Communicating and trusting proofs: The case for foundational proof certificates

Dale Miller

INRIA-Saclay & LIX/École Polytechnique, Palaiseau, France

1 Introduction

It is well recognized that proofs serve two different goals. On one hand, they can serve the *didactic* purpose of explaining why a theorem holds: that is, a proof has a *message* that is meant to describe the "why" behind a theorem. On the other hand, proofs can serve as certificates of validity. In this case, once a certificate is checked for its syntactic correctness, one can then trust that the theorem is, in fact, true. (For additional discussions of these two aspects of proof, see, for example, [4, 24].)

In this paper, we argue that structural proof theory and computer automation have matured to such a level that they can be used to provide a flexible and universal approach to *proof-as-certificate*. In contrast, the notion of *proof-as-message* is still evolving and deals with structures, such as diagrams and natural language texts [34], that are not yet well formalized.

Since the notion of proof-as-certificate is at times strongly debated in the literature, we discuss in Section 2 several aspects of proof in order to identify those situations in which certification by proof can prove valuable. After that discussion, we use the rest of this paper to outline more specifics of how proof theory can be used to provide for a foundational approach to the design of a universal notion of proof certificate.

2 Characterizing proofs and their roles

To understand the role and the nature of proof, we need to take a step back and review why proofs exist and how they are used. A key aspect of proof seems to be that they are documents that are communicated within a group of individuals (possibly separated in both space and time) in order to inspire trust.

2.1 Societies of humans and machines

Communications takes place within various "societies" comprised of individuals dedicated to common ends: such individuals can be human or mechanical. Admitting machines into such societies seems sensible in the many modern situations where computers are making decisions and are reacting to other individuals to further the goals of a society. We list here various kinds of societies of agents and some possible goals for them: while such societies may have several goals, we select here those goals for which a notion of proof is helpful.

1. A *sole mathematician* writes an argument that convinces herself and she then moves to address new problems. In this small society, a proof is a communication between the mathematician at one moment (the time she developed the proof) and some future time (when she works on the next problem). A goal of such a sole mathematician is to continue to develop a line of mathematical research.
2. A collection of *mathematician colleagues* searches for beautiful and deep mathematical concepts. The energies of such a group are put into finding good definitions and connections among ideas.
3. An *author of a mathematics text and his readers* is a society that is typically distributed by both geography and by time: the readers are located in a future after the text is written. The goal of this society is to have a successful *one-way communication*: that is, the author must be able to communicate with readers without getting feedback on how successful was the communications.
4. A group consisting of *programmers*, who are writing code for a popular operating system, and *users*, who are attempting to use that operating system on their computers, has a goal of producing quality software that the users find convenient and secure.
5. A *group of programmers, users, mobile computers, and servers* can form a society that exchanges money for various services (e.g., email, news, backups, and cloud computing).

Notice that in example 4, machines are not meant as individuals of the group: instead, they are tools used by the individuals. On the other hand, it seems appropriate to classify smart phones, electronic banking systems, and software servers all as individuals in example 5 since the choices and decisions that they take affect the goals of the society.

2.2 Proofs as documents communicated within societies

By logical formulas we mean the familiar notion of syntactic objects composed of logical connectives, quantifiers, predicates, and terms: these have, of course, proved useful for encoding mathematical statements and assertions in computational logic. Proofs can be seen as one kind of document that is communicated within a society of agents (human or computer) with the purpose of instilling trust in an assertion (written as a logical expression). We return to the example societies in the previous section and illustrate roles for proofs in them.

1. The only communication possible within a society consisting of a *sole mathematician* involves that mathematician telling a future instance of herself to trust that a certain formula is a theorem. If at some point in the future, that mathematician trusts her proof, she might take certain actions, such as developing consequences of that theorem.
2. Consider a group of *mathematician colleagues* such as the one featured in Lakatos's *Proofs and Refutations* [21]. This society interacts within a

lively and narrow spacial dimension with the agents sitting together discussing. The individuals also interact across time, of course, as new examples, counter-examples, definitions, and proofs appear. The goal of such a society of mathematicians might be to "develop deeper insights and understanding of geometry." The group exchanges messages and makes presentations. Proofs in this setting are generally informal since the energies of the group are put into exploring and discovering definitions and connections among ideas.

3. A society involving the *author of a mathematics text* and his *readers* generally involves a one-way communication: the readers will have the text only after the book is written and the readers may be physically and temporally remote from the author. A good example of such an author and text is, of course, Euclid and his *Elements*, which has been an important text for the communication of deep results about geometry to readers for two millennia.
4. A *group of programmers and users* of an operating system might need to circulate among its members many kinds of documents: bug reports from users should alert programmers to things that need to be fixed; programmer release new versions of software components; programmers exchange programs, scripts, and interfaces; etc. Some of these documents, such as interfaces, probably contain typing information, which can often be seen as formulas for which the program is a proof: type checking is then a simple kind of proof checking for simple assertions about the program. In addition, certain parts of an operating system can be so critical to the proper functioning of the operating system that a formal proof of some correctness conditions might be required: for example, it might be desirable for certain guarantees about device drivers (low level code used to control devices attached to a computer) to be formally verified by, say, a model checker [7].
5. A *group of programmers, users, mobile computers, and servers* can be seen as a society involving machines as individuals since the decisions and actions they make can help the groups achieve its goals. For example, a mobile phone might be expect to maintain certain security policies and this might mean that certain mobile code might not be downloaded to the phone. As a result, certain services might not be available to the user of that phone and some income for those services might be lost. If the infrastructure behind the movement of code allows for proofs to be attached to mobile code, the phone may allow the execution of mobile code if the phone could check that the attached proof proves certain security assertions of the code. The development of such an infrastructure has been studied under the title "proof carrying code" [30].

As these examples illustrate, societies circulate a wide variety of documents in order to help meet their goals. Of these many documents, proofs can be roughly identified as those that inspire trust in one agent of the conclusions drawn by another agent. One might acquire trust in a program in a number of ways that do not use proofs: for example, one's trust in a program might be inspired by

the fact that over its lifetime, no one has found errors in it: while such evidence is an important source of trust, it is not a document nor a proof.

2.3 Formality of proofs

Proofs can be divided into those that are informal and those that are formal.

We generally expect that *informal proofs* are readable by humans and are didactic. We also expect that they do not contain all details and that they may have errors. Informal proofs are circulated within societies of humans where they can be evaluated in a number of ways: Is the proof proving something interesting? Are the assumptions the right ones? Are the proof methods appropriate? Is this situation an example or a counterexample? If an informal proof is evaluated highly enough, more might be done with it: it might be written for a broader audience and it might be formalized. Typically, an informal proof will be made "more formal" when the group of people with which it is intended to communicate becomes larger and more diverse (involving greater separation in time and space).

A *formal proof* is a document with a precise syntax that is machine checkable: in principle, an algorithm should make it possible to "perform" the proof described in the document. We shall not assume that formal proofs are human readable or that they contain "explanations" of why a formula is actually true. Trusted computer tools are used to check proofs so that other human or machine agents come to trust the truth of a formula.

2.4 Revisiting criticisms of proof-as-certificate

Given the discussion about proofs above, it seems useful to now revisit some of the criticisms often leveled at proofs-as-certificates.

Consider, for example, two different societies discussed by Lakatos in *Proofs and Refutations* [21]. One such society is Euclid and the readers of his *Elements*. Here, Lakatos criticizes this text for "its awkward and mysterious ordering" of definitions and theorems. Euclid's text is notable for the society that it has served: given the vast number of readers of the *Elements* that have been distributed over both space and time, it seems that part of that text's success comes, in part, from its formal (sometime unintuitive) structure which increased its universality. Another society famously considered by Lakatos is that of a small society of mathematicians with limited distribution in time and space. In such a setting, communications can be informal and the society of mathematicians functions more as explorers of truth and good mathematical design. Even though these two societies involve only humans, their different distribution in time and space leads to rather different requirements on proofs-as-documents.

Consider now a society of agents involved with building and using an operating system. Clearly, the quality of the operating system is important: it should perform various duties correctly as well as maintain certain security standards. Such a society is highly dynamic: new features are added and others are removed; bugs are discovered and patches are issued; and the operating system must allow

for extension to its function by allowing new device drivers to be added or new executable code to be loaded and run. In such a setting, it seems futile to expect that there is a unique formal specification of the operating system to which members of the society are attempting to find a formal proof. None-the-less, informal proof and formal proof could still have some role to play among some agents of this society. Some programmers may want informal proofs that their programs satisfy certain requirements while other programmers might want to have completely formal proofs involving possibly weak properties of some other programs.

In light of this description of a society working to develop an operating systems, consider some of the criticisms of formal methods raised by De Millo, Lipton, and Perlis in [28]. They argued, for example, that formal verification in computer science does not play the same role as proofs do in mathematics: this certainly does not seem problematic because of the differences among the many agents in this society: informal proof may play an important role among some agents while formal proof may play an equally important role among other agents. Those components of an operating system that are static parts of many generations of such a system (such as, for example, sorting algorithms, file system functions, and security protocols) may need to be trusted at a level that formal verification could provide. Those components that are dynamic, experimental, and constantly changing would not be sensible targets for formal verification. De Millo, Lipton, and Perlis state that "Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection; we argue that insofar as it is successful, mathematics is a social, informal, intuitive, organic, human process, a community project." Given the richness of societies that are part of building large software systems, it seems clear that both views of proofs are important and both serve important roles.

If we allow for machine-to-machine communications of proofs, then formal proof can play a central role. The *proof carrying code* project of Lee and Necula [30] illustrates just such a situation. In that setting, a society of agents contains at least two machine agents, one that provides executable code and the other that is charged with permitting the accumulation of new code as long as that code maintains certain security assurances. Ensuring that security assurances are maintained requires some knowledge about the executable code. Examples of such assurance are that the code does not access inappropriate memory cells or that a typing discipline is maintained: e.g., that a "string" object is not transformed into, say, an "electronic wallet" object. The approach described by Lee and Necula requires that the executable code is paired with a formal proof that that code satisfies the necessary assurances: such a proof can be checked prior to accepting to execute the code.

To underline again the different roles of proof in different societies consider the following statement from Lakatos [21]: "'Certainty' is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge." While this criticism of formal proof sounds appropriate for those charged with the discovery

of mathematical concepts, it is not a valid criticism (nor was it intended to be) of those building safety critical software where formal proof can play an important role in establishing certainty [24].

2.5 Formal proofs and machine agents

While much of the value of proofs comes from sharing and checking them, the current state of affairs in computational logic systems makes exchanging proofs the exception instead of the rule. Many theorem proving systems use proof scripts to denote proofs and such scripts are generally not meaningful in other theorem provers: they may also fail to denote proofs for different versions of the same prover. There is also a wide variety of "evidence of proofs" that appear in computational logic systems: these can range from proof scripts to resolution refutations and tableau proofs to winning strategies in model checkers. It is hard to imagine that a given theorem prover would be able to read and check proofs in all these forms. Still, there is growing evidence that proofs need to be communicated between different computational logic systems: for example, an SMT prover is combined with the Isabelle prover in [14]. Generally, this work proceeds by integrating two specific provers: the general problem of integrating provers is seldom addressed.

In the remainder of this paper, we turn our attention to formal proof and how these can be designed to be universal and amenable to communicating and checking.

3 Formulas and logical interpretation

Before describing proof certificates in more specifics, we fix the language of formulas and inference rules that will hopefully allow a wide range of logics and proofs to be encoded naturally. In fact, Church's *Simple Theory of Types (STT)* [12] provides a syntactic framework for unifying propositional, first-order, and higher-order logics. Such formulas allow quantification at all higher-order types which in turns allows for rich forms of abstractions to be encoded. This framework also comes with an elegant and powerful mechanism for binding, quantification, and substitution by its incorporation of the simply typed λ-calculus into its equational theory. A remarkable feature of STT is that by making simple syntactic restrictions to the types of constants, one can restrict STT to propositional logic or to (multisorted) first-order logic. It is also immediate to add to formulas modal, fixed point, and choice operators. This choice of a framework for specifying formulas is not only one of the oldest such frameworks but also a common choice in several modern theorem proving systems.

Our approach to proofs of formulas depart from the simplistic setting of Church's original proposal where Axioms 1-6 described the logical core of higher-order logic and the remaining axioms enable mathematical theories by introducing extensionality, infinity, and choice. Instead, we mix Church's approach to formulas, bindings, and λ-calculus with the sequent calculus proofs provided by

Gentzen for classical and intuitionistic logics [15] and by Girard for linear logic [16]. In particular, the LKU proof system of Liang and the author [23] appears to be appropriate framework for specifying proofs in classical, intuitionistic, and linear logic.

For the rest of this paper, we shall then assume that we will be using a single language of logical formulas (namely, STT) and a single framework for describing proofs (LKU). We shall not, however, assume that the reader is intimately familiar with either of these two formalisms.

It is worth noting that we are not proposing to use the LF framework for specifying logics [18]. While LF can accommodate essentially the same formulas as Church's STT, the design of LF fixes a particular proof system (natural deduction for intuitionistic logic) which appears to lacks the kind of flexibility that we shall introduce in the next section.

4 Two desiderata for proof certificates

We shall now use the term "proof certificate" to mean a document that should denote a proof but for which some computation and search might be necessary to formally reconstruct that proof. We list now the first two of four desiderata for proof certificates.

> **D1:** *A simple checker can, in principle, check if a proof certificate denotes a proof.*

Proof checkers should be simple and well structured so that they can be inspected and possibly proved formally correct. The correctness of a checker should be much easier to establish than the correctness of a theorem prover: in a sense, a proof checker removes the need to have trust in theorem provers. The separation of proof generation from proof checking is a well understood principle: for example, Pollack [36] argues for the value of independent checking of proofs and the Coq proof system has a trusted kernel that checks proposed proof objects before accepting them [38]. Proof checking is likely to be at times computationally expensive, so different proof checkers may perform differently depending on the resources (say, memory and processors) to which they have access.

> **D2:** *The format for proof certificates must support a wide range of proof systems.*

In other words, a given computational logic system should be able to take the internal representation of the "proof evidence" that it has built and output essentially that structure as the proof certificate. Thus, this one proof certificate format should be usable to encode natural deduction proofs, tableau proofs, and resolution refutations, to name a few. Thus, if a system builds a proof using a resolution refutation, it should be possible to output a certificate that contains an object that is roughly isomorphic to the retained refutation.

A theorem prover is said to satisfy the "de Bruijn criterion" if that prover produces a proof object that can be checked by a simple checker [8]. Desiderata **D1** and **D2** together imply a "global" version of the de Bruijn criterion: if every theorem prover can output a proper proof certificate, then any prover can trust any other prover simply by using its own trusted checker. The tension between "simplicity" of the checker (**D1**) and the "flexibility" of the certificates (**D2**) is clearly a challenge to address. Section 5.1 briefly describes an approach to addressing this tension by identifying "macro" and "micro" inference rules and the rules that allow micro rules to be assembled into macro rules.

Before presenting two additional desiderata, we examine two implications of desiderata **D1** and **D2**.

4.1 Marketplaces for proofs

Formal proofs of software and hardware are developing some economic value. For example, some professional and contractual standards (for example, DefStan 00-55 of the UK *Defence Standards* [29]) mandate formal proofs for software that is highly critical to system safety (see [11] for an overview of such standards). The cost of going to market with a computer system containing an error can, in some cases, prove so expensive that additional assurances arising from formal verification can be worth the costs. For example, an error in the floating point division algorithm used in an Intel processor proved to be extremely costly for Intel: more recently, formal verification has been used within Intel to improve the correctness of its floating point arithmetic [19].

Where there is economic value there are opportunities for markets. If proof certificates satisfies desiderata **D1** and **D2**, it should be possible to develop a marketplace for proofs in the following sense. Assume that the ACME company needs a formal proof of its next generation safety critical system (such as might be found in avionics, electric cars, and medical equipment). ACME can submit to the marketplace a formula that needs to be proved: this can be done by publishing a proof certificate in which the entire proof is elided. The market then works as follows: anyone who can fill the hole in that certificate in such a way that ACME's trusted proof checker can validate it will get paid. This marketplace can be open to anyone: any theorem prover or combination of theorem provers can be used. The provers themselves do not need to be known to be correct. ACME must make certain that the proposed theorem that it places on the market is really the one that it needs. The people submitting completed proof certificate must also try to ensure that the ACME proof checker, with its restrictions on computational power, can perform the checking: otherwise they would not be paid for their proof certificate.

If someone working in the marketplace finds a counterexample to a proposed theorem, then that person should also get paid for that discovery as well. Similarly, partial progress on proving a theorem might well have some economic values. A comprehensive approach to proof certificates should formally allow counterexamples and partial proofs: we will not pursue these issues here.

4.2 Libraries of proofs

Once proof certificates are produced they can be archived within libraries. In fact, libraries might be trusted agents that are responsible for checking certificates. Since such checking is likely to be computationally expensive in many cases, libraries might be designed to focus significant computational resources (e.g., large machines and optimizing compilers) on proof checking. Once a proof certificate is checked and admitted to a library, others might be willing to trust the library and to use its theorems without rechecking certificates. To the extent that formal proofs have economic value, libraries will have economic incentives to make certain that the software that it uses to validate certificates is trustable. If someone else (a competing library, for example) finds that a non-theorem is accepted into a library, trust in that library could collapse along with its economic reason for existing. Libraries can also provide other services such as searching among theorems and structuring collections of theorems.

5 Two more desiderata for proof certificates

We shall now present two additional desiderata.

> **D3:** *A proof certificate is intended to denote a proof in the sense of structural proof theory.*

By "structural proof theory" we mean the literature surrounding the analysis of proofs in which restricting to analytic proofs (e.g., cut-free sequent proofs or normal natural deductions) still preserves completeness. For references to the literature on structural proof theory, see [15, 37, 39, 33]. Checking a certificate should mean that a computation on the certificate should yield (at least in principle) a formal proof in the sense covered by that literature.

This desideratum insists that certificates can be related to a well studied notion of proof and, as such, it should be possible to apply many well known and deep formal results from proof theory (cut-elimination, normalization, constructive content, etc) to certificates. For example, proof certificates might support the extraction of witnesses and, hence, programs: given a (constructive) proof of $\forall x.A(x) \supset \exists y.B(x,y)$ and a proof of $A(c)$, these two proofs together (via their certificate format) might be expected to yield a witness d such that $B(c,d)$ holds. Similarly, one might hope to do *proof mining* [20] with or *program extraction* [9] from proof certificates stored in a library. By using such sophisticated techniques for manipulating proofs, it should be possible to building browsers of certificates that would allow humans to interact with proof certificates in order to get a sense of their "message" (see Section 1).

Our final desiderata (**D4** below) addresses the fact that formal proofs can be large and that certificates must, somehow, allow proofs to be redacted. Large proofs will tax computational resources to store, communicate, and check them. Thus, any definition of proof certificates must provide some mechanism for making them compact even if the proof they denote is huge. One approach to making

proofs smaller could be "cut-introduction": that is, examine an existing proof for repeated subproofs and then introduce lemmas that accounts for the commonality in those subproofs. In this way, the lemma could be proved once and the various similar subproofs could be replaced by "cutting-in" instances of that lemma. There are clearly situations where cut-introduction can make a big difference in proof size. Proof certificates must, obviously, permit the use of lemmas (clearly permitted by desideratum **D3**). But this one technique alone seems unlikely to be effective in general since proofs without cuts (without lemmas) can be so large that they cannot be discovered in the first place. Our fourth desideratum suggests another way to compress a proof.

D4: *A proof certificate can simply leave out details of the intended proof.*

Things that can be left out might include entire subproofs, terms for instantiating quantifiers, which disjunct of a disjunction to select, etc. Thus, proof checking may need to incorporate proof-search in order to check a proof certificate that left out some details. As a result, proof checkers will not be simple programs that just check that all requirements of inference rules match correctly. Instead, they will need to be logic programming-like engines that involve unification and (bounded) backtracking search. An early experiment with using logic programming engines to reconstruct missing proof information was reported by Necula and Lee [31].

This desideratum forces the design of proof certificates in rather particular directions. While the other desiderata seems general and even obviously desirable, this fourth desideratum is the most distinctive in our proposal here.

5.1 Flexible description of proof systems

Taken together, desideratum **D2** and **D3** require that we can provide a rich set of *inference rules* similar to the *analytic* rules (introduction, elimination, and structural) used in proof theory. One way to achieve such richness is to identify a comprehensive set of "atoms" of inference as well as the rules of "chemistry" that allow us to build the "molecules" of inference. We briefly describe how such an approach might work: see [26] for more specifics.

The atoms of inference The sequent calculus provides an appealing set of primitive inference rules: these include the introduction of one logical connective and the deletion and copying (weakening and contraction) of formulas. Gentzen use this setting to distinguish classical and intuitionistic logic simply as different restrictions on structural rules [15]. Linear logic [16] provides a finer analysis of the roles of introduction rules and the structural rules: this analysis provides additional atoms of inference by, for example, separating connectives into their multiplicative and additive forms. The decomposition of the intuitionistic implication $B \supset C$ into $!\,B \multimap C$ is another example of this finer analysis of logical connectives. In order to capture inductive and co-inductive reasoning (including model-checking-like inference), the atoms of inference should also include fixed

points and equality [5, 6, 25]. Since the trusted proof checker needs to only implement the atomic inference rules, the checker can be simple in its design, thus satisfying **D1**.

The molecules of inference Without any additional structure, the structure of the atomic inferences within sequent calculus proofs is chaotic: the application of one inference can have little relationship with the application of any other inference rule. A well studied discipline for organizing atoms of inference into the molecules of inference is provided by the technical notion of a *focused proof system* [1, 22]. These proof systems attribute "polarity" to atomic inference rules. Atoms of the same polarity can stick together to form molecules: atoms of different polarities form boundaries between molecules. The resulting collection of molecules of inference form a proper proof system since they satisfy such properties as cut-elimination. In this sense, the resulting molecules of inference satisfy desideratum **D3**. There is also flexibility in how polarities are attributed so it is possible to "engineer" the set of molecules to cover a wide range of proof evidence, thus satisfying desideratum **D2**. Finally, when details of a proof are elided in a proof certificate (desideratum **D4**), the proof checker will need to conduct a search and that search should be understood as being conducted at the molecular and not atomic level: when filling in details to a proof, one should not be searching for new molecules via new combinations of atoms.

Since adequately representing one proof system within another proof system is central to our design of proof certificates, we expand on this topic next.

5.2 Three levels of adequacy

When comparing two inference systems, we follow [35] by identifying three "levels of adequacy." The weakest level of adequacy is *relative completeness*: a formula has a proof in one system if and only if it has a proof in another system. Here, only *provability* is considered. A stronger level of adequacy is of *full completeness of proofs*: the proofs of a given formula are in one-to-one correspondence with proofs in another system (such a correspondence must also be compositionally described). If one uses the term "derivation" for possibly incomplete proofs (proofs that may have open premises), an even stronger level of adequacy is *full completeness of derivations*: here, the derivations (such as inference rules themselves) in one system are in one-to-one correspondence with those in the other system. When claiming equivalences between proof systems, one should describe the level of adequacy of the associated correspondence: in general, we shall strive to always have proof certificates encode proof systems at the third and most demanding level of adequacy. These degrees of adequacy appear to correspond roughly to Girard's proposal [17, Chapter 7] for three levels of adequacy based on semantical notions: the levels of *truth*, *functions*, and *actions*.

The third level of adequacy (which, of course, implies the other two levels) is particularly significant here since it provides a sensible means for addressing desideratum **D4**. If a proof certificate elides an entire subproof then the proof

checker will need to reconstruct that subproof. The designer of the proof certificate presumably has elided that subproof because he feels that it is an easy proof for the proof checker to discover. This impression is only useful, however, if the search conducted by the proof checker (which strings together the atoms of inference) can be related directly to the search for the elided proof. This match must hold for successful applications of inference rules as well as for failing applications of inference rules. The notion of full completeness of derivations allows making this match.

6 Mixing computation and deduction

Proofs and computations have, of course, a great deal in common. The Curry-Howard Isomorphism views certain (constructive) proofs as programs. Here, we are interested in another connection between proofs and computation: that is, during checking of (or performing) a proof, certain computations must be made. For example, a condition on a step in a proof might require that a certain number is prime: such a condition can be established by a straightforward computation.

Proof checkers can be divided into those that rely solely on determinate (functional) computations and those that permit the more general, non-deterministic (relational) computations. Proof checkers of proofs in typed λ-calculi generally rely on extensive uses of β-reduction. Via the *deduction modulo* approach to specifying proof system, theories can, at times, be turned into functional computations that sit within inference rules [13]. The Dedukti proof checker [10] implements deduction modulo by compiling such computations into the functional programming language Haskell.

On the other hand, there are proof checkers that are built using non-deterministic search principles and that employ logic programming engines. For example, some of the early proof checkers [2, 3, 31] involved in the *proof carrying code* effort used logic programming based on (subsets of) higher-order logic [27]. These systems experimented with backtracking search (sometimes, even within the unification process). In one paper, the non-determinism inherent in a Prolog-based proof checker was resolved by supplying the checker with an oracle that was responsible for having all the answers to the question "I have several choices to consider, which should I take?" [32].

While placing significant amounts of computation (either functional or relational) into inferences seems necessary for capturing a wide range of proof certificates, this integration comes with some costs. First, one must accept that a compiler and a runtime system for a programming language are part of the trusted core of a proof checker. While compilers and interpreters for both functional and logic programming implementations are well understood, their presence in a proof checker will certainly complicate one's willingness to trust them. Second, proof checkers running on different hardware could have rather different resources available to them: thus, the computation required to check a proof might be available to one checker and not to another. This problem could be addressed by having a network of trusted libraries of proofs: such libraries could

publish theorems only after their own proof checkers have checked a given proof certificate. Such libraries could, of course, have computational resources available that might not be available on, say, desktop or mobile computers.

The use of proof search (non-deterministic, logic programming) to do proof checking may introduce some special issues of its own. Most logic programming engines (the efficient ones) usually come with a depth-first search strategy for building proofs. This style of search is notoriously poor when dealing with problems related to deduction. Since a proof checker is only rechecking or reconstructing a object that is already known to exist, the proof certificate could well come with useful bounds on how much search needs to be done in order to reconstruct a particular, elided subproof. For example, a depth-bound for a depth-first-search process should be a natural value to estimate when eliding an existing proof object. Another issue with using a non-deterministic proof checker is that there can be a mismatch between finding *the* proof or finding *a* proof: theorems generally have many proofs. Since a proof certificate might elide information, there is no guarantee that the proof the checker reconstructs is the original proof. This discrepancy does not appear to be serious since the proof checker will, at least, find a proof.

7 Conclusion

A proof is often expected to explain why a given theorem is true: such explanations are generally informal and flow from human to human. On the other hand, a proof can also serve as certification: such certificates are generally formal objects and flow from machine (the prover) to machine (the checker). We have advanced four desiderata for proof certificates and have outlined how results from structural proof theory and the automation of logic can be used to build certificates satisfying those desiderata. The resulting approach to proof certificates is based on *foundational* rather than *technological* considerations. There are several important consequences of having foundational proof certificates. First, one must not trust theorem provers but only proof checkers: since checkers are based on simple and universal proof principles, they should be much easier to check. Second, open markets for proofs can exist where those who need a proof can unambiguously request a proof of a theorem (via an empty certificate) and can unambiguously check that a proposed proof is, in fact, correct (using their own checker). Third, since proof certificates are not based on changing technological considerations but on a permanent foundation, libraries of proofs are possible. Such libraries offer the possibility to become trusted proof checkers as well as agents for structuring theories.

Acknowledgments. This research has been funded in part by the ERC Advanced Grant ProofCert.

References

1. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. Andrew W. Appel. Foundational proof-carrying code. In *16th Symp. on Logic in Computer Science*, pages 247–258, 2001.
3. Andrew W. Appel and Amy P. Felty. Lightweight lemmas in Lambda Prolog. In *16th International Conference on Logic Programming*, pages 411–425. MIT Press, November 1999.
4. Andrea Asperti. Proof, message and certificate. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - Proceedings of AISC, DML, and MKM 2012*, volume 7362 of *LNCS*, pages 17–31. Springer, 2012.
5. David Baelde. *A linear approach to the proof-theory of least and greatest fixed points.* PhD thesis, Ecole Polytechnique, December 2008.
6. David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), April 2012.
7. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proc. Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.
8. H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Transactions A of the Royal Society*, 363(1835):2351–2375, October 2005.
9. Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
10. Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo.* PhD thesis, Ecole Polytechnique, 2011.
11. J. P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium*, pages 168–177. IEEE Computer Society Press, 1993.
12. Alonzo Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.
13. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *J. of Automated Reasoning*, 31(1):31–72, 2003.
14. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS: Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.
15. Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969. Translation of articles that appeared in 1934-35.
16. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
17. Jean-Yves Girard. *Le Point Aveugle: Cours de logique: Tome 1, Vers la perfection.* Hermann, 2006.
18. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

19. John Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 113–130. Springer, 1999.
20. Ulrich Kohlenbach and Paulo Oliva. Proof mining in L_1-approximation. *Annals of Pure and Applied Logic*, 121(1):1–38, 2003.
21. Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.
22. Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
23. Chuck Liang and Dale Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011.
24. Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.
25. Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
26. Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 54–69, 2011.
27. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
28. Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the Association of Computing Machinery*, 22(5):271–280, May 1979.
29. U. K. Ministry of Defence. UK defence standardization, August 1997. DefStan 00-55.
30. George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pages 106–119, Paris, France, 1997. ACM Press.
31. George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *13th Symp. on Logic in Computer Science*, pages 93–104, Los Alamitos, CA, 1998. IEEE Computer Society Press.
32. George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.
33. Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001.
34. Roger B. Nelson. *Proofs Without Words: Exercises in Visual Thinking*. Mathematical Association of America, 1993.
35. Vivek Nigam and Dale Miller. A framework for proof systems. *J. of Automated Reasoning*, 45(2):157–188, 2010.
36. Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998.
37. Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.
38. The Coq Development Team. The Coq Proof Assistant Reference Manual Version 7.2. Technical Report 255, INRIA, February 2002.
39. Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.