples of such false conclusions in commonly used symbolic computation software have been collected in [Sto91].

Therefore, the results of those systems cannot be used by proof-checkers, except in cases where the result, once obtained, can be independently checked. For example, if an equation has been solved, it may be possible to check that is a solution; but it may not be easily possible to prove that it is the only solution. If an indefinite integral has been found, it can perhaps be differentiated to verify the answer. But even these examples require that the checking be performed in a different, trusted system. In other words, the "believer" approach to computation is too dangerous, when using most symbolic computation software; we will need to use the "witness" or "skeptic" method instead.

6.1 *MathXpert* as logically correct computational software

Maybe engineers and scientists can be trusted to recognize an incorrect answer, although I personally don't like the idea of flying on jets whose engines were designed using such software. But incorrect answers are surely unacceptable in education. I am the author of *MathXpert*, which is computational software designed for education, specifically to help students learn algebra, trigonometry, and calculus. When I was designing it, I was aware of the difficulties about correctness. I therefore designed *MathXpert* to keep track of the assumptions valid at each line.

MathXpert does keep track of the assumptions, and hence, except for possible programming errors (bugs), it can never derive an incorrect result.

Assumptions are generated as "side conditions" of executing certain computational transformations. For example, if we divide by some expression, this expression must not be zero. But we want to avoid, as much as possible, generating either unnecessary assumptions (that follow from the existing ones) or contradictory assumptions. Deriving contradictory assumptions is especially to be avoided, since the assumptions are not displayed and it superficially appears that we have derived an incorrect result. At the worst we have derived a useless result, but not a wrong result.

To put this another way: *MathXpert* does use the "believer" approach to some extent, because it does not use formally verified algorithms or even provide formal proofs of its results. But because it keeps track of assumptions, the only risk in believing its results is the risk of a programming error, as opposed to a logical error due to side conditions that are not satisfied.

More recently, work has been done on the automation of assumption generation when using a reflection-based approach to computation within a proof-checker. See for example [Kal08] and [KW08]. These papers show that it is possible, at least in principle, to have your cake and eat it too: computation as in a computer algebra system, but verified to be correct.

6.2 Infer, refute, assume

In order to avoid deriving superfluous or contradictory assumptions, *MathXpert* employs an algorithm called **infer-refute-assume**. It works like this:

—We first try to infer the condition from the current assumptions.

—If that fails, we try to refute it. (In which case, the rule cannot be applied.)

—If that too fails, then we assume the required side condition and proceed.

According to the classical undecidability result of Church, the algorithms for "infer" and "refute" cannot be complete; and moreover, they must run without causing a noticeable delay; so it can very well happen that contradictory assumptions are still generated. Then the next line would appear false, but it also has generated a false assumption, so we technically have not derived a contradiction.

Example: Divide $x(x-1) = 0$ by x, generating the assumption $x \neq 0$. Then we find that the only solution of the equation is $x = 1$, which looks wrong, but technically, under the assumption $x \neq 0$ it is OK. Of course we can't have that in educational software; so various warnings are generated in *MathXpert*, and you can't divide by the unknown. But that is irrelevant to the present theoretical discussion.

MathXpert is much less powerful than *Mathematica*, Maple, and SAGE, and by design, as it purposely uses only algorithms that are taught to students. Nevertheless, the design is interesting and permits one to explore the interface between proof and computation. After publishing *MathXpert*, I used the symbolic computation code as part of some later theorem-proving projects.

The current version of *Mathematica* has "options" *GenerateConditions* and *Assumptions*, which can give the user some control over the treatment of assumptions during calculations; but *Mathematica* never makes assumptions unless explicitly told to do so, and even if it is told to do so, those assumptions do not persist to further lines of computation. Instead, when GenerateConditions is set to true, *Mathematica* returns an answer, together with the assumptions it had to make to get that answer; but internally those assumptions are then discarded rather than remaining in force.

HOL Light can sometimes generate side conditions along with answers. For example, `DIFF_CONV` differentiates $\ln x$ to $1/x$, returning a theorem with the (generated) hypothesis $x > 0$ and the answer $1/x$. See also [FGT93] and [Kal08] for other approaches to handling side conditions.

6.3 Partial functions and undefined terms

One topic that has already been discussed enough in the literature is the proper treatment of partial functions. For example, is $1/0$ undefined, or is it some (unspecified and therefore irrelevant) number, or is it some unspecified object that is not a number? It turns out that any of these answers can be made workable. In 1993, a systematic theoretical approach was given in [Far93]; although theoretical, the work supported the implementation of IMPS [FGT93]. In 2000 [GWZ00], $1/0$ is legal as a syntactic expression, but has no interpretation. There the work is not merely theoretical but worked out in Coq. See [Har11], p. 151 for the treatment of derivatives in HOL Light; it has to use a relation instead of a functional, because functions have to be total; similarly [Har09a], pp. 252–253 for complex integration. In [Kal08], a systematic approach to representing partial functions in HOL Light is given. As one often experiences in programming languages, if there is more than one way to do something, then errors are possible by mixing the ways.

6.4 Treatment of bound variables

In the standard logical languages (based on first-order logic) there is no way to represent a definite integral, limit, or an indexed sum, because these constructs involve a bound variable, while first-order logic only allows quantifiers to bind variables. In languages permitting lambda-abstraction (such as finite type theory), such bindings are represented as lambda-abstractions, and also quantifiers are reduced to lambda-abstraction, so that only one kind of variable binding is needed. This approach is used in proof-checkers based on type theory.

Whatever the underlying treatment of variable binding, the correct integration of computation with indexed sums (finite or infinite), limits, and integrals will involve the correct handling of assumptions. We show below how MathXpert deals with bound variables in integrals, sums, and limits. Even though in principle, all these kinds of bindings can be reduced to lambda-abstraction, in practice different techniques are required for reasoning with them.

Example. Determine the domain of (definedness conditions for) $\sum_{n=1}^{100} x^n$. The "obvious" way to handle this expression involves assuming that n is an integer. When the expression (tree) is traversed, we assume n is an integer between 1 and 100, while traversing that part of the tree. So $n \neq 0$ and no problem is created by the fact that 0^0 is not defined. No assumption should be generated. On the other hand, if the lower index of the sum is 0 instead of 1, then the assumption $x \neq 0$ should be generated.[8]

[8]In HOL Light, $0^0 = 1$ rather than being undefined. In calculus textbooks it is undefined, perhaps because x^y is not continuous as a function of two variables at the origin. Knuth has argued that it should be 1. Maybe there are really two functions: one defined on integers by iterated multiplication, belonging to discrete mathematics; one defined as the inverse of the logarithm, belonging to analysis, and they happen to agree except at 0^0. But no actual system (or textbook for that matter) distinguishes these two functions.

Definite integrals can be treated in a similar way: when traversing the expression tree for $\int_a^b f(x)\,dx$, we assume $a \le x \le b$.[9]

7. LIMIT COMPUTATIONS AND INFINITESIMALS

It is less obvious how one should treat variables bound by a limit. For example, consider the problem

$$\lim_{x\to 0}\left(\frac{\sin x^2}{x}+\frac{1}{x-1}\right)$$

Since x is a bound variable, no assumptions involving x are appropriate. The traditional analysis of this expression involves ϵ and δ, and two alternating quantifiers. But the spirit of computation is that it should be quantifier-free!

This difficulty can be overcome by using infinitesimals. Traversing the expression tree, when we go inside the limit, we assume x is infinitesimally near the limit point (in this case 0), but not equal to 0. Then we can infer that the denominators are not zero. The expression is therefore defined.

This approach avoids the complications of asymptotic inequalities, quantifiers, and the dependence of δ on ϵ, and is purely computational, given certain rules for computing with infinitesimals.

In *MathXpert*, the steps of computations are visible to the user, but the infer-refute-assume algorithm is invisible. Hence, for the design of *MathXpert*, it did not matter what methods were used to generate or infer assumptions, as long as the answers were correct. Normal users will never see an infinitesimal, so *MathXpert* is free to use them internally, as long as it gets correct answers. The question then arose, how can we compute with infinitesimals, and guarantee that the answers are correct?

We consider a second example of a limit computation. To calculate the derivative of $\sqrt{x}$ from the definition, we consider

$$\lim_{h\to 0}\frac{\sqrt{x+h}-\sqrt{x}}{h}.$$

[9] A referee remarked that it suffices to assume $a < x < b$, saying that the behavior at the endpoints is irrelevant to the value of the integral. This delicate distinction, and other similar distinctions about endpoints, can lead to trouble. For example: is $\sqrt{x}$ differentiable on $[0,1]$? Answer: In most of the world, no, because the function is not differentiable at the endpoint 0 (only its derivative from the right exists). (In France, it is differentiable, because only difference quotients of points in the domain are considered. There are other particular features of French mathematics as well.) Now, we ask whether

$$\int_0^1 \frac{d}{dx}\sqrt{x}\,dx = \sqrt{1}-\sqrt{0}$$

holds. In the US (and perhaps everywhere outside France) it does not, since the Riemann integral on the left does not exist, because the integrand is not defined on the closed interval. (And indeed, *MathXpert* rejects this integral as undefined.) In *MathXpert*, a definite integral is defined if the integrand is defined on the (finite, closed) interval of integration, because you can't define a non-integrable function in the system. That scheme would break if we weakened the assumption as suggested.

The usual approach is to "rationalize the numerator" by multiplying numerator and denominator both by $\sqrt{x+h}+\sqrt{x}$. Then we get an expression involving

$$(\sqrt{x+h})^2 - (\sqrt{x})^2$$

Now observe

—We need to rewrite $(\sqrt{x+h})^2$ as $x+h$.

—That is only legal when $x+h \geq 0$, but we should not generate an assumption depending on the bound variable h.

—The proper assumption to generate is $x > 0$. Note $\sqrt{x}$ is not differentiable at 0 (not even from one side).

How can that be done by a general-purpose algorithm? The answer is that $x+h \geq 0$ for all infinitesimal h if and only if $x > 0$. The implicit universal quantifier over infinitesimals is expressed in the computation rule that rewrites $x+h \geq 0$ as $x > 0$.

This approach eliminates quantifiers in favor of computation. But, one may ask how this can be useful for QED, since most proof-checkers do not work in a theory that allows infinitesimals. The answer might lie in the observation that the trick is coming up with the right new assumption. Once we have it, even if we pulled it out of a hat with a flourish of a multi-colored scarf, we may be able to prove the theorem using the new assumption, within a proof-checker using semi-automated tactics. Thus an external program could be used, not only to generate integrals, solve differential equations, factor integers, and express rational functions as sums of squares, but also to generate appropriate conditions using calculations with infinitesimals, and the results verified within the checker, regardless of how obtained.

7.1 Infinitesimal elimination and its correctness

When expression traversal enters a limit as $x \to a$, *MathXpert* introduces an infinitesimal (variable) $h = x - a$. This infinitesimal can be used in infer-refute-assume, while still inside the limit, but neither the infinitesimal h nor the bound limit variable x can appear in any generated assumption. Thus the infinitesimal must be "eliminated" before we finish evaluating the limit. This is to be accomplished by an "infinitesimal elimination algorithm."

Although the results of infinitesimal elimination might be useful for QED even if produced by magic, to ensure that *MathXpert* gives only correct results, I needed to be sure that was also true of infinitesimal elimination. Therefore I needed to specify the infinitesimal elimination algorithm, and prove its correctness. (To avoid any misunderstanding: we are talking about an ordinary, informal correctness proof, not a computer-checked formal correctness proof.)

Such a correctness proof must make use of some defined semantics. The semantics used is "interval semantics": A formula $\phi(\alpha)$ involving a nonstandard variable α means that $\phi(x)$ is true for all x in every sufficiently small punctured neighborhood of the limit point a. We use one-sided neighborhoods for one-sided limits. There are obvious modifications for limits at infinity.

In [Bee95] I specified the infinitesimal elimination algorithm and proved its correctness with respect to this semantics, at least for one infinitesimal: nested limits have not been treated (and are not in-scope for MathXpert, or for freshman calculus). Obviously an adaptation of this method to support professional mathematics would require treating nested limits.

It is interesting to note that in the HOL Light library, differentiation is not developed via limits of difference quotients; instead it is developed directly from an epsilon and delta definition. See §19.3, p. 151 of [Har11].

7.2 Remarks on limit problems

Before the infinitesimal elimination algorithm, early *MathXpert* used a form of second-order logic, similar to interval semantics, but coded directly in C. The infinitesimal elimination algorithms saved six or seven thousand lines of code.

Students have great difficulties understanding the semantics of limits. Epsilon-delta semantics was not properly understood by even one of hundreds of students to whom I lectured on the subject, as exam results showed. Such understanding of limits as some students do achieve is based (in effect) on a direct axiomatization of the undefined notion of limit. In other words, they learn certain laws to manipulate limits and work with those computation rules rather than with logic. In my last years of teaching calculus, I gave up the pretense of teaching the alternating-quantifier epsilon-delta definition, and relied on computation rules and pictures. I think mathematicians too prefer to work with computation rules; logic is a last resort.

7.3 Asymptotic expansions

In the last two decades, significant effort has been devoted both to the theory and the implementation of asymptotic methods, i.e. various kinds of series and "transseries" expansions of functions at infinity. The basic class of functions to start with is the exp-log functions, the least field of functions (defined on some positive half-line) closed under the exponential and logarithm functions. Shackell [Sha90] gave an algorithm for computing the limit at infinity of any such function, and that algorithm was implemented in *Maple* [Gru96]. This class of functions is not closed under inverse functions. To handle inverse functions it was necessary to develop more esoteric kinds of expansions. See [SS99] for more information and references to the symbolic computation literature. It seems that all the work on asymptotic expansions works on various "Hardy fields" of functions; these functions all have limits (finite or infinite) at infinity, so trigonometric functions are not included; moreover only real-valued functions are considered.

We remark that although systematic asymptotic expansion algorithms are sufficient to compute limits, they are not always necessary. Salvy and Shackell (*op. cit.*) give the example

$$\exp(x^{-1} + e^{-x}) - \exp(x^{-1})$$

whose asymptotic expansion is difficult to compute. But a good calculus student should have no trouble computing its limit at ∞ (and *MathXpert* has no trouble either).

The question arises whether this body of work is relevant to QED. As far as I know, there is no natural "certificate" for an asymptotic expansion. That is, there is no simple and reliable way to verify (even given additional data) that a given expression really is a correct asymptotic expansion of a given function. That makes it difficult to use the "witness" method with asymptotic expansions. Since these methods are quite difficult to program, they have not been re-implemented autarkicly, and nobody wants to trust without verification; so they have never yet, to my knowledge, been used in formal proofs. That is too bad, because there certainly have been formal proofs that used asymptotic limits, e.g. the proof of the prime number theorem; so these have had to use independent methods.

One might consider asymptotic expansions as "generalized infinitesimals". This is an intriguing idea (suggested by one of the referees). The different terms of the expansion then correspond to different "orders" of infinitesimal. In this analogy the role of the infinitesimal elimination algorithm is taken over by conservativity theorems to the effect that limits calculated using asymptotic expansions are correct. That is all very well, but if we ask *Mathematica* to calculate

$$\lim_{h \to 0} \frac{\sqrt{x+h} - \sqrt{x}}{h}$$

we get the answer $1/(2\sqrt{x})$, without any assumption $x > 0$ or even $x \geq 0$. This shows that there is something yet to do before asymptotic expansions are useful in QED.

8. INFINITE SERIES

In the US, there are the so-called Advanced Placement (AP) tests in calculus. High-school students can earn college credit by passing these tests. There are two of these: Calculus AB and Calculus BC. The most difficult topic in Calculus BC is infinite series. Nested (double) series are not considered. There are three things calculus students are expected to learn:

—To know certain basic series and to be able to use them to sum other series.

—To calculate the first few terms of a Taylor series by differentiating the given function.

—The classical convergence tests: ratio test, root test, alternating series with decreasing terms test, and possibly some fancier ones.

Infinite series is the only topic on the American AP Calculus exam that was not supported by MathXpert originally. Since June 2013 it is supported (to the first-year college level only).

Clearly any serious contender for a QED system will have to handle infinite series easily and correctly, in such a way that normal mathematical computations with series do not require jumping through logical hoops, but also that correctness is ensured. Infinite series are widespread in mathematics and the failure to work with them in a natural way will hold up the arrival of the "QED singularity". Let us list some of the difficulties in meeting this requirement.

—We want to write series down without knowing whether or not they converge (i.e., are defined).

—Many computational operations have complicated side conditions: rearrange order of terms, regroup terms, differentiate or integrate term-by-term, multiply or divide.

—The side conditions often involve two quantifiers: absolute convergence, for example.
—The hypotheses for the convergence tests reduce to asymptotic inequalities.

By an *asymptotic inequality* I mean an equality that holds "for all sufficiently large x", or "for all sufficiently large n", or "for all sufficiently small x", or "for all x sufficiently near to a". It takes two quantifiers to express this concept in first-order logic. For example,

$$\exists N \forall m \geq N(2^m > m^3)$$

expresses that asymptotically, $2^m > m^3$.

The three most important "convergence tests" are as follows. From their statements, one can see that computing with asymptotic inequalities is the key.

—*Comparison test.* If asymptotically $|b_n| \leq |a_n|$ and $\sum_{n=1}^{\infty} a_n$ converges, then so does $\sum_{n=1}^{\infty} b_n$.
—*Divergence test.* If asymptotically $|b_n| \leq |a_n|$ and $\sum_{n=1}^{\infty} b_n$ diverges, then so does $\sum_{n=1}^{\infty} b_n$.
—*Root test.* If asymptotically $a_n \leq a^n < 1$ then $\sum_{n=1}^{\infty} a_n$ converges.

The essential point is that the convergence tests all reduce to asymptotic inequalities. Hence, if we want to use infinite series in a logically correct manner, the ability to decide asymptotic inequalities correctly (and verifiably) will be quite useful. A start to this project was made in *MathXpert*, but it is nowhere near adequate to support "real mathematics," although it works for a good fragment of freshman calculus. But as described in §7.3, the symbolic computation community now knows how to compute asymptotic expansions for all exp-log functions and their inverses. An important point is that these asymptotic expansions enable one to settle inequalities algorithmically, since the difference of two functions also has an asymptotic expansion and approaches a (calculable) limit.

The work so far on asymptotic expansions appears to be limited to real-valued functions. Hence Fourier series and other trigonometric series are not covered; and in general complex-valued series are used throughout number theory and analysis. Even for real-valued functions, it has yet to be applied in proof-checking, for the reasons sketched in §7.3.

8.1 A Feynman story

When I was an undergraduate at Caltech, Richard Feynman told us never to worry about whether an infinite series converges. He said,

> *Just add it up!* After you've got the answer, there's plenty of time to worry about whether it converges.

At the time I did not realize that Feynman had earned his Nobel Prize for finding a way to "just add up" some infinite series that do not converge; his method was called "renormalization." Feynman wouldn't have appreciated the goals of QED.

Compare Feynman's remark that "there's plenty of time to worry about convergence after you have the answer" with the technique of getting an answer externally and then verifying it. Even if you have the answer for an infinite series in hand, it is often not easy to verify it, by contrast with an integral or expression for a rational function as a sum of squares.

8.2 A John Harrison story

Here is a quotation from his paper [Har09a], p. 257, on formalizing the analytic proof of the prime number theorem. In the quotation, he is talking about a single step in the proof by Newman that he is formalizing.

$$f(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}\left(\sum_{p\leq n} \frac{\log p}{p}\right) = \sum_{p} \frac{\log p}{p}\left[\sum_{n\geq p} \frac{1}{n^z}\right]$$

> The implied equality between these different orders of summation of the double series, simply stated without comment by Newman, also occurs in our proofs, and *we needed a 116-line proof script to justify it* [emphasis added].

This shows how far we have yet to go towards the goal of a QED system that mathematicians would gladly use. A step that a human mathematician takes without comment, and about which readers feel no doubt as to its correctness, should not require 116 additional steps to formalize. No criticism is meant here–the system and its use are state-of-the-art. My intention is simply to point out the difficulties yet to be overcome.

What exactly are the "difficulties to be overcome" that are highlighted by this example? Certainly one of them is the matter of the "rich enough library." Newman's "library" included theorems allowing for the interchange of order of integration. HOL Light's library perhaps did not (at that time); or perhaps the 116 lines were needed to verify the hypotheses of those theorems. A human "sees" that the terms are all positive, all the series are convergent, hence absolutely convergent, etc. But that might take 161 lines to prove in the class where these theorems are first introduced. There are thus two difficulties here: (i) the "accumulation of knowledge", and (ii) the "ease of applying well-known knowledge."

9. MATHEMATICAL INDUCTION, OTTER-λ, AND QED

Another kind of proof in which computation often mixes with logic is in proofs of some (computational) formula by mathematical induction. These are often used to introduce mathematics students to the concept of proof. A typical example is the formula for the sum of the first n squares,

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)}{2}$$

The "proof plan" for mathematical induction reduces this to two algebra problems (base case and induction step). The induction step is equational reasoning with an equational assumption. "Equational reasoning" often reduces to computation, and of course in the example, it does. Today's proof-checkers have no serious difficulties coping with proofs by induction. The example just mentioned occurs as the first theorem in file `fourier.ml` distributed with HOL Light. Its proof script is five lines long; the first two lines name and state the theorem, and the last three lines say, in HOL Light syntax, "first apply induction to reduce the problem to the base case and induction step; then rewrite those cases by algebra and arithmetic." The actual script is explicit about the algorithms to be used for rewriting.

Nevertheless, there may be some lessons for QED to be learned from my experiment in combining *MathXpert* with a resolution theorem-prover. The program in question is called Otter-λ, because it is based on adding (a certain kind of) second-order unification to the theorem-prover OTTER [McC]. With second-order unification, you can give Otter-λ a second-order axiom (schema) like induction, and it can try to find an instance of induction that will work. It did well enough at this to do all the interesting examples of proofs by induction that Bundy's provers [Bun01] could do.

The relevance to QED is that, in order to do simple examples of induction, like the one above, Otter needed high-school algebra. I linked Otter-λ to MathXpert (literally, linked the source codes producing a single executable). Otter-λ passed MathXpert the assumptions from the proof, and retrieved the result (including any changed assumptions). Unless MathXpert has a bug, this result will be correct, so I just let Otter-λ take such steps, accepting the result without further verification.

This combination (Otter-λ plus MathXpert) can indeed do high-school proofs by induction–a successful proof of concept. Is this relevant to QED? There are two points of relevance: The use of second-order unification to find the right instance of induction, and the part about linking symbolic computation code to a resolution theorem-prover may have some relevance. The proofs produced look like resolution proofs, with steps taken externally justified by "simplify." Trusting those proofs relies on trusting that MathXpert had no relevant bugs. But because MathXpert is designed to be logically sound, there was no other soundness risk.

The resulting proofs therefore have a level of trustworthiness less than what one would get from Coq or HOL Light, but greater than what one would get from Maple, Sage, or *Mathematica*. If the aim of formalization is to find errors in putative proofs, this level of trustworthiness could be valuable. If the aim is to achieve absolute certainty, it is not sufficient.

10. THE *WEIERSTRASS* PROVER

An earlier attempt to link computational code from *MathXpert* to a theorem-prover was called *Weierstrass*, because originally I used it for epsilon-delta proofs in the style introduced by Weierstrass. The theorem-prover used a form of backwards search for a Gentzen sequent-calculus proof, and added new inference rules that incorporated calculation rules. The code to implement these rules came from *MathXpert*.

After enough tinkering the program could automatically find a proof that e is irrational [Bee01]. The irrationality of $e = \sum \frac{1}{n!}$ is a famous result. It is not trivial, but not terribly difficult either. It mixes logic and computation very thoroughly. Although the automatic finding of the proof is not relevant to QED, other aspects of this proof are very relevant, as they illustrate several of the difficulties in combining proof and calculation. The proof involves various mathematical techniques:

—inequalities

—bounds on infinite series
—type distinctions (between real and natural numbers)
—simplification of expressions involving factorials
—summing an infinite geometrical series
—proving a formula $2^{n-q-1}(q+1)! \leq n!$ by induction on n.

From the point of view of the sociology of the QED project, the reception of this work is interesting. I gave a couple of mathematics colloquium talks about this proof in 2000 or 2001. The questions asked indicated that (at least some) people were very willing to believe that soon computers would be proving much more complicated theorems, perhaps new ones. I was surprised, since it was obvious to me that this result had been achieved as a "dog and pony show" and one could not hope, e.g., to prove the irrationality of Euler's γ, which is an open problem.

Another aspect of this experiment that is relevant to QED concerns the presentation of the computer proof. By contrast to the difficult-to-read resolution proofs produced by OTTER, *Weierstrass* produced a TEX version of its proof, a structured proof that Leslie Lamport would like, with the structure shown by indentations. Some steps are justified by "simplification." Not every internal detail of the calculations so summarized is shown. In general, I believe people will be happy to accept "simplification" as a justification, if they trust the simplifier, and the missing steps are not too many. If there are 20 million omitted steps, then the trust in the simplifier must be correspondingly great.

Ten years after the *Weierstrass* proof of the irrationality of e, Jesse Bingham [Bin11] used HOL Light to formalize a proof of the transcendence of e, which is more difficult than the irrationality of e. His 3000-line proof is distributed with HOL Light. Curiously, it uses only finite series, not infinite series.

11. SUMMARY AND CONCLUSION

The difficulties in incorporating computations into proofs include

—the use (or avoidance) of undefined terms;
—the tracking of assumptions needed;
—how to handle assumptions when they are needed;
—whether to trust external code or not;
—in using reflection, whether to compile proved-correct internal code to external code for the advantage of speed.

After about 25 years of experience, the community has found workable solutions to all these problems. The only remaining small leak in our boat is the last point, since running translated code exposes us to possible errors in the translator, compiler, or operating system.

References

[BB02] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28:321–336, 2002.

[BC01] Henk Barendregt and Arjeh M. Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *Journal of Symbolic Computation*, 32:3–22, 2001.

[BCZ96] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica — an experiment in combining theorem proving and symbolic computation. In *Artificial Intelligence and Symbolic Mathematical Computation*, volume 1138 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 1996.

[Bee89a] Michael Beeson. Logic and computation in Mathpert: An expert system for learning mathematics. In *Computers and Mathematics '89*, pages 202–214. Springer-Verlag, Berlin Heidelberg New York, 1989. Mathpert was the original name of MathXpert.

[Bee89b] Michael Beeson. Mathpert: An expert system for learning mathematics. In *Proceedings of the Conference on Technology in Collegiate Mathematics Education, Columbus, Ohio, October, 1988*, pages 9–14. Addison-Wesley, 1989. Mathpert was the original name of MathXpert.

[Bee89c] Michael Beeson. The user model in Mathpert, an expert system for learning mathematics. In Bierman, Breuker, and Sandberg, editors, *Artificial Intelligence and Education '89*, pages 9–14, Amsterdam, 1989. IOS. Mathpert was the original name of MathXpert.

[Bee95] Michael Beeson. Using nonstandard analysis to verify the correctness of computations. *International Journal of Foundations of Computer Science*, 6(3):299–338, 1995.

[Bee97] Michael Beeson. MathXpert Calculus Assistant, 1997. The current version of this software is sold by Help With Math.

[Bee01] Michael Beeson. Automatic generation of a proof of the irrationality of e. *Journal of Symbolic Computation*, 32(4):333–349, 2001.

[Bee06] Michael Beeson. Mathematical induction in Otter-lambda. *Journal of Automated Reasoning*, 36(4):311–344, 2006.

[Ber02] Stefan Berghofer. Program extraction in simply-typed higher order logic. In Jean-Christophe Filliatre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, (TYPES 2002)*, number 2646 in Lecture Notes in Computer Science, pages 21–38. Springer, 2002.

[Bin11] Jesse Bingham. Formalizing a proof that e is transcendental. *Journal of Formalized Reasoning*, 4(1):71–84, 2011.

[BM79] Robert Boyer and J. Strother Moore. *A computational logic*. Academic Press, New York, 1979.

[Boo06] A. R. Booker. Turing and the Riemann hypothesis. *Notices of the American Mathematical Society*, 53(10):1208—1211, 2006.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takayuso Ito, editors, *TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer-Verlag, 1997.

[Buc06] Bruno Buchberger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470–504, December 2006.

[Bun01] Alan Bundy. The automation of proof by mathematical induction. In Alan Robinson and Andre Voronkov, editors, *andbook of Automated Reasoning, Volume II*, chapter 13. MIT Press, 2001.

[BW05] Michael Beeson and Freek Wiedijk. The meaning of infinity in calculus and computer algebra systems. *Journal of Symbolic Computation*, 39(5):523–538, 2005.

[Coe10] Claudio Sacerdoti Coen. Declarative representation of proof terms. *Journal of Automated Reasoning*, 44:25–52, 2010.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M Laudet, Daniel Lacombe, and M Schuetzenberger, editors, *Symposium on Automatic Demonstration, INRIA, Versailles*, volume 125 of *Lecture Notes in Computer Science*, pages 29–61, Berlin, 1970. Springer-Verlag. reprinted in [**?**].

[dB94] N. G. de Bruijn. Reflections on automath. In R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors, *Selected Papers on Automath*, number 133 in Studies in Logic, pages 201–228. North-Holland, Amsterdam, 1994.

[Far93] William M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993.

[GAA$^+$13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, Francois Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive theorem proving, 4th international conference, ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[Geu09] Hermann Geuvers. Proof assistants: History, ideas and future. *Sādhaā*, 34(1):3–25, 2009.

[Gon] Georges Gonthier. A computer-checked proof of the four colour theorem. http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, December 2008.

[Gru96] D. Gruntz. *On computing limits in a symbolic manipulation system.* PhD thesis, ETH Zurich, 1996.

[GWZ00] Hermann Geuvers, Freek Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In M. Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000, (Portland)*, volume 1689 of *Lecture Notes in Computer Science*, pages 162–178, Berlin, 2000. Springer.

[HAB$^+$15] Thomas Hales, Mark Adams, Gertrude Bauer, Dan Tat Dat, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Nguyen Tat Thang, Nguyen Quang Truong, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Ta Thi Hoai An, Tran Nam Trung, Trieu Thi Diep, Josef Urban, Vu Khac Ky, and Roland Zumkeller. A formal proof of the Kepler conjecture. http://arxiv.org/pdf/1501.02155.pdf, January 2015.

[Har95a] John Harrison. Meta theory and reflection in theorem proving: a survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.

[Har95b] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`.

[Har09a] John Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43:243–261, 2009.

[Har09b] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, UK, 2009.

[Har11] John Harrison. Hol light tutorial (for version 2.20). on line, January 2011.

[Harar] John Harrison. Formal proofs of hypergeometric sums. *Journal of Automated Reasoning*, to appear.

[HN10] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010.

[How88] Doug Howe. Computation meta theory in Nuprl. In E. Lusk and R. Overbeek, editors, *Proceedings of the Ninth International Conference of Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 238–257. Springer-Verlag, 1988.

[HT98] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[Kal08] Cezary Kaliszyk. Automating side conditions in formalized partial functions. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28-August 1, 2008, Proceedings*, volume 5144 of *Lecture Notes in Computer Science*. Springer, 2008.

[KW08] Cezary Kaliszyk and Freek Wiedijk. Merging procedural and declarative proof. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Proc. of the Types for Proofs and Programs International Conference (TYPES'08)*, volume 5497 of *LNCS*, pages 203–219. Springer Verlag, 2008.

[Lam95] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, September 1995.

[Lam12] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, pages doi:10.1007/s11784–012–0071–6, March 2012.

[McC] William McCune. OTTER. http://www.mcs.anl.gov/AR/otter.

[Nit89] Johannes C. C. Nitsche. *Lectures on Minimal Surfaces*, volume 1. Cambridge University Press, Cambridge, UK, 1989.

[OSR92] Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science.* Springer, Berlin Heidelberg, 1992.

[PWZ96] Marko Petkovvsek, Herbert S. Wilf, and Doron Zeilberger. *A=B.* A. K. Peters, Wellesley, MA, 1996.

[Sha90] John Shackell. Growth estimates for exp-log functions. *Journal of Symbolic Computation*, 10:611–632, 1990.

[Sha09] Natarajan Shankar. Automated deduction for verification. *ACM Computing Surveys*, 41(4), October 2009.

[SS99] Bruno Salvy and John Shackell. Symbolic asymptotics: Multiseries of inverse functions. *Journal of Symbolic Computation*, pages 543–563, 1999.

[Sto91] David R. Stoutemyer. Crimes and misdemeanors in the computer algebra trade. *Notices of the American Mathematical Society*, 38(7):778–785, 1991.

[Tit51] E. C. Titchmarsch. *The theory of the Riemann zeta-function.* Clarendon Press, Oxford, 1951.

[Wen99] Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, Andre Hirschowitz, Christing Paulin, and Laurent Théry, editors, *TPHOLs '99 Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer-Verlag, 1999.

[Wie99] Freek Wiedijk. Formalizing 100 theorems . http://www.cs.ru.nl/F.Wiedijk/100/index.html, 1999.

[Wie04] Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003, Torino, Italy*, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2004.

[Wie08] Freek Wiedijk. Formal proof–getting started. *Notices of the American Mathematical Society*, 55(11):1408–1414, 2008.