# How to Believe a Machine-Checked Proof*

Robert Pollack
rap@dcs.ed.ac.uk

September 4, 1996

## 1 Introduction

Suppose I say to you "Here is a machine-checked proof of Fermat's last theorem". How can you use my putative machine-checked proof as evidence for belief in Fermat's last theorem? I start from the position that you must have some personal experience of understanding to attain belief, and to have this experience you must engage your intuition and perhaps other mental processes which it is not possible to formalize.

By machine-checked proof I mean an explicit formal derivation in some given formal system; I am talking about derivability, not about truth. This excludes computer mathematics systems that don't claim to check a specified formal system. Further, I want to talk about *actually* believing an *actual* formal proof, not about formal proofs in principle: to be interesting, any approach to this problem must be feasible. You might try to read my proof, just as you would a proof in a journal; however, with the current state of the art, this proof will surely be too long for you to have confidence that you have read and understood it. In this paper I attempt a technological approach for reducing the problem of belief in a formal proof to the same psychological and philosophical issues as for belief in a conventional proof in a mathematics journal. The approach is not entirely successful (see section 3.2) but is satisfactory in practice, and I argue it is the best we can do.

In the rest of this introduction I outline the approach and mention related work. In following sections I discuss what we expect from a proof, add details to the approach, addressing many of the problems that arise, and concentrate on what I believe is the primary technical problem: expressiveness and feasibility for checking of formal systems and representations of mathematical notions.

**Acknowledgements** Almost everybody I know has some interesting opinion on these matters; I thank everyone I have discussed it with.

*Submitted to *Twenty Five Years of Constructive Type Theory: Proceedings of the Venice Meeting.* Comments are welcome.

## 1.1 Outline of the approach

My approach is to separate the problem of how to believe a theorem when given only a putative formal proof into two subproblems: deciding whether the putative formal proof is really a derivation in the given formal system (a formal question), and deciding if what it proves really has the informal meaning claimed for it (an informal question).

To be be more specific about the problem, I give you a putative proof of Fermat's last theorem formalized in a given logic. Assume it is a logic that you believe is consistent, and appropriate for Fermat's last theorem. The "thing" I give you is some computer files; there may be questions about the physical representation of the files, how to read them, the abstract language the proof is written in (i.e. how to parse the files as a purported proof), and correctness of the hardware and software to do this parsing. Ignore all of these for the moment (see section 4.4).

**Is it a theorem?** Is the putative formal proof really a derivation in the given formal system? This is a formal question; it can be answered by a machine. The difficulty is how can you believe the machine's answer; i.e. do you trust the the proof-checking program, the compiler it was processed by, the operating system supporting it, the hardware, etc. This is usually taken to be the crux of the problem of believing machine-checked theorems.

To address this problem, you can independently check the putative proof using a simple proof checking program for the given logic, written in some meta-language, e.g. a programming language or a logical framework. In order to believe the putative derivation is correct, you must believe this simple proof checker is correct. I have in mind a proof checking program that only checks explicit derivations in the given logic, verifying that each step in the derivation actually follows by a specified rule of the logic; no heuristics, decision procedures, or proof search is required for checking, although these techniques may have been used in constructing the proof in the first place. Such a simple proof checking program is a formal object that is much smaller and easier to understand than almost any non-trivial formal proof (and most informal proofs), so this approach greatly simplifies the problem[1]. I am not suggesting such a simple proof checker be used to discover or construct formal proofs, only to check proofs constructed with more user-friendly tools.

Since my goal is to reduce believing a formal proof to the same issues as believing a conventional proof, my favored technique for believing the correctness of a simple proof checker is to read and understand the program in the context of your knowledge of the logic being checked and the semantics of the meta-language in which the checker is written. We should use available techniques to make this task as simple as possible; e.g. using LCF style to implement the simple checker, so very few lines of code are critical for its correctness, or using an executable specification of the logic in a logical framework or generic proof checker. If the logic is simple enough, and the meta-language has a simple enough and precise enough semantics, then the sum total of what you are required to read and understand is neither

[1] J Moore once commented that, until Shankar did his NQTHM proof of Gödel's Incompleteness Theorem, if you wanted to believe everything checked by NQTHM, you would do better to read all the proofs than to read the code of NQTHM, as the code was longer than all the proofs.

longer nor more difficult to understand than a conventional proof, and belief in the putative derivation is attained through your personal experience of understanding. This approach differs from the conventional one, of reading and understanding the proof yourself, only in being indirect, a kind of cut rule at the meta-level of the readers' understanding; rather than using personal intuition to believe a proof, you use personal intuition to believe a mechanism to check proofs. If you have understood a simple proof checker, and believe it correctly checks derivations in the given formal system, then you have reason to believe the correctness of a derivation it accepts.

You can also use other techniques to gain confidence in the simple proof checker; e.g. you can get the opinion of an expert on simple checkers, or use a publicly available checker from a library of checkers that are refereed by several experts, and that have high confidence from being used to check previous examples. If a few logics become accepted as appropriate for formalization, and large bodies of formal mathematics are developed in these few logics, then only a few independently refereed simple proof checkers are necessary, even though users may prefer many different tools for constructing proofs in the first place. These techniques are similar to those used to gain confidence in conventional proofs, and seem to be even more reliable in the present approach. You can even formally verify that the proof checking program meets the specification of the formal system it claims to check, given some semantics of the meta-language.

**What theorem is it?** Having believed that my putative proof is actually a derivation in the claimed formal system, you ask "does it prove Fermats last theorem?" Is the meaning of the formal theorem really what is claimed? This is an informal question; it cannot be answered by a machine, as one side of the "equivalence" is informal[2]. You must bridge this fundamental gap by using your own understanding; you will want to consider the formal theorem in light of your understanding of the formal system (the logic) being used, any assumptions used in the proof, and all the definitions used in stating the formal theorem. The difficulty is how can you read the formal proof to decide its meaning for yourself, given the size and obscure presentation of many formal proofs. This issue is sometimes overlooked in discussions of the usefulness of formal proof.

You don't need to read the entire proof in order to believe the theorem. Given that you have reason to believe the putative proof is a correct derivation in the given logic (by independent checking as discussed above), only the outstanding assumptions, the formal statement of the theorem, and the definitions used hereditarily in stating the formal theorem must be read. Although the formal proof, perhaps partially generated by machine, may contain many definitions, lemmas and local assumptions, these need not be read, as we trust that they are all correctly formulated and used as allowed by the logic, since they are checked by our simple, trusted proof checker.

[2]In the (distant?) future all this work of bridging the informal-formal gap may have been done; i.e. the gap is bridged at some foundational level. When all mathematics is done formally, in a few accepted logics, using accepted formal definitions for the basic mathematical notions, then new definitions and conjectures will be stated in terms of already formal notions, and no question will arise about whether some string of symbols is really Fermats last theorem.

You can use the trusted simple proof checker to print out the undischarged assumptions, the statement of the formal theorem, and the definitions used in stating this theorem. (It is necessary to trust the tool that shows us the assumptions used in the proof, as the formal proof is too big to read for ourselves and check that these really are all the assumptions used.) Then it is up you, using your own understanding of the formal system, the outstanding assumptions and the statement of the formal theorem, to decide if it means what is informally claimed. But this is anyway a subtask of believing a conventional proof in a textbook or journal; so this second subproblem of believing a formal proof is no more difficult than the corresponding aspect of believing a conventional proof.

**A weak point.** I claim to reduce belief in a proof to belief in the means of checking it, and that the means of checking it can be believed by processes of intellect of the same kind as used for believing conventional proofs, since a simple proof checker is a formal object that is simpler than a conventional proof. But this view of a proof checker as a simple thing depends on simple and believable semantics of the meta-language in which the proof checker is written. The problem is that I can't claim that believing the operation of a programming language and all its underlying software and hardware is a simple thing. Everybody knows that Unix has bugs, as does New Jersey SML; it would be foolish to think otherwise.

This is a serious philosophical criticism, although most computer scientists won't find it serious in practice. It is not a common problem that compiler, operating system or hardware bugs cause a proof checker to say "yes" when it means "no". Further, repeatability of such an error over several independent runs, using different compilers (for the same language), running on different operating system/cpu platforms is inconcievable. This is discussed further in section 3.2

## 1.2 Related work

The prototypical paper on this topic is [DLP79], where it is argued that "Mathematical proofs increase our confidence in the truth of mathematical statements only after they have been subjected to the social mechanisms of the mathematical community", whereas machine-checked proofs "...cannot acquire credibility gradually, as a mathematical theorem does; one either believes them blindly as a pure act of faith, or not at all." I mostly agree with the first statement, but completely disagree with the second, and present the means for social mechanisms of the mathematical community to operate on formal proofs, namely independent checking.

Independent checking is not a new idea. It has been discussed in terms of increasing confidence in computer-based enumerative search (e.g. [Lam90], see section 2.2 below). It is considered the standard approach in tasks such as computing many digits of $\pi$. A proposal similar to the present paper ("verify the proofs rather than the programs which produce them") is made, in less detail, in [Sla94].

There has been much discussion recently of the possibility and desirability of carrying out formal mathematics [Boy94, Har96]. The present paper addresses many points

necessary to carry out such a program.

# 2 What can we expect from a proof?

All belief held by a human being is based on that person's experiences of understanding, and all experiences of understanding derive from perception of evidence. In certain areas of discourse, like law and mathematics, there are more or less precise rules about what kind of perceptions should be accepted as evidence. No matter how precise the rules about evidence, it still depends on some operations of human consciousness to apply the rules and experience understanding or not. Without claiming anything deep about operations of human consciousness, there are some things we can say about human beliefs.

## 2.1 Truth

> If God has mathematics of his own that needs to be done, let him do it himself.
>
> Errett Bishop [Bis67]

We have no access to *truth* in either formal or informal mathematics; the way things "really are", with the natural numbers for example. For me this is neither a deep claim nor a serious limitation on our practice of mathematics; we have only our various notations for mathematical objects, and reasoning with and about them is the business of mathematics. Thus I will not discuss the truth of Fermat's last theorem, but will suggest how to approach the question of whether Peano Arithmetic (or ZF set theory, or the Calculus of Constructions, ...) *proves* Fermat's theorem for some given definition of "natural number", "addition", etc. Some readers may want to carry on the argument from provability to truth themselves, but I don't see any difference between informal and formal proof in this respect.

## 2.2 Certainty

> At the moment you find an error, your brain may disappear because of the Heisenberg uncertainty principle, and be replaced by a new brain that thinks the proof is correct.
>
> Leonid A. Levin, quoted in [Hor93]

Everyone has had the experience of understanding and believing a proof at one time, and seeing an error in it at a later time. After such an experience, you must accept that it might happen again. Therefore the notion of certainty, like that of truth, is not of particular relevance to human knowledge. This view is not always accepted in conventional mathematics, where practitioners often talk of the certainty of a (correct?) proof. For example, a paper [Lam90] about reliability of enumerative searches done by computer cautions "Notice that the assertion of correctness is not *absolute*, but only *nearly certain*, which is a special characteristic of a computer-based result." (Author's emphasis, but the

underlining is mine, to contrast with my belief that no knowledge is absolute.) It doesn't seem credible that any useful analysis of the probability of error caused by software bugs in big calculations is possible, and I don't think this is what readers of proofs are looking for. What's important is how knowledgeable people working in a field attain belief. Lam is commenting on his own proof that there do not exist any finite projective planes of order 10, which uses several thousand hours of supercomputer time, running many highly optimised (hence complicated) programs for different cases. It is clear why belief in such an argument is hard to come by, even with independent checking, which Lam suggests. This type of calculation is not a proof, not because it isn't "absolutely certain", but because there is no way for a reader to apply her own intuition to attain belief.

As an aside, my approach does raise a possibility that enumerative searches such as Lam's proof and the famous Appel and Haken proof of the four color theorem [AH77, Tym79], which can never be accepted as conventional proofs, might be made into formal proofs: e.g. we prove that some program (lambda term) correctly tests numbers for certain properties, we prove that if a certain finite set of numbers have those properties then every map is four-colorable, and we formally execute the program on that finite set, i.e. show that two lambda terms are convertible by computation.

There is another kind of uncertainty often mentioned regarding formal verification that some hardware or software meets its specification. Even when we believe the proof, there is uncertainty about the behavior of the physical object, since the specification is with respect to some model of the physical world, and we can never completely model the world. This issue is closer to that of truth, than to certainty in reliability of the proof.

**Probablistic proofs** A red herring sometimes arises (e.g. [DLP79]): since all proof is uncertain, why not abandon deterministic notions of proof in favor of probabilistic proof. Probabilistic proof systems [Gol94] can be much more powerful than their deterministic counterparts. While they carry a probability of error, this probability is explicitly bounded, and can be reduced to any desired positive number. Such approaches involve random choices (coin tosses), but it is still necessary to apply the rules of the system correctly. For a well-known example, using Rabin's algorithm for primality testing [Rab76] requires much less computation to test primality of very large numbers than conventional methods, but the probability bounds don't hold if you make mistakes in multiplication! In this paper I propose how to believe that you have correctly followed some set of rules. Further, even though probabilistic approaches require one to check very much smaller derivations than conventional approaches, this hardly supports a claim that the appropriate warrant for proof correctness is direct understanding of the proof, since probabilistic approaches abstract from the conventional meaning of a proof just as the indirect checking I propose.

## 2.3 Explanation

Explanation is purely informal, being the *pointing out* of what the author of the proof would like the reader to see. This pointing out is a kind of abstraction, and is at least as useful for a formal proof in a large file as for an informal proof. In formal mathematics, explanation

has no bearing on the correctness of a putative proof, but may be very important in the process of constructing a proof, and in the reader's work of bridging the formal-informal gap to see that the formal theorem expresses what it informally claims to express.

# 3 Some Details of the Approach

## 3.1 Is it a theorem?

How is the putative formal proof of Fermat's last theorem constructed? Users interacting with some proof tool such as Alf, Coq, HOL, Isabelle, LEGO, NQTHM, Otter, ..., develop a file that, when read by that proof tool, stimulates it to print "QED" or some such thing. This *proof script* is not a formal derivation, but rather contains instructions to the proof tool to find a derivation in the given formal system. That is, the script tells the proof tool to use heuristics, decision procedures, tactics, etc., that are particular to that proof tool. These are programs to compute derivations in the underlying formal system; e.g. derivable or admissible rules of the official logic, or searches for derivations which may fail (see [Pol95]). For example, many proof tools will support some kind of tautology checking and some kind of equality rewriting.

Crucially, there is no need for you to understand any of the tactics or heuristics in order to independently check the claimed proof of Fermat's last theorem. In principle such tactics and heuristics, when they succeed at their task, check that their results follow by basic derivations in the underlying logic: this is the definition of proof checking. The proof tool can write out the complete official derivation it constructs from the instructions in the proof script, and it is this official derivation that you can independently check in order to believe Fermat's last theorem. The questions to ask are whether it is feasible to write out the official derivation, and whether it is feasible to check it; section 4 is devoted to these questions.

## 3.2 The software/hardware platform

How can you have confidence in correctness of a proof just because it was checked by a computer program? Maybe the compiler that processed the proof checker has a bug, or the operating system, or the hardware. Hardware can have soft errors, caused by random events, as well as actual errors in the design or construction. There are techniques we can use to have confidence that the programming language and its platform are behaving properly. The most useful technique is independent checking: use a standard programming language for the simple checker, so it can be compiled using different compilers, running on different platforms. For example Standard ML has several compilers, running on many different operating system/processor platforms. Also we can check the proof using other simple checkers put forward as believable by expert referees (section 1.1). This is analogous to the social process by which conventional proofs are believed

**Verifying the platform** It is sometimes suggested that we should verify the correctness of the proof-checking program, and of the software and hardware that supports it [You95]. Such a verification is another formal proof that has to be believed. Since any interesting computational platform is too complicated to believe directly, and the only tool we have for believing it indirectly uses a trusted computational platform, the approach cannot be well-founded. But verification is very effective in producing reliable software and hardware, and should be used in the long term. We should always remember that even when software and hardware are proved to meet their specifications, we have no certainty that the physical object will behave as expected.

**Believing the platform, and the philosophical claim** Conventional mathematics is based on a large body of knowledge that is accepted. At best, mathematicians take many years to attain belief in all that they accept; nonetheless this at least is possible. It seems impossible to fully reduce to individual understanding the belief that a computational platform behaves as specified, due to the non-well-foundedness of any verification of such a claim.

Still, there is another way to look at it. In order to believe conventional mathematics by direct understanding we must believe that our own computational platform, our nervous system, behaves correctly. For example, that we identify symbols consistently, and that our short term memory of just having understood a certain statement is correct. Every subjective experience depends for its interpretation on some abstract correctness assumption about experience itself. In believing a formal proof indirectly by believing a proof checker, we are also shifting this abstraction to some computational platform outside of our consciousness. This is not simply giving up; just as we are careful about checking that our experiences are internally consistent and match with that of other people, so we are careful about the computational platform we use, and compare it with other independent platforms. Such a shift of abstraction seems unavoidable if we are ever to accept as a correct a putative proof that cannot actually be checked by a person.

## 3.3 What theorem is it?

You now have a file that you can read, and that you believe is a derivation in the given formal system. Since you can read it (section 4.4), you know what string is claimed to be Fermat's last theorem, and you can parse that string as a formula. That formula may contain defined names, and you can find and parse (as formulae) the definitions for these names. Is the proved formula Fermat's last theorem? You must read the formula and interpret it in the light of your understanding of the formal system. This is an informal operation; to do it, you must read all the definitions that are used hereditarily in the formula, but may skip definitions used in the proof but not in the formula. This is exactly how informal mathematics is done; and, no matter how big the formal proof, this process must be tractable, as the formula formally stating Fermat's last theorem is not significantly different than the informal statement. Informally we don't redefine the natural numbers, all their basic operations, etc., for every theorem, and we don't read these definitions when

we check a proof of Fermat's last theorem, while formally we do so. This is not a serious difference; conventional mathematics rests on a basis of mathematical knowledge that is previously believed, and formal mathematics must proceed in the same way, developing a library of formal knowledge covering this mathematical basis.

### 3.4 What have we gained, and where has it come from?

I have suggested how belief in correctness of a formal proof comes from engaging our own understanding to check it (and recheck it if necessary), and from the social process of many knowledgeable readers independently checking it. The only way this differs from informal mathematics is the extra indirectness in our checking, where we allow a machine to do the mechanical steps of pattern matching, substitution, etc. This extra indirectness is not a trivial matter: due to it we allow more things as proofs, e.g. derivations that are too big, or too combinatorially complicated, to be checked by a person, as mentioned in section 2.2.

An advantage hinted at above is that proofs in the same logic can be shared by different proof checkers for that logic, if a standard syntax can be found (or mechanical translations believed), because the official proofs don't depend on all the tactics and heuristics that are particular to individual proof tools. In current practice, of course, this idea is a can of worms, and even saying the phrase "in the same logic" causes experts in the field to roll on the floor with laughter. However, alternative suggestions such as using cryptographic means to certify that a theorem has been checked by some proof tool [Gru96] break the primary abstraction: the only way a proof checker can accept a theorem as proved is to actually check a proof of it.

It is necessary to restrict the notion of "proof checking program" to programs that actually check derivations in some given formal system, and to restrict the acceptable formal systems by criteria of feasibility of communicating and checking official derivations. This is discussed in section 4.

The eschewing of absoluteness is crucial to my argument. This is obvious, since absolute correctness cannot be attained by any means at all. However, some criticism of formalization seems based on the subtext that formal proof is not good enough since it cannot guarantee correctness. Formal proof can attain higher confidence than conventional proof, and can do so for more arguments.

## 4 About feasible formalisation

While correctness of derivations is defined *ideally* in conventional logic (e.g. the size of a derivation has no bearing on its correctness[3]), we are only interested in *actually* checked derivations.

[3]But both Hilbert and Wittgenstein require a proof be *surveyable*, or "given as such to our perceptual intuition", i.e. a feasible object in some sense

## 4.1 Formal systems for feasible checking

Work on *logical frameworks*, which are used as formal meta-theories, has enabled precise and concrete presentations of large classes of formal systems [AHMP92]. I have in mind such frameworks[4] as the Edinburgh Logical Framework (ELF) [HHP93, Gar92], Martin-Löf's framework [NPS90], Feferman's $FS_0$ [Fef88, MSB93], $\lambda$-prolog [NM88] and Isabelle [Pau94]. By "concrete presentation" I mean faithful to particular representation; all of these frameworks have been implemented as computer programs, and can be used as primitive proof checkers for any formal system they can express. However, for the purpose of actual formal mathematics, we are sensitive to properties of a formal system such as how large or complicated it is, for two distinct reasons. First, we are interested in believing mathematical statements from formal proofs, so we must be able to read and understand the formal system (logic) we are checking; this is an essential part of bridging the gap between a formal property and our informal belief. From this perspective, various presentations of first order logic (FOL) are suitable formal systems: there are few rules, they are organized around useful principles (introduction, elimination), and it is much studied and widely accepted. On the other hand, the Nuprl logic [Con86] is less satisfactory in this regard, as it has many rules, and some complicated side conditions (e.g. the `Arith` rule). The second reason that intensional properties of a formal system matter is that we want to actually check derivations of non-trivial statements in our formal systems, so the size of derivations, and more generally the feasibility of checking derivations is important. I will suggest technical means to ameliorate both of these limitations on our choice of formal system.

**How to believe a formal system.** One theme in this paper is using computers as a tool to bridge the formal-informal gap between a large formal object and our informal understanding of it. Since formal systems can themselves be studied mathematically in simpler meta-systems such as logical frameworks, we can apply the same technique to gain understanding and belief in a formal system that is otherwise too large or complicated. For example, the Nuprl logic can be formalized as an inductively defined class or relation in $FS_0$ or in Martin-Löf's framework, both of which are considerably simpler than Nuprl itself. Then various properties of formalized Nuprl can be proved in the framework, which might allow a reader who understands one of these frameworks to understand Nuprl as well.

**Formal systems that are feasible to check.** The second limitation is the requirement for feasible checking of derivations in a formal system. I discuss the relationship between styles of proof and feasible checking in section 4.2. Here I am interested in finding alternative presentations of a formal system, deriving the same judgements but with better intensional properties: smaller derivations, or ones that are easier to check. More generally, we can look for a different formal system (different language, deriving different judgements)

[4] The first logical framework was Automath [NGV94].

that allows us to more feasibly check the original formal system in some indirect way. Two things can make a formal system computationally expensive to check: the derivations can be big, or the side conditions can be expensive to check. There is a tradeoff between these two issues, as expensive side conditions can be replaced by big subderivations. However it is not always obvious how to factor big derivations into expensive side conditions.

As an example of big derivations, the Gentzen cut-free system for FOL is completely infeasible. It is well known that adding the cut rule doesn't change the derivable judgements and allows much smaller derivations, so the system with cut is a better choice for formal mathematics. Section 4.3 discusses how such meta-theoretic extensibility as adding admissible rules to a logic (e.g. the cut rule) can be supported by a simple, believable proof checker.

Another example of an alternative presentation with smaller derivations is "writing a derivation tree as a DAG". In a derivation tree, subderivations may have repeated occurrences, because they need to appear at different places in the tree. By using a linear presentation of derivations instead, where each line names the previous lines it depends on, only one occurrence of each subderivation is required; the indirectness of naming previous lines allows sharing[5]. Extending a formal system with definitions allows a similar kind of sharing. The Automath languages used the technique of naming expressions, lines, and contexts to avoid duplication of work. In these cases just mentioned, we depend on the constructor of the derivation to find the common substructures, but some formal systems duplicate work in such a uniform way that we can give an alternative system that shares some common substructures by construction. Martin-Löf [Mar71] gives an algorithm for type synthesis in his impredicative system (now called $\lambda^*$) that transforms official derivations to avoid duplicate work. This idea is used in Huet's Constructive Engine [Hue89]; an abstract explanation and machine-checked proof of correctness of this transformation on type systems is given in [Pol94] section 4.4.10.

Another common technique for improving efficiency of checking a formal system is annotation of judgements so that a full derivation of a judgement can be mechanically constructed from the judgement itself. The use of decidable type checking as a tool for proof checking uses this approach, where the terms are annotations that can be expanded into full derivations, so full derivations don't have to be constructed or communicated. Equivalently, we can think of omitting parts of official derivations that can be mechanically reconstructed. In the calculus of constructions (CC), terms are essentially derivations with instances of the conversion rule and variable lookup elided. There is a clear tradeoff to keep in mind: the more information we elide from derivations (making them smaller, so easier to communicate) the more has to be mechanically reconstructed (so making them more difficult to check). In CC, it is only possible to elide instances of the conversion rule because its side condition, convertibility of two well-typed terms, is decidable; but checking convertibility is certainly not feasible in general, so neither is proof checking without help from annotations.

In an interesting formal system there is always some derivation that is too hard to check

[5]This interesting way of viewing linear derivations was pointed out to me by Harold Simmons [Simar].

with any possible hardware and software. Nevertheless, in the context of independent checking of proofs, annotations solve an important problem: suppose I *have* constructed some proof in CC, using clever heuristics for conversion testing; how can you believe this proof? By my paradigm, you can believe a proof checker for CC, and then check the proof with this proof checker. This approach is hopeless if your simple, trusted proof checker doesn't use adequate heuristics for conversion testing; it may be equally hopeless if you must believe all my clever heuristics, and also believe their correct implementation in your proof checker. On the other hand, my heuristics must have found a feasible conversion path to verify the proof (assuming I did actually check it), so if I annotate the proof with this conversion path, your checker need only follow the annotations, not use heuristics to discover a conversion path for itself. In LEGO, supporting type theories with definitions, there is an observed problem of this kind: when interactively constructing a proof, a user can tell the proof tool which definitions to expand, but the term constructed in the official language to witness the proof is not annotated with this information, and may be very expensive to re-check.

## 4.2 Feasible formal proofs

In the preceeding section I discussed formal systems that are suitable for *actual* checking. This section discusses issues of feasible checking of formal proofs, and more generally, of whole formal developments of mathematics. Even in a well-behaved formal system, there will be proofs that are infeasible because of proof style. Analogously, in programming, it is well known that there are feasible functions with infeasible implementations. For example, the natural recursive defintion of the fibonacci function is exponential in its input, while an alternative definition is linear. Similarly, to support actually checking proofs, we will be restricted to proofs that can actually be checked. For example, if `ack` is the Ackermann function, trying to prove $\texttt{ack}(100) - \texttt{ack}(100) = 0$ by computation is hopeless, while proving $\forall n.n - n = 0$ is trivial, and leads to a trivial proof of $\texttt{ack}(100) - \texttt{ack}(100) = 0$.

**Representation** Just as in programming unsuitable representation of data is one of the causes of unfeasible programs, so in proofs unsuitable representation of the objects of discourse is a cause of uncheckable proofs. This may be hard to recognise in formal mathematics because we are used to representations from conventional mathematics, which were never intended to be used in actual formalization. Here is an example that made a measurable difference in some LEGO proofs:

**Example 4.1 (Inductive vs. recursive definition of tuples)** *A natural definition of the type of $n$-tuples over a type $A$ is as a repeated cartesian product, by primitive recursion over natural numbers:*

```
Tuple Zero    = unit
Tuple (Suc n) = A # (Tuple n)
```

*This is the definition one would expect in an informal presentation. An alternative is defined by induction:*

```
Inductive    [tuple: nat->Set]
Constructors [nil  : tuple Zero]
             [cons : {n:nat}A->(tuple n)->(tuple (Suc n))]
```

*I have never seen this definition in an informal presentation, but it has the advantage for machine-checking that* `(tuple n)` *is canonical for all* $n$*, while* `(Tuple n)` *is not.*

When we formalize a notion we make choices about representation, but there is no reason to believe there is a single "best" representation, or even any single good representation in the sense that it leads to natural statements and short proofs of theorems. However, we can make several definitions for a concept, prove something about their relationship, and move between them as convenient. Some alternative definitions may be used only for convenience or feasibility, while some may be true alternatives for the official definition of some concept. Also note that our choice of representations is constrained by our underlying formal system (e.g. $FS_0$ cannot express generalized induction, while Martin-Löf's framework can), and this may be a reason for choosing one framework over another.

An example is the use of unary representation and base representation for natural numbers. We probably want to use unary representation as the official definition of the naturals, and base representation for any actual computation, e.g. for the computational content extracted from constructive proofs. In order to do this, elementary school arithmetic must be formalized, i.e. the correctness of various algorithms for arithmetic operations on base representation numbers.

An often discussed example of a notion that is hard to reason about formally is binding, and there are many representations in the literature. Variable names [CF58] are the naive approach. De Bruijn indexes [dB72], inspired by needs of machine implementation, are also convenient for formal reasoning in many applications. Even simpler is *higher order abstract syntax* (HOAS) [PE88] as used in ELF and Isabelle. But HOAS doesn't naturally support structural induction, and sometimes it is desired to formalize expressions with names, since this is what people use. A new and interesting approach meets these needs [Gor93, GM96]. However useful this approach turns out to be, it represents expressions only up to alpha conversion (as with de Bruijn indexes, HOAS, and another alternative [Sat83]). A more intensional approach, using parameters and variables, is suggested in [Coq91] and formalized in [MP93, Pol94]. As just suggested, these representations are not all "isomorphic": a presentation of type theory using de Bruijn indexes has different theorems than one using parameters and variables[6], but by formalizing the relationship between them, theorems can be stated and proved in their most natural forms, and used in different forms when needed. It may not be obvious what the official formalization of some informal concept should be (e.g. are de Bruijn terms the real meaning of lambda terms, or just a convenient representation), but formal mathematics doesn't have to split hopelessly over such questions. As long as your favorite definition can be shown to be appropriately related to other definitions in the formal literature, you can use existing results.

[6] For example, compare the thinning lemma in [MP93], which is close to the informal statement, with that in [Bar95], where explicit variable lifting is required.

Even what I have said about new and different representations for mathematical notions is too restricted. We can look for entirely new ways to do mathematics that are especially suited for formalization in particular formal systems. An example of this is the use of formal topological models that has recently received interest in the Type Theory community. Persson [Per96] describes the formalization of a completeness theorem for intuitionistic first-order logic, using formal topological models as suggested by Sambin [Sam95]. Coquand [Coq] proposes a program of proof-theoretic analysis of non effective arguments using formal topological models. Such problems seemed infeasible for Constructive Type Theory until this approach was developed.

## 4.3 How to believe a mechanical proof checker

Every computer program is a formal object, and in this sense, every output of every computer program is a mechanically-checked consequence of some formal system. The problem is, we don't know what formal system it is, and even if we could decompile the program to discover the formal system it checks, that system would be too big, complicated and ad-hoc for us to understand very much about it, or care. A proof checker is a program that checks a particular, specified formal system. For actually constructing formal proofs we may use a proof tool with many proof search heuristics, tactics, etc, but for believing a theorem from a formal proof we want a proof checker that we can believe is correct just by reading and understanding some of its code. The prototypical way to do this is using the "LCF style" of proof checker construction, such as used in HOL [GM93]. HOL afficionados might argue that HOL is suitable for both constructing proofs, because of it's tactic language and rich library of available tactics, and for independently checking them to attain belief, since each tactic actually expands to atomic steps of the formal system, and only a small kernel of code must be read and understood to believe in the system's correctness. Only one thing is missing: the current implementation of HOL doesn't actually store or write out the official proof that is constructed by expanding all the tactics, although this is also under study by the HOL group [Won93]. Without this feature, a reader cannot use other, independent checkers for the HOL logic to increase confidence in a proof, and cannot easily use proofs developed by other proof tools for HOL.

**A concrete suggestion: the three-level approach.** LCF style uses two-levels, with a computational meta-language (e.g. SML [MTH90]) in which a proof checker for an object-level logic is programmed. Strong typing of the meta language is seen to guarantee that access to the atomic proof constructors of the object logic is safely controlled. Logical frameworks are suitable meta-languages for implementing believable proof checkers; these frameworks are more precisely and concretely specified than even the best programming languages and are designed specifically for representing formal systems (although with current implementations there may be problems of efficiency). Further, in the frameworks mentioned above, derivable rules of represented object systems can be proved correct; these meta-proofs are analogous to LCF tactics that expand into an official proof. However a reader who believes in the framework can believe in a proof using derivable rules without

actually expanding them, as s/he believes they could be expanded in principle. In the stronger frameworks, $FS_0$ and Martin-Löf's framework, many admissible rules can also be proved correct, and again can be used as if they were official rules, since they could, in principle, be expanded into official proofs. I just used the phrase *in principle* twice, but am not going back on my committment for actual checking: we actually check derivations in an extended system that we believe, by actually checking admissibility of some rules, represents some other, less feasible system. Such meta-theoretic extensibility of believable proof checkers can greatly reduce the burden of actually checking fully expanded proofs [Pol95].

The question then arises: where will we find a believable implementation of a logical framework? We can use an LCF style implementation for this. Isabelle is an already existing example of this approach: you can believe Isabelle by reading only the safe kernel implementing the rules of its meta-logic, and you can get a proof checker for your chosen object logic just by specifying it in a "high-level" form. Isabelle does support proof of derivable rules of object logics, that are safely used without expanding to official steps of the object logic, but is not a perfect realization of my suggestion for two reasons. First, Isabelle uses proof theoretically strong higher-order logic as its meta-logic, and uses higher-order unification inside its safe kernel; we might want a weaker meta-logic and a simpler kernel (perhaps using Miller's $\beta_0$ unification [Mil91]). Second, Isabelle supports derivable rules, but not admissible rules of object logics [Pol95]. The three-level approach places few restrictions on the object logic. A different approach is *reflection* [ACHA90, ACUar, Har95], that collapses the framework and the object logic in order to provide admissible rules that can safely be used without expansion. However reflection makes demands of the object logic, and seems much harder to believe in every way.

## 4.4 Reading the proof

Both parts of the approach, independent checking and understanding the statement of the theorem, require reading the proof files. There are two classes of question involved with this: correct specification and operation of hardware and software to read the files as a long ascii string, and of software to parse this ascii string as a proof, and to pretty-print parts of it so you can read them. The former is the job of the software/hardware platform; see section 3.2. For the latter, parsing is one of the formally best understood areas of computer science, so this should be no problem. Nonetheless, it does constrain the language of our official formal system: this must be parseable. User friendly proof tools often support complex user-extensible synatx, and even unparsable syntax entered using control keys and special editors, e.g. Nuprl[7]. Nuprl is an LCF style implementation, so suppose we believe the safe kernel correctly checks the rules of the formal system. On the screen we see that Nuprl accepts a proof whose conclusion appears to be Fermat's last theorem, but that string of symbols is *not* formally related to the official judgement that the kernel accepted, as Nuprl allows any string of characters to stand for any official expression. Part of the problem is that definitions are meta-notation in Nuprl, not part of the official logic.

[7] My knowledge of Nuprl is very dated [Con86].

Even proof systems that have official object-level definitions often support (more or less principled) meta-notation (e.g. LEGO, Isabelle), and this should be viewed as part of what has to be understood and believed about a simple proof checker for their logics. This is usually overlooked in LCF style systems.

### 4.5 The business of formal mathematics

Some people believe that formal mathematics is "just filling in the details" of conventional mathematics, but "just" might be infeasible unless some thought is given to representation and notation. This can be considered to be a hassle, or to be the business of formal mathematics. This latter view leads us to be interested in areas of feasibility and expressiveness of formal systems, such as strict finitism (e.g. [Car95]) and the power of formal systems to efficiently represent algorithms (e.g. [Col91, Fre95]), and to study formal representations of mathematical notions.

## References


[ACHA90] Allen, Constable, Howe, and Aitken. The semantics of reflected proof. In *LICS Proceedings*. IEEE, 1990.

[ACUar] William Aitkin, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. *Journal of Automated Reasoning*, To appear.

[AH77] K. Appel and W. Haken. Every planar map is four-colorable. *Illinois Journal of Mathematics*, XXI(84):429–567, September 1977.

[AHMP92] Arnon Avron, Furio Honsell, Ian Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–352, 1992.

[Bar95] Bruno Barras. Coq en Coq. Master's thesis, INRIA-Rocquencourt (Project Coq), October 1995.

[Bis67] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967. No longer in print.

[Boy94] Robert S. Boyer. A mechanically proof-checked encyclopedia of mathematics: Should we build one? can we? In Alan Bundy, editor, *CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June/July 1994*, number 814 in LNAI. Springer-Verlag, 1994.

[Car95] Felice Cardone. Strict finitism and feasibility. In Daniel Leivant, editor, *Logic and Computational Complexity. Proceedings, 1994*, number 960 in LNCS. Springer-Verlag, 1995.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[Col91] Loïc Colson. *Représentation Intentionnelle d'Algorithmes dans les Systèms Fonctionnels: Une étude de Cas*. PhD thesis, University of Paris 7, January 1991.

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[Coq] Thierry Coquand. Formal topology and constructive type theory. Talk at *Twenty Five Years of Constructive Type Theory, Venice, Oct 1995*.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34(5), 1972.

[DLP79] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22:271–280, 1979.

[Fef88] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquium '88, Padova*. North-Holland, August 1988.

[Fre95] Daniel Fredholm. Intensional aspects of function definitions. *Theoretical Computer Science*, 152:1–66, 1995.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, July 1992.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.

[GM96] Andrew Gordon and Tom Melham. Five axioms of alpha conversion. In *International Conference on Theorem Proving in Higher Order Logics, Turku, Finland*, 1996. To appear.

[Gol94] Oded Goldreich. Probabilistic proof systems (a survey). Technical Report RS-94-28, BRICS, Aarhus, 1994.

[Gor93] Andrew Gordon. A mechanism of name-carrying syntax up to alpha-conversion. In *Higher Order Logic Theorem Proving and its Applications. Proceedings, 1993*, LNCS 780. Springer-Verlag, 1993.

[Gru96] Jim Grundy. Trustworthy storage and exchange of theorems. Technical report, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, 1996.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, UK, 1995. `http://www.cl.cam.ac.uk/ftp/hvg/papers/`

[Har96] John Harrison. Formalized mathematics. Technical report, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, 1996. `http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html`.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. Preliminary version in LICS'87.

[Hor93] John Horgan. The death of proof. *Scientific American*, pages 74–82, October 1993.

[Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[Lam90] C.W.H. Lam. How reliable is a computer-based proof? *The Mathematical Intelligencer*, 12(1):8–12, 1990.

[Mar71] Per Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, 1971.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93, Utrecht*, number 664 in LNCS, pages 289–305. Springer-Verlag, March 1993. `ftp://ftp.dcs.ed.ac.uk/pub/lego/formalPTS.ps.Z`.

[MSB93] Seán Matthews, Alan Smaill, and David Basin. Experience with $FS_0$ as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*. The University Press, Cambridge, 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NGV94] R.P. Nederpelt, J.H. Geuvers, and R.C. De Vrijer, editors. *Selected Papers on Automath*. North-Holland, 1994.

[NM88] Gopalan Nadathur and Dale Miller. An overview of lambda prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference, Seattle, Washington, August 1988*, pages 810–827. MIT Press, 1988.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208, June 1988.

[Per96] Henrik Persson. Semantics of first order languages in type theory. In preparation for Licenciate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, 1996.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994. `ftp://ftp.dcs.ed.ac.uk/pub/lego/thesis-pollack.ps.Z`.

[Pol95] Robert Pollack. On extensibility of proof checkers. In Dybjer, Nordstrom, and Smith, editors, *Types for Proofs and Programs: International Workshop TYPES'94, Båstad, June 1994, Selected Papers*, LNCS 996, pages 140–161. Springer-Verlag, 1995.

[Rab76] M.O. Rabin. Probabilistic algorithms. In J.F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–40. Academic Press, 1976.

[Sam95] Giovanni Sambin. Pretopologies and completeness proofs. *Journal of Symbolic Logic*, 60:861–878, 1995.

[Sat83] Masahiko Sato. Theory of symbolic expressions, I. *Theoretical Computer Science*, 22:19–55, 1983.

[Simar] Harold Simmons. *Algorithmic Proof Theory: Taking the Curry-Howard correspondence seriously.* Cambridge University Press, To appear.

[Sla94] John Slaney. The crisis in finite mathematics: Automated reasoning as cause and cure. In Alan Bundy, editor, *CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June/July 1994*, number 814 in LNAI. Springer-Verlag, 1994.

[Tym79] Thomas Tymoczko. The four-color problem and its philosophical significance. *The Journal of Philosophy*, LXXVI(2):57–83, February 1979.

[Won93] Wai Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, 1993.

[You95] William Young. System verification and the CLI stack. Technical Report 108, Computational Logic, Inc., May 1995.