Basic Research in Computer Science

How to Believe a Machine-Checked Proof

Robert Pollack

BRICS Report Series **RS-97-18**

ISSN 0909-0878 **July 1997**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/97/18/`

How to Believe a Machine-Checked Proof[1]

Robert Pollack
BRICS, [2] *Computer Science Dept., Aarhus University*
DK-8000 Aarhus C, Denmark

1 Introduction

Suppose I say "Here is a machine-checked proof of Fermat's last theorem (FLT)". How can you use my putative machine-checked proof as evidence for belief in FLT? 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 other mental processes which are impossible to formalise.

By machine-checked proof I mean a formal derivation in some given formal system; I am talking about derivability, not about truth. 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 understood it. This paper presents a technological approach for reducing the problem of believing a formal proof to the same psychological and philosophical issues as believing a conventional proof in a mathematics journal. The approach is not entirely successful philosophically as there seems to be a fundamental difference between machine checked mathematics, which depends on empirical knowledge about the physical world, and informal mathematics, which needs no such knowledge (see section 3.2.2).

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, pointing out 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.

1.1 Outline of the approach

The problem is how to believe FLT when given only a putative proof formalised in a given logic. Assume it is a logic that you believe is consistent, and appropriate for FLT. The "thing" I give you is some computer files. There may be questions about the physical and abstract representations of the files (how to physically

[1] A version of this paper appears in Sambin and Smith (editors) *Twenty Five Years of Constructive Type Theory*, Oxford University Press.

[2] Basic Research in Computer Science, Centre of the Danish National Research Foundation. The author also thanks Edinburgh University and Chalmers University.

read them and how to parse them as a proof), and correctness of the hardware and software to do these things; ignore them until section 3.1.

My approach is to separate the problem 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).

Is it a theorem? Is the putative 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, so this approach greatly simplifies the problem[3] (see section 3.2). 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 light 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*[4] (Gordon, Milner and Wadsworth 1979) 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 semantics, then the sum total of what you

[3] Conversely, 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.

[4] LCF style is an architecture for proof checker implementation using a strongly typed programming language such as Standard ML, such that object logic theorems form a type, and the construction of theorems is controlled for soundness by the programming language type system. Pollack (1995) gives a modern view and some variations.

are required to read and understand is neither longer nor more difficult to understand than a conventional proof, and belief in the putative derivation is attained through your personal experience of understanding a simple proof checker. 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 proof. You can recheck it with another proof checker, perhaps one publicly available from a library of checkers that are refereed by experts, and that have high confidence from being used to check previous examples. If a few logics become accepted as appropriate for formalisation, 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.

What theorem is it? Having believed that my putative proof is actually a derivation in the claimed formal system, you ask "does it prove FLT?" 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[5]. 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 formal proofs. This issue is sometimes overlooked in discussions of reliability 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), 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 and lemmas, 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 trusted proof checker. You can use

[5]In the (distant?) future all the 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, 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 FLT.

the trusted proof checker to print out the parts of the formal proof you need to read. (It is necessary to trust the tool that shows us the assumptions used in the proof, as the formal proof is too big to ascertain for ourselves that these really are all the assumptions used.) Then it is up you, using your own understanding of the formal system, to decide if the formal statement means what is informally claimed. But this is anyway a subtask of believing a conventional proof; so this second subproblem of believing a formal proof is no more difficult than the corresponding aspect of believing a conventional proof. Having used your own understanding, you can gain confidence by discussing the problem with other knowledgeable readers, as in the first step of the approach.

1.2 Related work

The prototypical paper on this topic is DeMillo, Lipton and Perlis (1979), 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 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 for increasing confidence in computer-based enumerative search (Lam 1990) (see section 2.2 below). It is considered the standard approach in tasks such as computing many digits of π. It is mentioned by Cohn (1989). A proposal similar to the present paper ("verify the proofs rather than the programs which produce them") is made, in less detail, by Slaney (1994).

There has recently been discussion of the possibility and desirability of pursuing formal mathematics (Boyer 1994, Harrison 1996). My guess is that technologies of automated proof search will be highly developed because of economic pressure for reliable hardware and software, and these, applied as tactics to proof checking, will make formal mathematics a practical reality in the foreseeable future. This paper addresses some points necessary for this 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.
>
> Bishop (1967)

We have no access to *truth* in any aspect of human experience, including either formal or informal mathematics; predictions about the world might always be falsified by experiment. Even when we formally verify that some hardware or software meets its specification, 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.

For me this is neither a deep claim nor a serious limitation on our practice of mathematics. The problem stems from too broad a notion of truth; I will restrict my comments to *proof*, suggesting how to approach the question of whether Peano Arithmetic (or ZF set theory, or the calculus of constructions (CC), ...) proves FLT for some given definitions of "natural number", "addition", etc.

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.
>
> L.A. Levin, quoted in (Horgan 1993)

Everyone has had the experience of understanding and believing a proof at one time, and later seeing an error in it. After such an experience, you must acknowledge that it might happen again. Therefore the notion of certainty, like that of truth, is not relevant 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 Lam (1990), talking about reliability of enumerative searches 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**." (Lam's *italics*, but my **bold**, to show the contrast with my belief that no knowledge is absolute.) No useful analysis seems possible of the probability of error caused by software bugs in big calculations, and I don't think this is what readers of proofs want. What is 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 that belief in such an argument is hard to come by, even with independent checking (which Lam suggests), 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 (1977) proof of the four color theorem, which can never be accepted as conventional proofs, might be

made into formal proofs that are believable by the indirect means of independent checking. For example in type theory we might construct an inhabitant of the four color theorem (not just a believable argument, but a formal object) by proving that some program (lambda term) correctly tests numbers for certain properties, proving that if a certain finite set of numbers have those properties then every map is four-colorable, and executing the program on that finite set (showing that two lambda terms are convertible by computation).

Probabilistic proofs A red herring sometimes arises (DeMillo et al. 1979): since all proof is uncertain, why not abandon deterministic notions of proof in favor of probabilistic proof. Proof systems involving random choices (coin tosses) can have much smaller derivations than their deterministic counterparts. While they carry a probability of error, this probability is bounded, and can be reduced to any desired positive number. But the probabilistic nature of such systems doesn't mean you are allowed to make mistakes in applying their rules. Further, since "random choices" must be independent of each other, a trace of all the steps of a probabilistic proof is not convincing at all: if I toss the coin myself I may believe a probabilistic proof, but if I only see a written trace of the proof the coin toss outcomes may have been faked to give the desired result. Thus probabilistic approaches do not support a claim that the appropriate warrant for proof correctness is direct understanding of the proof; it is the procedure for probabilistic testing (that asks for occasional random input) that must be believed, not the proof itself, and in this indirectness probabilistic proof is similar to my suggestion of indirect checking.

2.3 Explanation

Explanation is the purely informal *pointing out* of what the author of the proof wants the reader to see. This pointing out is a kind of abstraction, and is at least as useful for a formal proof 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.

3 Some details of the approach

This section treats some points that were postponed or suppressed in section 1.1, and concludes with an overview of what has been gained.

3.1 Reading the files

Both parts of the approach, independent checking and understanding the statement of the theorem, require examining the proof files. Questions arise about
• correctness of hardware and software to read the files as a long ascii string, and
• correctness 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 (section 3.2.2). Here I consider parsing and pretty-printing.

The second part of my approach requires us to read the formula that is derived, and verify that it is really FLT. We read a concrete representation of the formula (an ascii string), but the proof checker uses an abstract representation (an abstract syntax tree). If we don't understand the relationship between these representations, then nothing the proof checker says can be believed, no matter how trustworthy the checker is at the level of abstract representation. (This is often overlooked in discussing LCF style proof checkers, where correctness of the kernel, implementing abstract proof constructors, is taken to be the only critical part of the program.) Consequently the language of our formal system must be parseable in a simple and formally explained way. Some proof tools support complex user-extensible syntax, and even unparseable syntax entered using control keys and special editors. This may be helpful to users while constructing proofs and browsing libraries, but to believe such a checker, it must also support an official syntax that is parseable and printable.

3.2 Is it a theorem?

How is the putative formal proof of FLT constructed? Users interacting with some proof tool (Alf, Coq, HOL, Isabelle, LEGO, NQTHM, ...), develop a file that stimulates the tool to print "QED". This *proof script* is not a formal derivation, but contains instructions to the proof tool to find a derivation, i.e. the script refers to heuristics, decision procedures, tactics, etc., that are particular to that proof tool. These are programs to compute derivations in the official formal system; e.g. derivable rules, or arbitrary searches that may fail (Pollack 1995). For example, many proof tools support tautology checking and equality rewriting tactics.

Crucially, there is no need for you to understand any of the tactics or heuristics in order to independently check the claimed proof of FLT. In principle such tactics, when they succeed at their task, check that their results follow by official derivations: this is the definition of proof checking. The proof tool can write out the complete official derivation it constructs from the proof script. Since checking a derivation is a simple thing, you should be able to independently check this official derivation of FLT using a simple proof checker that you trust. The questions to ask are • is it feasible to write out the official derivation and to check it, and • how can you trust any proof checker. The former question is addressed in section 4; here we consider the latter.

The hardware/software stack How can you have confidence in a proof just because it was machine checked? There are many layers of hardware and software involved in checking the proof of FLT. The top layer is a simple proof checking program for some specified formal system, coded in some programming language. The bottom layer is a physical machine. Intermediate layers include compiler, linker, operating system, etc. Bevier, Hunt, Moore and Young (1989) show that such a system can be formalised as a stack of abstract machines. Interfaces in this stack are specified, for example, by programming language semantics, operating system definition and hardware definition. Each layer implements its

specification in terms of the next layer down, and if each layer is verified then the whole stack is a verified implementation of the top layer specification in terms of the physical model of the machine at the bottom layer. The work of Bevier et al. (1989) is the limit of current technology, and in current practice, very few of these layers are formally specified, let alone verified.

Every (unverified) computer system has bugs, but we have confidence in the behavior of a general purpose computing environment because there are many users "testing" the environment over time, allowing a consensus to develop. Further, lower layers of the stack are largely interchangeable, allowing for independent checking. For example if I've coded my LCF style proof checker in Standard ML (SML), I can compile it using different SML implementations and run it on different operating system/processor platforms; then a proof can be re-checked without depending on any particular system platform. It is unlikely that a bug in the computing platform causes a proof checker to erroneously accept a proof, and incredible that independent platforms erroneously agree, but a bug in the proof checking program itself may cause an erroneous proof to be accepted when checked using independent platforms. Thus in practice we are more interested in validating the proof checking program than the rest of the computing environment; we can try to verify it and we can also use independent simple proof checkers for the same logic that have gained a consensus of trust over time. For these reasons, most people feel that machine checking a proof increases its reliability even if the system platform is not verified.

But my approach calls for more than consensus by random testing; it calls for consensus among readers who have each attained belief by personal intuition applied directly or indirectly. The top layer of the stack, the simple proof checker program itself, can be believed by direct understanding of a small amount of code; this is discussed in section 3.2.1. That the rest of the stack is not so easy to believe, even when verified, is discussed in section 3.2.2.

3.2.1 *How to believe a proof checking program*

The top layer of the hardware/software stack is a simple proof checking program. Its specification is a definition of the formal system to be checked,[9] and its job is to implement this definition in terms of the specification of the next layer in the stack, i.e. the programming language in which it is coded. We want to believe that the checking program is correct by understanding an amount of code that is small compared to the size of a formal proof, but in order to understand any code at all we must understand the semantics of the programming language it is written in, and we will use non-trivial properties of the semantics to attain belief in the checker. For example, if the checker is coded in LCF style, we are depending on strong type correctness properties of the SML *definition* (Milner, Tofte and Harper 1990), not just of particular SML implementations, when we

[9] The question whether this definition captures our informal understanding of the logic is not about correctness of the checker, but about bridging the formal-informal gap, and is treated in step two of my approach.

claim that no non-theorem can be in the type of theorems. Thus we must use a programming language with a simple formal semantics and study the properties that are needed to trust a proof checker in this language. SML is perhaps too complicated to do this; e.g. see (Kahrs 1993) and (VanInwegen 1996).

The three-level approach. I suggest that the "programming language" for the checking program be a logical framework, i.e. a formal meta-theory, enabling precise and concrete presentations of a class of formal systems. I have in mind such formalisms as the Edinburgh Logical Framework (ELF) (Harper, Honsell and Plotkin 1993), Martin-Löf's framework (Nordström, Petersson and Smith 1990), FS_0 (Feferman 1988), and Isabelle (Paulson 1994). These frameworks are precisely and concretely specified and are designed for representing formal systems. The user gives a definition of the object logic to be checked (details vary), and either the implementation of the framework itself becomes a checker, or we program a checker in the internal language of the framework which is formally proved to be correct w.r.t. the definition (Pollack 1994, Barras 1996). The latter of these variations can be seen as an application of LCF style to type systems more expressive than SML (Pollack 1995) and supports LCF style tactics. For technical reasons, classical LCF tactics must be expanded to official proofs (which can be very costly) even when they are meta-theoretically proved to be sound, while the suggestion of Pollack (1995) allows admissible rules as tactics that don't have to be computed to official proofs. This is very important for feasibility of checking. Considerable progress is being made in the theory and application of logical frameworks, e.g. (McDowell and Miller 1997).

The question then arises: where will we find a believable implementation of a logical framework? We can use classic LCF style, with an SML-like programming language (perhaps simpler than SML). Isabelle (Paulson 1994) is an example of this approach: an LCF style implementation of a higher order logic is used as a logical framework; you get a proof checker for your chosen object logic by specifying it in a high-level logical form. But Isabelle is not a perfect realization of my proposal for two reasons. • Higher-order unification is part of the safe kernel; I suggest a kernel supporting the programming of unification as a tactic in terms of simpler atomic actions. • Due to its style of framework, Isabelle supports derivable rules but not admissible rules of encoded object logics.

This three-level approach (programming language, logical framework, object logic) places few restrictions on the object logic; some frameworks restrict the induction principles available for defining object logics, but this is not a problem in practice. There is a different proposal in the literature, called *reflection* (Allen, Constable, Howe and Aitken 1990, Harrison 1995), that collapses the framework and the object logic in order to provide admissible rules that can safely be used without expansion. However reflection distorts the object logic (with an added *reflection* rule), and is much harder to believe.

3.2.2 *Believing the hardware/software stack and the philosophical claim*

It seems that a computer system (CPU, operating system, compiler, ...), is too complex to believe by direct understanding[10]. Thus the only possibility is to attempt indirect belief by verification. I think this must fail, but I want to make clear that I am not criticising verification as such; it clearly improves the reliability of computer systems. I am questioning whether verification can satisfy my goal to reduce believing a formal proof to the same psychological and philosophical issues as believing a conventional proof. Assume we have a verified hardware/software stack whose top layer is a simple proof checking program. The the formal-informal gap at the top of the stack has been mentioned in section 3.2.1, and does not challenge the philosophical goal. I see two other problems: • the formal-physical gap at the bottom of the stack, and • believing the verification of the stack itself.

At the bottom of the stack is a formal model of the behavior of physical hardware, predicted by quantum mechanics and tested by experiment. Belief in machine checked mathematics must depend on scientific theory, i.e. empirical knowledge about the physical world whose correctness cannot be believed by individual understanding. In order to believe conventional mathematics by direct understanding we must believe that our own computational platform, our nervous system, behaves correctly, e.g. that we identify symbols consistently, and that our short term memory is correct. But there is no question of belief in conventional mathematics depending on empirical knowledge for two reasons: • we don't (yet) have an empirical theory of cognition, and • the operations we would seek to explain by such a theory are subjective, and cannot be invalidated by failure of such a theory. Thus my philosophical goal is not met for the paradoxical reason that we demand more justification for indirect belief than for direct belief, since its physical foundation includes non-subjective aspects.

The verification of the stack itself is another formal proof that must be believed. Since this proof is surely too big to believe directly, we must believe it indirectly by independent checking with a trusted proof checker. But this cannot be well founded, as the problem we are now addressing is how to trust a proof checker.[12] This problem, too, will prevent meeting my philosophical goal.

The philosophical claim Subjective experience depends for its interpretation on abstract correctness assumptions about experience itself. In believing a formal proof indirectly by believing a proof checker, we are 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 indepen-

[10] A system architecture designed specifically for simplicity might overcome this problem.

[12] This might be overcome by using a very simple directly believable computing platform to check the verification of a more realistic platform.

dent 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 believe is a derivation in the given formal system. Is the derived formula really FLT? You must interpret the formula with your own understanding. This is exactly how informal mathematics is done; no matter how big the formal proof, this process must be tractable, as the formal statement of FLT is not significantly different than the informal statement.

One apparent difference is that informally we don't redefine the natural numbers and all their basic operations, and prove the properties of these operations for every theorem we want to believe. Conventional mathematics rests on a basis of mathematical knowledge that is previously believed, and formal mathematics must proceed in the same way, developing libraries of formal knowledge covering this mathematical basis. Such libraries should be developed, independently checked and widely used by a community of mathematicians. Then we can have confidence, not only that their theorems are provable, but also that their definitions mean what they informally claim.

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 the tactics that are particular to individual proof tools. In current practice this idea is a can of worms, and 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 (Grundy 1996) 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 (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 formalisation 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[14]), we are only interested in *actually* checked derivations.

4.1 Presentations of formal systems

For the purpose of formal mathematics, we are sensitive to properties of a formal system such as how large or complicated it is, for two distinct reasons: • we must understand a logic in order to believe a theorem it proves, • its derivations must be feasible to check. Thus it is the presentation of a formal system that interests us, not just the consequence relation it defines.

How to believe a formal system. We are interested in believing mathematical statements from formal proofs, so we must be able to read and understand the formal system 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 widely studied and accepted. On the other hand, the Nuprl logic (Constable et al. 1986) is less satisfactory in this regard, as it has many rules and some complicated side conditions.[15]

By meta mathematical study of a complex formal system, we may see how it is related to some simpler or better known system. For example in 1987 I struggled with ELF (with 3 classes of terms, 5 judgements and 17 rules) until I realized it translates into a subsystem of CC (with one class of terms, 2 judgements and 8 rules). One theme of this paper is using computers as a tool to bridge the gap between a large formal object and our informal understanding of it; since formal systems can themselves be studied mathematically in simpler meta-systems (e.g. logical frameworks), we can apply the same technique to gain understanding of a formal system that is otherwise too large or complicated by formally proving some of its properties. Of course there are limitations to this approach, and system presentations that are too complicated (e.g. Nuprl or the SML definition) may not be considered formal at all.

Formal systems that are feasible to check. The second reason we care about intensional properties of a formal system is that we want to actually check derivations of non-trivial statements, so the size of derivations, and more generally the feasibility of checking derivations is important. In section 4.2 I

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

[15] I count 108 rules in (Constable et al. 1986). There are several pages of informal mathematical text describing side conditions.

discuss styles of proof for feasible checking. Here I am interested in the formal system itself, and finding alternative presentations deriving the same judgements 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) that allows us to more feasibly check the original system in some indirect way. Two things can make a formal system computationally expensive to check: the derivations can be big and the side conditions (non-recursive premises) can be expensive to check.

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 does not change the derivable judgements and allows much smaller derivations; the system with cut is a better presentation (for formal mathematics) of the same consequence relation. Section 3.2.1 discusses how a simple proof checker can support meta-theoretic extensibility by adding admissible rules (such as cut) to a logic.

Another way to reduce the size of derivations is to eliminate duplicate subderivations. In a derivation tree, subderivations may have repeated occurrences at different places in the tree. By using a linear presentation of derivations, where each line names the previous lines it depends on, only one occurrence of each subderivation is required; the indirectness of naming lines allows sharing.[16] Extending a formal system with definitions allows a similar kind of sharing. In the cases just mentioned, we depend on ad-hoc identification of common substructures by the user, but some formal systems duplicate work in such a uniform way that we can give an alternative presentation that shares some common substructures by construction. Martin-Löf (1971) gives an algorithm for type synthesis that transforms official derivations to avoid duplication of context validity checking. This idea is used by Huet (1989) in the Constructive Engine, and abstractly explained and proved by Pollack (1994).

The other computational expense in checking a formal system is the side conditions. For example in CC the rule of type conversion has convertibility of two well-typed terms as a side condition. This is decidable, but certainly not feasible in general, so neither is proof checking. For the purpose of independent checking we can trade derivation size for computation of side conditions. For example if I have constructed some proof in CC, I can annotate the proof with the conversion paths found by my proof checker, which must be feasible if I did actually check the proof. (However, the size of the annotations may be prohibitive; we must work to find good annotations.) To independently re-check the proof, your checker need only follow the annotations (checking that they are legal), not discover a conversion path for itself. In this way, independent checking doesn't depend on heuristics for feasibility any more than for proof discovery.

Annotation of judgements can allow smaller derivations: with annotation, a full derivation of a judgement may be mechanically constructed from the judge-

[16] This interesting way of viewing linear derivations was pointed out to me by Harold Simmons. Automath used naming of expressions, lines, and contexts to avoid duplication.

ment itself so full derivations don't have to be constructed or communicated. This idea underlies the use of decidable type checking as a tool for proof checking; the *proof terms* are annotations that can be expanded into full derivations. Equivalently, we can think of omitting parts of official derivations that can be mechanically reconstructed; e.g. in CC, terms are essentially derivations with instances of the conversion rule and variable lookup elided. There is a tradeoff: 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).

4.2 Feasible formal proofs

In the preceeding section I discussed choosing formal systems that are suitable for actual checking. This section discusses how to make formal proofs, and more generally, whole bodies of formal mathematics, suitable for actual checking. In programming it is well known that there are feasible functions with infeasible implementations; e.g. the natural recursive definition of the fibonacci function is exponential in its input, while an alternative definition is linear. Analogously, even a well-behaved formal system will have proofs that are infeasible because of proof style. For example, if `ack` is the Ackermann function, trying to prove $\texttt{ack}(100) - \texttt{ack}(100) = 0$ by computing $\texttt{ack}(100)$ is hopeless, while proving $\forall n.n - n = 0$ is trivial, and gives a trivial proof of the goal.

Representation In proof, just as in programming, unsuitable representation of the objects of discourse is a cause of infeasibility. This is especially so because we are used to representations from conventional mathematics, which were never intended to be used in actual formalisation. When formalising a mathematical notion we make choices about representation, but there is no reason to believe there is a single "best" representation that leads to natural statements and short proofs. We can make several definitions for a concept, prove something about their relationship, and move between them as convenient; this is done implicitly in informal presentation. 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 vs. base representation for natural numbers. We probably want to use unary representation as the official definition of the naturals, and base representation for actual computation, e.g. the computational content extracted from constructive proofs. In order to do this, elementary school arithmetic must be formalised, i.e. the correctness of various algorithms for arithmetic operations on base representation numbers.

An often mentioned example of a notion that is hard to reason about formally is binding, and there are many representations of λ terms in the literature, including naive variable names (with or without a total ordering on variables), de Bruijn indexes, higher order abstract syntax (Pfenning and Elliott 1988), an axiomatic approach (Gordon and Melham 1996), and an approach using distinct

classes of free and bound names (Coquand 1991, McKinna and Pollack 1993). These representations are not all "isomorphic", e.g. some distinguish between α-equivalent terms, some do not. They have different theorems, e.g. a presentation of type theory using free and bound names (McKinna and Pollack 1993) has a thinning lemma which is close to the informal statement, while one using de Bruijn indexes (Barras 1996) requires explicit variable lifting.

By formalising the relationship between various representations, theorems can be stated and proved in natural forms, and used in different forms when needed. It may not be obvious what the official formalisation of some concept should be (e.g. are de Bruijn terms the "real meaning" of λ 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.

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 formalisation. An example of this is the use of formal topological models that has recently received interest in the type theory community. Following the ideas of Sambin (1995), Persson (1996) describes the formalisation of a completeness theorem for intuitionistic first-order logic using formal topological models. Coquand (1995) proposes a program of proof-theoretic analysis of non effective arguments using formal topological models. Such problems seemed impossible to formalise until this approach was developed.

4.3 The business of formal mathematics

Many people who pursue formal mathematics are seeking the beauty of complete concreteness, which contributes to their own appreciation of the material being formalised, while to many outside the field formalisation is "just filling in the details" of conventional mathematics. But "just" might be infeasible unless serious thought is given to representation of both the logic of formalisation and the mathematics being formalised. This can be viewed as an annoying hassle, or as the business of formal mathematics. The latter view leads us to be interested in areas of feasibility and expressiveness of formal systems and the power of formal systems to represent algorithms (Cardone 1995, Colson 1991, Fredholm 1995), and to seriously study formal representations of mathematical notions.

Bibliography

Allen, Constable, Howe and Aitken (1990). The semantics of reflected proof, *LICS Proceedings*, IEEE.

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

Barras, B. (1996). Coq en Coq, *Rapport de Recherche 3026*, INRIA.

Bevier, W., Hunt, W., Moore, J. and Young, W. (1989). An approach to systems verification, *Journal of Automated Reasoning* **5**(4): 411–428.

Bishop, E. (1967). *Foundations of Constructive Analysis*, McGraw-Hill, New York.

Boyer, R. S. (1994). A mechanically proof-checked encyclopedia of mathematics: Should we build one? can we?, *in* A. Bundy (ed.), *CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June/July 1994*, number 814 in *LNAI*, Springer-Verlag.

Cardone, F. (1995). Strict finitism and feasibility, *in* D. Leivant (ed.), *Logic and Computational Complexity. Proceedings, 1994*, number 960 in *LNCS*, Springer-Verlag.

Cohn, A. (1989). The notion of proof in hardware verification, *Journal of Automated Reasoning* **5**(2): 127–140.

Colson, L. (1991). *Repŕsentation Intentionnelle d'Algorithmes dans les Systèms Fonctionnels: Une ́tude de Cas*, PhD thesis, University of Paris 7.

Constable et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System*, Prentice–Hall, Englewood Cliffs, NJ.

Coquand, T. (1991). An algorithm for testing conversion in type theory, *in* G. Huet and G. D. Plotkin (eds), *Logical Frameworks*, Cambridge University Press.

Coquand, T. (1995). Formal topology and constructive type theory, Talk at *Twenty Five Years of Constructive Type Theory, Venice*.

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

Feferman, S. (1988). Finitary inductively presented logics, *Logic Colloquium '88, Padova*, North-Holland.

Fredholm, D. (1995). Intensional aspects of function definitions, *Theoretical Computer Science* **152**: 1–66.

Gordon, A. and Melham, T. (1996). Five axioms of alpha conversion, *in* J. Von Wright, J. Grundy and J. Harrison (eds), *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL, Turku, Finland*, Vol. 1125 of *LNCS*, Springer-Verlag, pp. 173–190.

Gordon, M., Milner, R. and Wadsworth, C. (1979). *Edinburgh LCF: A Mechanized Logic of Computation*, Vol. 78 of *LNCS*, Springer-Verlag.

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

Harper, R., Honsell, F. and Plotkin, G. (1993). A framework for defining logics, *Journal of the ACM* **40**(1): 143–184. Preliminary version in LICS'87.

Harrison, J. (1995). Metatheory and reflection in theorem proving: A survey and critique, *Technical Report CRC-053*, SRI Cambridge, UK.

Harrison, J. (1996). Formalized mathematics, *Technical report*, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland.

Horgan, J. (1993). The death of proof, *Scientific American* pp. 74–82.

Huet, G. (1989). The constructive engine, *in* R. Narasimhan (ed.), *A Perspective in Theoretical Computer Science*, World Scientific Publishing. Commemorative Volume for Gift Siromoney.

Kahrs, S. (1993). Mistakes and ambiguities in the definition of Standard ML, *Technical Report ECS-LFCS-93-257*, University of Edinburgh. A later addendum extends this report.

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

Martin-Löf, P. (1971). A theory of types, *Technical Report 71-3*, University of Stockholm.

McDowell, R. and Miller, D. (1997). A logic for reasoning with higher-order abstract syntax, *Proceedings of LICS'97*, IEEE.

McKinna, J. and Pollack, R. (1993). Pure Type Systems formalized, *in* M.Bezem and J.F.Groote (eds), *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93, Utrecht*, number 664 in *LNCS*, Springer-Verlag, pp. 289–305.

Milner, R., Tofte, M. and Harper, R. (1990). *The Definition of Standard ML*, MIT Press.

Nordström, B., Petersson, K. and Smith, J. (1990). *Programming in Martin-Löf's Type Theory. An Introduction*, Oxford University Press.

Paulson, L. C. (1994). *Isabelle: A Generic Theorem Prover*, number 828 in *LNCS*, Springer.

Persson, H. (1996). *Constructive Completeness of Intuitionistic Predicate Logic: A Formalisation in Type Theory*, Chalmers University of Technology and University of Göteborg, Sweden. Licentiate Thesis.

Pfenning, F. and Elliott, C. (1988). Higher-order abstract syntax, *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pp. 199–208.

Pollack, R. (1994). *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*, PhD thesis, University of Edinburgh.

Pollack, R. (1995). On extensibility of proof checkers, *in* Dybjer, Nordstrom and Smith (eds), *Types for Proofs and Programs: International Workshop TYPES'94, Båstad, June 1994, Selected Papers*, LNCS 996, Springer-Verlag, pp. 140–161.

Sambin, G. (1995). Pretopologies and completeness proofs, *Journal of Symbolic Logic* **60**: 861–878.

Slaney, J. (1994). The crisis in finite mathematics: Automated reasoning as cause

and cure, *in* A. Bundy (ed.), *CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June/July 1994*, number 814 in *LNAI*, Springer-Verlag.

VanInwegen, M. (1996). *The Machine-Assisted Proof of Programming Language Properties*, PhD thesis, University of Pennsylvania.

Recent BRICS Report Series Publications

RS-97-18 **Robert Pollack.** ***How to Believe a Machine-Checked Proof*****. July 1997. 18 pp. To appear as a chapter in the book** ***Twenty Five Years of Constructive Type Theory*****, eds. Smith and Sambin, Oxford University Press.**

RS-97-17 **Peter Bro Miltersen.** ***Error Correcting Codes, Perfect Hashing Circuits, and Deterministic Dynamic Dictionaries*****. June 1997. 10 pp.**

RS-97-16 **Noga Alon, Martin Dietzfelbinger, Peter Bro Miltersen, Erez Petrank, and Gábor Tardos.** ***Linear Hashing*****. June 1997. 22 pp. A preliminary version appeared with the title** ***Is Linear Hashing Good?*** **in** ***The Twenty-ninth Annual ACM Symposium on Theory of Computing*****, STOC '97, pages 465–474.**

RS-97-15 **Pierre-Louis Curien, Gordon Plotkin, and Glynn Winskel.** ***Bistructures, Bidomains and Linear Logic*****. June 1997. 41 pp.**

RS-97-14 **Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup.** ***Dictionaries on*** AC^0 ***RAMs: Query Time*** $\Theta(\sqrt{\log n / \log\log n})$ ***is Necessary and Sufficient*****. June 1997. 18 pp. Appears in** ***37th Annual Symposium on Foundations of Computer Science, FOCS '96 Proceedings*****, pages 441–450.**

RS-97-13 **Jørgen H. Andersen and Kim G. Larsen.** ***Compositional Safety Logics*****. June 1997. 16 pp.**

RS-97-12 **Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro.** ***Trans-Dichotomous Algorithms without Multiplication — some Upper and Lower Bounds*****. May 1997. 19 pp. Appears in Dehne, Rau-Chaulin, Sack and Tamassio, editors,** ***Algorithms and Data Structures: 5th International Workshop*****, WADS '97 Proceedings, LNCS 1272, 1997, pages 426–439.**

RS-97-11 **Kārlis Čerāns, Jens Chr. Godskesen, and Kim G. Larsen.** ***Timed Modal Specification — Theory and Tools*****. April 1997. 32 pp.**

RS-97-10 **Thomas Troels Hildebrandt and Vladimiro Sassone.** ***Transition Systems with Independence and Multi-Arcs*****. April 1997. 20 pp. Appears in Peled, Pratt and Holzmann, editors,** ***DIMACS Workshop on Partial Order Methods in Verification*****, POMIV '96, pages 273–288.**