About Trust and Proof: An experimental framework for heterogeneous verification

Farah Al Wardani$^{[0000-0003-1520-7090]}$,
Kaustuv Chaudhuri$^{[0000-0003-2938-547X]}$, and Dale Miller$^{[0000-0003-0274-4954]}$

Inria Saclay & LIX, Institut Polytechnique de Paris, Palaiseau, France

Abstract. Information and opinions come to us daily from a wide range of actors, including scientists, journalists, and pundits. Some actors may be biased or malicious, while others rely on physical measurements, statistics, or in-depth research. Some sources may be signed or edited, while others are anonymous and unmoderated. Trusting information from such diverse sources is a serious challenge facing society today. In this paper, we will describe another domain—the world of machine-checked logic and mathematics—in which many similar issues can appear but in which tractable solutions are possible. Many actors (people or software systems) assert that certain logical statements are theorems in this domain. We describe the Distributed Assertion Management Framework (DAMF) that explicitly manages claims by theorem provers that they have proved certain theorems from associated contexts. Provers willing to trust other provers will be able to avoid rechecking proofs.

1 Introduction

Confidence in formal methods to provide significant practical benefits in the construction of digital infrastructure goes back several decades and is illustrated by the following quote by Cliff Jones in 1987: "Of the many problems presented by the development of major computer systems, some can be ameliorated using formal methods [24]". Today's society is deeply integrated with powerful computer systems like the World Wide Web and cloud-based computing. While concerns about faulty implementations persist in the decades since 1987, novel challenges have emerged. A particularly pressing concern involves the trustworthiness of information and data that rapidly and fluidly traverses the globe. This paper considers how formal methods and trust might influence each other.

2 Trust crisis in the digital world

Trust in our understanding of how the world functions has been a long standing problem. An early chapter in systematically addressing such trust dates back to Sir Francis Bacon's introduction of the scientific method—with its focus on reproducible results—and the creed *Nullius in verba* (take no one's word for it): that is, before trusting something, check it for yourself. In human affairs different from those involving scientific experimentation and the analysis of data, other methods of gaining trust involved the inventions of such social institutions as judges, magistrates, and jury trials. In recent centuries, trust in the world of politics and foreign affairs is often offered by a limited number of media organizations acting as gatekeepers of information for which there was an economic incentive to maintain the trustworthiness of their media products.

In today's internet-dominated world, information flows freely and without gatekeepers, but trust is scarce. Our current experience of attempting to trust information in the internet era is made worse by the existence of various individuals, groups, and governments who deliberately carry out propaganda and misinformation campaigns. For example, the RAND Corporation described Russian propaganda tactics used during the 2016 USA presidential election as a "Firehose of Falsehood" [36]. This technique involves generating a large volume of false or misleading information and spreading it rapidly and repeatedly across multiple channels, such as newspapers, social media, and online forums. Often, the goal of this tactic is not to convince an audience about specific policies but rather to overwhelm the audience, sow confusion, and make it difficult to distinguish facts from fiction. There are also perverse financial incentives for media agencies to fuel misinformation campaigns by prioritizing clicks and revenue over truth.

While Internet technology has enabled the rapid composing and global distribution of information and misinformation, it has also created another shift in the media world: almost all media is now in electronic form. This shift makes it possible to consider the following approach to addressing misinformation.

> Agents should cryptographically sign the information sources they produce. Consumers of information should maintain curated *allow-lists* of agents they have explicitly or provisionally chosen to trust.

Of course, the remarkable naivety of this approach will certainly stop people from considering it seriously. However, history shows that exploring "naive" solutions can lead to unexpected breakthroughs. Consider the following two seemingly naive-sounding ideas: both are problems that the digital era has forced society to consider and for which digital solutions have been proposed.

– **Problem:** Your mobile phone gives out too much information about you and your location. **Solution:** Have your phone lie for you. This approach is at the heart of *differential privacy*, which studies how the degree of lying can affect the utility of the data collected from multiple users [17].
– **Problem:** The binary file you plan to download could be a security risk on your computer. **Solution:** Require that that code is paired with a formal proof that it is not dangerous. This approach has been studied under the title *proof-carrying code* [32].

We now add the following problem and solution pair.

– **Problem:** Worried that the documents you get are forged, fake, or generated by an internet bot farm. **Solution:** Have all documents cryptographically signed by their authors. This problem and the proposed solution is the starting point for this paper.

It is worth noting that the sign-everything-by-trusted-parties approach is used in some computer systems. For example, the secure booting of computers can be accomplished with the UEFI (Unified Extensible Firmware Interface) framework, which is designed to prevent the loading of unauthorized software during the boot process of a computer. All code that is eventually loaded into a machine's firmware and memory is signed by agents whose public keys are on an allow-list [34]. Another example is Debian's SecureApt, which secures the `apt` manager in Debian Linux distributions. It uses cryptographic signatures to verify the integrity and authenticity of packages downloaded from software repositories, ensuring that what is installed is genuine and unmodified software [14].

3 A shift in scope to trust within theorem proving

We (the authors) have insufficient expertise to address the crisis in trust described above. This crisis is vast and multifaceted: if a comprehensive solution is possible, then expertise beyond computer science will certainly be needed. However, since our expertise is limited to computer science, we will significantly narrow the scope of our focus in this paper. In particular, some problems surrounding trust in the digital world reappear in mechanized proof-checking systems, where we have more expertise. This paper focuses on the following two goals: (1) to determine how trust within the theorem-proving community can be addressed, and (2) to explore the costs and benefits of a particular approach to managing trusting relationships.

The beginning of *proof checking* can be traced back to at least 1666, when Leibniz envisioned that there could be a universal symbolic language (*characteristica universalis*) for stating propositions and that two people who were arguing over the veracity of some particular statement could agree on *calculemus* ("Let us calculate"). The result of such a calculation would indicate which person was making the correct statement [40].

A more modern effort at proof-checking can be found in the work of Gordon, Milner, & Wadsworth in 1979, where proofs in the *logic for computational functions* (LCF) were built in a programmable system using an early version of the ML functional programming language [20]. Since those early days, a great many interactive theorem-proving systems have been built, a short list of which includes nqthm (a.k.a. Boyer-Moore theorem prover) [11], Isabelle [33,37], Coq [10], HOL [21], PVS [35], Abella [7], and Lean [28].

It is now common to hear of large and complex formal proofs being built for mathematical theorems or computer system properties. Such proofs can involve many people working over many years. Even in that setting, most of the theorem provers used in such tasks are *autarkic*: they only trust their proof-checking kernels. Conversely, some theorem provers in the domain of program verification have been designed to exploit and explicitly trust specific, special-purpose theorem provers. For example, the Why3 prover relies on external theorem provers, such as Coq, and SMT solvers, such as CVC4 and Z3, to discharge verification conditions. Also, the TLA+ Proof System (TLAPS) relies on back-end provers, such as Isabelle and Zenon, and SMT solvers, such as CVC3, Yices, and Z3. In general, however, provers are not designed to trust other provers.

The implemented logical framework Dedukti [6] is an interesting component in the space of theorem-proving systems. Dedukti provides a simple but expressive logic and proof system based on a small logical core (dependently typed λ-calculus with rewriting). Its simplicity makes it relatively easy to implement a proof checker and trust its correctness; in particular, a skeptic could re-implement it. This system can provide an independent, secondary proof checker for other provers that can output significant parts of their proof libraries (e.g., HOL, Isabelle, Coq) [15]. Such independent proof checking offers more confidence in proofs. In practice, however, once a proof is available in a rather explicit and straightforward format such as that offered by Dedukti, it is not a big step to formally print Dedukti proofs into a range of other proof formats. Thus, if prover A wants to use a proof by prover B, the latter prover outputs its proof to Dedukti, which can then print that proof in a format that prover A can check. As a result of this role of Dedukti, prover A does not need to trust either Dedukti or prover B. Thus, Dedukti can be used to maintain the autarkic environment of provers. The framework we describe in this paper is orthogonal to systems like Dedukti since we may wish to trust a theorem prover even if no formal proof certificate is made available.

4 "Trust requires proof" vs "Proof requires trust"

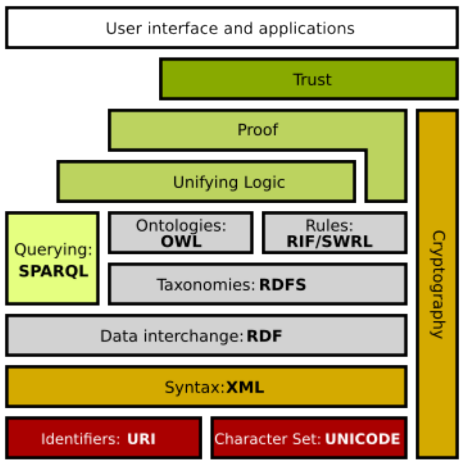

Fig. 1. The semantic web stack

We are all familiar with the implication that the existence of proofs can instill trust in the veracity of statements. This "trust requires proof" perspective has been a part of mathematics since at least the times of Euclid. This perspective has also been promulgated in the design of the *semantic web stack* [9], which is displayed in Figure 1. In that stack, cryptography provides some notion of trust (via digital signatures, for example) while proofs of logical statements based on taxonomies and database queries provide the bulk of trust. Presumably, the proofs involved in the semantic web will be significantly more shallow than those involved in modern mathematics and program verification.

In the rest of this paper, we shall, however, explore the converse perspective, namely, that "proof requires trust". This perspective arises from the following two facts.

1. Formal proofs are generally large and detailed objects; they can only be checked by computer programs.
2. Computer programs and their executions can be wrong.

Indeed, carefully designed and constructed proof checkers have been found to have errors, usually leading to proofs of false. We must speak explicitly about trusting proof checkers.

5 Establishing a design in a distributed context

In this section, we reiterate over the analogy made in Section 3, between problems surrounding trust in the digital world and in the world of mechanized proof-checking systems. We illustrate how they should be considered similarly in a distributed and heterogeneous setting, and then set the stepping stone for our investigation.

The following example illustrates a typical approach to using a proof assistant. Consider that one wants to build a formal and machine-checked proof that the number $\zeta(3) = \sum_{i=1}^{\infty} \frac{1}{i^3}$ is irrational. (A three-page proof outline was published in 1978 by Apéry [5] and a more complete proof was eventually developed by others; see [39] for a description of that development). The process of developing a formal proof of this result is described in [29]. As documented there, a particular prover is chosen (in this case, Coq), along with a specific set of definitions and theorems already verified within Coq (in this case, the Mathematical Components libraries). While the authors utilized additional computer tools like Maple to help organize the Coq proof; these tools were not relied upon for verification. In the end, all computation and deduction steps were achieved within Coq, and, as a result, only Coq needed to be trusted for the verification of Apéry's Theorem.

5.1 Considerations for distribution and heterogeneity

As exemplified by the scenario above, the traditional approach to mechanized theorem proving in mathematics is autarkic. In today's world, alternatives to centralized systems and authorities are desirable and achievable using existing and well-understood technologies. We briefly describe three reasons why heterogeneity and distribution are worth considering in the mechanized theorem-proving context.

1. Logic and proof serve diverse purposes, ranging from their uses in programming, type systems, model checking, SAT and UNSAT solving, and mechanized theorem proving. Expecting a single proof-checking kernel to handle all of these uses equally seems undesirable since the computational demands of checking proofs in these various domains can vary greatly. For example, type checking generally requires unification, while checking UNSAT proofs often requires optimizing memory storage. Expecting one checker to provide good performance on both of these dimensions while also being simple enough to inspect for a lack of errors seems unreasonable.

2. Generally, people already trust several agents in a passive sense. For instance, users within the Coq community that use the Coq standard libraries do not believe or claim that the theorems developed by the Isabelle community, curated in the Isabelle standard libraries, are wrong. We propose extending this trust paradigm to an active form, enabling users to reference and depend explicitly on externally proved theorems. Such enabling leads to distributed trust, where multiple entities, not just a single one, contribute to the trust foundation of a formal development.

3. Having advocated for enabling a common context that incorporates diverse expression forms, verification procedures, and interacting agents, it naturally leads us to consider an alternative approach for constructing libraries of formal developments. In this new paradigm, structures become emergent and interconnected rather than hierarchical and isolated. Provers, previously packaged with their own libraries, transition to the edge, acting as tools for certifying results. Consequently, curation processes no longer inherently depend on specific provers. Instead, the fact that a theorem within a curation has been formally certified, whether by a single agent or a combination of agents and tools, becomes simply another piece of information to be conveyed.

5.2 Off-the-shelf, enabling technologies used by DAMF

After establishing why we are addressing the *non-autarkic* approach to theorem proving in a *distributed* and *heterogeneous* context, we proceed by asking two simple questions: How can we allow a user of one theorem prover to be able to reuse and refer to a theorem proved in another theorem prover by another *agent*? How can such a scenario keep track of dependencies and multiple agents in a clear, verifiable, reproducible, and, thus, trustable manner?

We now describe our system, the *Distributed Assertion Management Framework* (DAMF) [4], which provides structures that track who and what is being trusted. As its name implies, DAMF centers around the concept of assertions. When an agent produces a result or makes a claim, that agent generates an assertion to allow others to do one of the following. *Reuse the information with trust*: this option involves trusting the agent and accepting the information without further verification. *Simply reference the information*: users can refer to the information without necessarily trusting the agent. An assertion acts like a stamp attached to the information, signifying the agent's statement: "You can trust me if you want to use this information without further verification." Technically, an assertion is a signed claim. DAMF employs various readily available technologies, the most important of which are described next.

Public-key cryptography Assertions link two pieces of information: *who* makes a claim, and *what* is claimed. In DAMF, the *who* is called an *agent*, and an agent is identified by a public-private key pair. The private key is used to sign the content of the claim, and the public key is used to validate that signature. Public-key cryptography is used in DAMF because it is tamper-resistant and associates the signature with a globally identifiable signer. As such, assertions are independent information units that do not need to be tracked back to their source for verification. Typical examples of agents are human users or automated proof-checking services provided by a cloud computing platform. Different users using the same tool are naturally considered distinct agents. On the other hand, a single user using different tools may be associated with multiple keys, considered as multiple agents.

Content-addressable, distributed storage: IPFS Sharing information in a distributed setting traditionally relies on the internet, particularly the web, where resources are referenced by URLs pointing to their location on specific servers. However, this use of the web is problematic for at least a couple of reasons: the content of a URL can be altered, compromising the trustworthiness of any links pointing to it, and server outages can render information inaccessible. Content-addressed storage offers a solution to those shortcomings since it identifies information using a cryptographic hash, essentially a unique digital fingerprint. This use of hashes separates information from its physical location, eliminating the issues associated with location-based addressing. This decoupling also aligns well with the considerations of DAMF, as will be elaborated in subsequent sections. These sections will define specific objects, each uniquely identified by its content and equally retrievable from multiple locations. Issues arising from potential name conflicts (for instance, two different objects being given the same name by two agents unaware of each other) and with circular dependencies are naturally avoided.

The InterPlanetary File System (IPFS) [8] provides the necessary infrastructure to interact with and leverage a content-addressed protocol within our DAMF implementation. The next section showcases an example of an assertion object in IPFS, identified by its Content Identifier (`cid`) and represented as JSON:[1]

```
// cid: bafyreiek2t75whn7gi6ygrymegguescqi4iudjj56ui..
{ "format": "assertion",
  "agent": "-----BEGIN PUBLIC KEY-----\nMFIwEA...",
  "signature": "3040021e10db76a6606d7a813747849028c79e..."
  "claim": {"/": "bafyreibvtxzqhvht5rfxpw3rkgx..." } }
```

Notice the `"claim"` attribute: its value represents an *IPLD Link*[2] to another object stored in IPFS. The above *assertion* object represents a DAG with nodes that can be traversed from the root object or accessed separately by their `cid`: adding `/claim` to the mentioned link in the footnote returns the referenced *claim* object, and so on. This use of linked data proves to be an essential enabling mechanism in the implementation of DAMF. For instance, presenting the *claim* as a separate, independent object *linked* from the assertion structure instead of being included directly within it allows a clear, transparent representation of multiple agents asserting the same claim.

[1] Can be accessed through this link.
[2] InterPlanetary Linked Data; a way to represent linked data in IPFS

6 Designing DAMF: structures and main concepts

We mentioned in Section 5 that *assertions* are the principal concept in DAMF. The main kind of assertions addressed in our current development is meant to denote asserting whether a *formula* is a proved *theorem*, a *conjecture*, or a theorem that depends on some *lemma*, where that lemma is also a *formula*. We thus need a clear representation of a *formula* object, which we describe first.

6.1 Languages, contexts, and formulas

To be as general as possible, we represent the formulas used in assertions as *strings*, i.e., in a format suitable as an input to a parser of the source proof system. In order to determine that the input is well-formed, the source proof system may need further information about the *features*—symbols, predicates, functions, types, notations, hints, etc.—used in the formula. Such additional information is the *context* of the formula, which we represent as a document fragment in the language of the source proof system.

For example, take the following theorem written in `Coq 8.16.1`:

```
1 Definition lincomb (n j k : nat) := exists x y, n = x * j + y * k.
2 Theorem ex_coq : forall n:nat, 8 <= n -> lincomb n 3 5.
```

The formula corresponding to the theorem `ex_coq` is the literal string `"forall n:nat, ··· lincomb n 3 5"`. The symbols `8`, `<=`, etc. are part of the standard prelude of this language, and the symbol `lincomb` is defined in line 1, so a sufficient context necessary for `Coq 8.16.1` to parse and type-check the theorem statement is the text of line 1, which is also written in the `Coq 8.16.1` language.

Abstractly, a *formula object* in DAMF is a triple (L, Σ, F) where L denotes a *language*, Σ denotes a *context*, and F denotes a *formula*, all of which may conceptually be thought of as strings. We will use the schematic variable N to range over such formula objects. The language L is a canonical identifier (specifically, the `cid` of a DAMF language object) which may optionally represent information about a suitable loader for the language that will make sense of the strings Σ and F; DAMF compares languages just by their identifiers. Moreover, L is interpreted as defining all the globally available features; for instance, the symbol `nat` is part of the standard prelude of this version of Coq and should therefore be understood as being defined in the language `Coq 8.16.1`. The context Σ introduces any user-defined features such as the definition `lincomb` above that is not part of Coq's standard prelude.

Note that DAMF formula objects are considered to be *closed*, i.e., every symbol used in the formula is defined in the language or the context. From the perspective of DAMF, a formula object is an atomic entity. Additionally, DAMF does not need to be aware of any reasoning principles of the language or context components. For instance, no mechanism in DAMF would allow the substitution of a declared symbol in the context with a concrete definition. The purpose of differentiating a formula object into three parts is purely pragmatic: the language part will in most cases be a well known object used by many agents, and the context part may potentially be shared between multiple assertions. DAMF consumers may be able to use this sharing of information to consolidate tasks such as context-processing.

6.2 Sequents, productions, and assertions

A *sequent* in DAMF is abstractly of the form $N_1, \ldots, N_k \vdash N_0$ where each of the N_i is a DAMF formula object defined in the previous subsection. We will use the schematic variable Γ to range over ordered lists of formula objects, and S to range over sequents. In a sequent $\Gamma \vdash N$, we say that N is the *conclusion* and Γ are the *dependencies*. Such sequent objects may be produced whenever a formal proof has been checked in a proof checker: the conclusion represents the statement of the theorem, and the dependencies are external lemmas that were used during that proof.

A sequent is a purely mathematical object: if a reader knows the languages the formulas in the sequent are built from, they can understand the meaning of the sequent. Most sequents will be *produced* by particular agents and using particular tools. Thus, DAMF has the concept of a *production* object that enriches an underlying DAMF sequent with the metadata necessary to reliably determine how the sequent was produced. This is known in DAMF as a *mode*, which can be one of the following:

- `null`, which denotes the *absence* of any mode information. If an agent signs such a production, then they assume direct and full responsibility for its correctness.
- a *tag* such as `"axiom"` or `"conjecture"`, which is like `null` except that the intended purpose of the sequent is made clear. For instance, a `"conjecture"` tagged production would mean that any agent who signs that assertion does not assert its truth directly.
- a *tool* link, which would generally be a link to a DAMF *tool* object that we do not elaborate further in this paper. Suffice it to say that any well known system would have a similarly well known DAMF tool object.

We write DAMF productions abstractly as $\Gamma \vdash_M B$ where $\Gamma \vdash B$ is an abstract DAMF sequent and M denotes the mode.

The mere fact that a production is tagged with a tool mode does not guarantee that the tool indeed was used to produce the sequent in the first place. In DAMF, the only entities that can make such guarantees are *agents*. Abstractly, an agent is a globally unique name; we use the schematic variable K to range over agents. Agents will be implemented as public-private key pairs.

We define a simple multi-sorted first-order logic where agents and sequents are primitive sorts and where the infix predicate *says* is the sole predicate; the atomic formula K *says* P, where K is an agent and P a production, is an *assertion*. The *says* predicate is implemented in DAMF using public-key cryptography. In a DAMF-aware proof system, the assertion K *says* $(N_1, \ldots, N_k \vdash_M N_0)$ is interpreted as follows:

– The pair (K, M) of the agent and the mode is treated as *trusted*. If the agent cannot be trusted for some reason, such as if K occurs in a deny list, then the assertion is unusable. Likewise, if M cannot be trusted in isolation, such as if M denotes a version of a tool known to be unsound, then the assertion is unusable. Separating agents and modes allows consumers of assertions to tailor their trust parameters to particular agents; for example, they can trust all modes for a reliable agent, or they can restrict a given agent to only those modes that they are reliable with. Note that the agent signs the entire production, including the mode, to prevent tampering of the assertions by third parties.
– The conclusion of the assertion, N_0, contains the formula representing the lemma to be used in the DAMF-aware proof system. Note, in particular, that the dependencies $N_1, \ldots, N_k$ are not relevant when using this assertion as an external dependency.

In the remainder of this paper, we will omit the modes unless relevant to simplify the presentation.

6.3 Adapters

Because every formula object packages the formula together with its context and language identifier, every formula object is independent of every other formula object. Thus, in a sequent $N_1 \vdash N_0$, there is no requirement that the conclusion N_0 and the dependency N_1 be in the same language or have a common context. When working within a single autarkic system (e.g., a proof checker using a single logic), the sequents that are generated for every theorem will probably place the conclusion and dependencies in the same language and context; however, in the wider non-autarkic world, we can use multilingual sequents as first class entities that are documented and tracked the same way as any other kind of sequent.

An important class of multilingual sequents comes from *adapters*. In order for a theorem written in the `Coq 8.16.1` language to be used by a different system with a different language, say `Abella 2.0.9`, we will need to transform the formula objects in the former language to those in the latter language. This kind of translation is an example of a *language adapter*, which falls into the general class of *adapters*, and which creates a sequent by translating between languages or modifying the logical context by standard logical operations such as weakening (adding extra symbols), instantiation (replacing a symbol by a term), or unfolding (replacing a defined symbol by its definition).

As an example, the `Coq 8.16.1` example above can be translated to the `Abella 2.0.9` language as follows, where the function symbols `+` and `*` are replaced by relations in Abella.

```
Import "nats". % some natural numbers library
Define lincomb : nat -> nat -> nat -> prop by
  lincomb N J K := exists X Y U V,
    times X J U /\ times Y K V /\ plus U V N.
Theorem ex_ab : forall n, nat n -> le 8 n -> lincomb n 3 5.
```

Lines 1–4 determine the context $\Sigma_{\texttt{ex\_ab}}$ for the formula `ex_ab` on line 5. The sequent that represents this translation therefore has the form

$$\big(\texttt{Coq 8.16.1}, \Sigma_{\texttt{ex\_coq}}, \texttt{ex\_coq}\big) \vdash (\texttt{Abella 2.0.9}, \Sigma_{\texttt{ex\_ab}}, \texttt{ex\_ab}).$$

Suppose agent K_1 signs this translation and that agent K_2 signs the sequent $\vdash \big(\texttt{Coq 8.16.1}, \Sigma_{\texttt{ex\_coq}}, \texttt{ex\_coq}\big)$. As long as K_1 and K_2 are trusted by the user of `Abella 2.0.9`, then the formula object $(\texttt{Abella 2.0.9}, \Sigma_{\texttt{ex\_ab}}, \texttt{ex\_ab})$ can also be treated as a theorem by that user thanks to *composition*, discussed next.

6.4 Composing assertions, trust

Assertions will be composed by means of a single rule of inference that implements a cut-like rule for sequents, Compose.

$$\frac{K\ \textit{says}\ (\Gamma_1 \vdash M) \qquad K\ \textit{says}\ (M, \Gamma_2 \vdash N)}{K\ \textit{says}\ (\Gamma_1, \Gamma_2 \vdash N)}\ \textsc{Compose}$$

The effect of this rule means that the *says* predicate does not correspond one-to-one with cryptographic signatures. The conclusion of the Compose rule may not be a sequent explicitly signed by agent K even if both premises are. Instead, the rule states that whenever K can be said to reliably claim, *either* by a cryptographic signature *or* by a Compose-derivation tree, that both $\Gamma_1 \vdash M$ and $M, \Gamma_2 \vdash N$, then K must also reliably claim $\Gamma_1, \Gamma_2 \vdash N$.

There are many variations to *access control logic* in the literature. For example, some such logics use inference rules such as:

$$\frac{\Gamma \vdash N}{K\ \textit{says}\ (\Gamma \vdash N)} \quad \text{or} \quad \frac{K\ \textit{says}\ (\Gamma \vdash N)}{K\ \textit{says}\ (K\ \textit{says}\ (\Gamma \vdash N))}.$$

Such rules are neither syntactically well-formed nor desirable for our purposes. We use here a very weak access control logic (see [1] for a survey of such logics). Instead, checking the validity of a given derivation using Compose is computationally trivial: each instance of it must eliminate exactly the leftmost dependency in the second premise, which is a DAMF formula object that is compared by `cid`.

Observe that the agent K does not participate in a meaningful way in a derivation that is built with the Compose rule. Thus, for a given end sequent of the form K *says* $(\vdash N)$, a Compose derivation can be seen as a *proof outline* for the desired theorem N, with the leaves of the derivation being the assertions that need to be sourced from an assertion database (such as the DAMF global store). We say that an assertion (K *says* S) is *published* if it can be retrieved from such a database. The inference system is then enlarged with the following rule that can be used to complete the open leaves of the Compose derivation using assertions made by different agents.

$$\frac{(K_1\ \textit{says}\ S)\ \text{is published}}{K_2\ \textit{says}\ S}\ \textsc{Trust}\ [K_1 \mapsto K_2]$$

This rule is parameterized by a pair of agents, K_1 and K_2, and is understood to be applicable only when K_1 is in the user-specified *allow list* of K_2 (i.e., K_1 *speaks for* K_2, which we write as $[K_1 \mapsto K_2]$).

We do not assume that agents have any additional closure properties beyond COMPOSE and TRUST. For example, suppose N_A, $N_{A\to B}$, and N_B are the formula objects that correspond to the formulas A, $A \to B$, and B respectively in some language. We do not assume that the following rule is admissible:

$$\frac{K \textit{ says } (\Gamma \vdash N_{A\to B}) \qquad K \textit{ says } (\Gamma \vdash N_A)}{K \textit{ says } (\Gamma \vdash N_B)} \;\textsc{mp}.$$

That is, we do not assume that the formulas asserted by agent K are closed under modus ponens. Similarly, we do not assume that what agents assert are closed by substitution or instantiation of any symbols that are defined in the contexts of the formula objects. While a particular agent may not be closed under modus ponens, substitution, or instantiation, it is possible to employ other agents that can look for opportunities to apply such inference rules on the results of trusted agents. In particular, if we want the query engine to be able to use the MP rule, then the engine must construct an agent K_{MP} whose sole function is to generate assertions such as K_{MP} *says* $(N_{A\to B}, N_A \vdash N_B)$ that correspond to applications of the MP rule. Of course, K_{MP} will need to be in the *allow list* for any agent wanting to use this agent.

7 Experiment: a heterogeneous verification using **Abella**, **Coq**, and **λProlog**

This section presents a complete example of proving the following theorem in Abella [7] by trusting external lemmas from Coq and λProlog [30]:

> For $n \in \mathbb{N}$, $\texttt{fib}(n) = n^2$ if and only if $n \in \{0, 1, 12\}$ where $\texttt{fib}(n)$ denotes the nth Fibonacci number.

The purpose of this example is to illustrate the communication with DAMF and the various edge provers, so the theorem itself is not particularly challenging. Nevertheless, a complete proof of this theorem inside Abella would currently require formalizing a sizable amount of integer arithmetic, not to mention automated tactics for reasoning about arithmetic. Since Coq has these components already, we will use Coq to prove the following theorem by making heavy use of its linear arithmetic solvers:

> For $n \in \mathbb{N}$, if $n \geq 13$ then $\texttt{fib}(n) > n^2$.

On the other hand, we will use λProlog to find all the pairs $\langle n, m\rangle$ where $\texttt{fib}(n) = m$ and $n \in \{0, 1, \ldots, 12\}$. We could of course have used Coq to perform these computations as well, but it is pedagogically useful to see an example that combines both functional and relational programming.

Our DAMF implementation's architecture consists of a global layer stored in IPFS. This layer stores DAMF global objects, such as formulas and sequents, that any participating agent can create, publish, and access. Instead of requiring provers to interact directly with IPFS, we have implemented Dispatch, an optional intermediary tool. Dispatch acts as a bridge between edge provers/tools and the global layer. It provides standardized input and output formats corresponding to specific DAMF objects, simplifying system interaction. A DAMF-aware prover must only build and parse specific JSON objects into and from theory files. Dispatch is designed for human and tool use and currently implements the publishing and retrieval of DAMF objects to and from the global store. It also implements a lookup facility that can enumerate paths that yield some theorem starting from a given set of assertions and report the pairs *(agent, mode)* corresponding to assertions along each path, along with any remaining dependencies. Both DAMF global objects and the dispatch tool are designed to be adaptable and expandable.

Further details on this example can be found through the *distributed assertions website.*[3]

7.1 Setup in Abella

Abella has no built in notion of natural numbers. We therefore begin an Abella development (in a `.thm` file) by declaring the `nat` type together with its constructors `z` and `s` to obtain a unary representation for natural numbers. The Abella type system is only used for syntactic checks and yields no induction principles for logical reasoning, so we have to define an auxiliary inductively defined relation, also called `nat`, that is used for inductive reasoning. In Abella, the namespaces of types and predicates are separate, so the same name `nat` can be used both for type names and for predicate names. Finally, because Abella uses only λ-equivalence as its equational theory of simply-typed λ-terms, we will have to capture recursive computations in the form of relations; thus, operations such as addition and multiplication, and relations such as $\leq$, are defined using inductively defined relations. Even the Fibonacci function will be encoded using a binary relation. Thus, our Abella development begins as follows.

```
%% FibExample.thm
Kind nat type.
Type z nat.
Type s nat -> nat.
% nat X ≡ X is a natural number
Define nat : nat -> prop by nat z ; nat (s X) := nat X.
% le X Y ≡ X ≤ Y
Define le : nat -> nat -> prop by le z X ; le (s X) (s Y) := le X Y.
% lt X Y ≡ X < Y
Define lt : nat -> nat -> prop by ···
% plus X Y Z ≡ Z = X + Y
```

[3] https://doi.org/10.5281/zenodo.11163505.

```
Define plus : nat -> nat -> nat -> prop by ···
% times X Y Z ≡ Z = X × Y
Define times : nat -> nat -> nat -> prop by ···
```

The nth Fibonacci number is defined in Abella relationally as well:

```
Define fib : nat -> nat -> prop by
; fib z z ; fib (s z) (s z)
; fib (s (s X)) N := exists L M, fib X L /\ fib (s X) M /\ plus L M N.
```

7.2 Using λProlog to compute ground instances

While Abella has a logic programming engine, which implements a fragment of λProlog, as part of its `search` tactic, it is inefficient and cumbersome to use. We could improve this implementation in Abella, but we could also use a trusted external λProlog engine such as Teyjus [41] or ELPI [16]. In λProlog, we can define the `nat` type and the predicates `plus` and `fib` analogously to the Abella formulation above.

```
%% fib.sig: type, term, and predicate constants
sig fib.
  kind nat type. type z nat. type s nat -> nat.
  type plus nat -> nat -> nat -> o.
  type fib  nat -> nat -> o.
```

```
%% fib.mod: program clauses for predicates
module fib.
  plus z X X.
  plus (s X) Y (s Z) :- plus X Y Z.
  fib z z. fib (s z) (s z).
  fib (s (s X)) N :- fib X L, fib (s X) M, plus L M N.
```

With this definition, we can ask a λProlog engine to solve `fib` goals where the first argument is ground. We can also, of course, check that a given ground predicate is indeed derivable. We have instrumented a variant of the Teyjus implementation of λProlog to produce a Dispatch assertion (i.e., in the input language of Dispatch) for such checks. For example, the above check will be written as the following JSON object.

```
{ "format": "assertion", "agent": "exampleAgent",
  "claim": {
    "format": "annotated-production",
    "annotation": {"name": "fib5"},
    "production": {
      "mode": "damf:bafyreic3...",
      "sequent": { "conclusion": "fib5", "dependencies": [] } } },
  "formulas": {
    "fib5": {
      "language": "damf:bafyreibv...",
      "content": "fib (s (s (s (s (s z))))) ...",
12        "context": ["fib"] } },
13    "contexts": {
14      "fib": {
15        "language": "damf:bafyreibv...",
16        "content": [
17           – contents of fib.sig as a string – ,
18           – contents of fib.mod as a string –
19        ] } } }
```

The `"language"` values in lines 10 and 15 are understood to be canonical references to a DAMF object referencing the language of λProlog. Similarly, the `"mode"` value in line 6 is a canonical reference to a DAMF object describing the Teyjus implementation. The `"agent"` value in line 1 is the name of some agent profile created by running `dispatch create-agent`; Dispatch uses the private key of this agent profile to sign the assertion when publishing it to DAMF. This JSON object claims that a specific version of the Teyjus implementation of λProlog has computed the fifth Fibonacci number to be 5.

7.3 Proving arithmetic facts in Coq

The lemma we are ultimately interested in depends on fairly significant arithmetic reasoning. We will use Coq's linear integer arithmetic solver `lia` to write fairly straightforward proofs of the lemma. However, in Coq we will define `fib` not as a binary relation but as a recursively defined fixed point with one argument. The Coq v. 8.16.1 development is shown below, with proofs elided.

```
Fixpoint fib (n : nat) :=
  match n with 0 => 0 | S j =>
    match j with 0 => 1 | S k => fib j + fib k end end.
Theorem fib_square_lemma : forall n, 2 * n + 27 <= fib (n + 12).  ···
Theorem fib_square_above : forall n, 13 <= n -> n ^ 2 < fib n.  ···
```

These proofs are built using the linear integer arithmetic solver `lia` in the proofs that is distributed with Coq.

7.4 Adapting λProlog and Coq sequents for Abella

Taking stock, we have ground facts built in the higher-order logic programming language λProlog using the tool Teyjus, and a lemma about the rate of growth of the Fibonacci function written in the calculus of inductive constructions using the tool Coq. Obviously, neither of these languages correspond to the language $\mathcal{G}$ that forms the basis of the Abella theorem prover. Thus, we need adapters for translating these external dependencies to Abella's language.

These adapters can, in principle, be quite sophisticated; for instance, they can be written using Dedukti. For illustration purposes in the present paper, we adapt the sequents by hand by asserting in Abella the intended translation at the point of importing the assertion. Imagine, for instance, that the `fib5` assertion shown in Section 7.2 is given the `cid`: `bafyreifu`.... Here is how we would import it in Abella:

```
18 %% FibExample.thm continuing...
19 Import "damf:bafyreifu..." as
20 Theorem fib5: fib (s (s (s (s (s z))))) (s (s (s (s (s z))))).
```

From the perspective of Abella, this looks just like an ordinary `Theorem` statement, except there is no proof that follows. Instead, Abella would generate the following adapter sequent (which it could then publish using Dispatch):

```
 1 { "format": "assertion", "agent": "exampleAgent",
 2   "claim": {
 3      "format": "annotated-production",
 4      "annotation": {"name": "fib5"},
 5      "production": { "mode": null,
 6        "sequent": {
 7          "conclusion": "fib5",
 8          "dependencies": [
 9            "damf:bafyreifu.../claim/sequent/conclusion" ] } } },
10   "formulas": {
11     "fib5": {
12       "language": "damf:bafyreiga...",
13       "content": "fib (s (s (s (s (s z))))) ...",
14       "context": ["fib5!context"] } },
15   "contexts": {
16     "fib5!context": {
17       "language": "damf:bafyreiga...",
18       "content": [
19         "Kind nat type.", "Type z nat.", "Type s nat -> nat.",
20         "Define fib : nat -> nat -> prop by ...." ] } } }
```

Lines 12 and 17 above are references to a DAMF object describing the Abella language. In line 5, the `"mode"` field is left as `null` to indicate that this assertion was not created by any tool; in other words, the agent `"exampleAgent"` is solely responsible for the assertion. If a tool had been used instead, this field would refer to the DAMF description of that tool. Finally, in line 9, the dependency that is included is the `cid` of the *conclusion* of the assertion object that was produced by λProlog, and in turn imported by Abella in line 19 of `FibExample.thm`. The dependencies in a sequent are *formula objects*, not assertions; the same formula object can have several different proofs of it asserted by a variety of agents, and the use of the formula as a lemma should not be seen as privileging any particular assertion above others. From Abella's perspective, then, the name `fib5` denotes just the formula on line 20 of `FibExample.thm`.

The Coq lemma `fib_square_above` is imported into Abella in a similar fashion. The only difference is that the Abella translation of the Coq theorem needs to be sensitive to the fact that the type `nat` of Abella is not inductively defined as in Coq, and arithmetic operations are defined relationally. A conservative treatment is as follows:

```
21 %% FibExample.thm continuing...
22 Import "damf:bafyreibr..." as
23 Theorem fib_square_above : forall n, nat n -> le (s^13 z) n ->
```

```
24  forall u, times n n u -> forall v, fib n v -> lt u v.
```

The `cid` `damf:bafyreibr...` on line 22 is that of the assertion corresponding to the theorem `fib_square_above` in Coq. The imported assertion is rewritten as shown in lines 23–24. As before with λProlog, the assertion corresponding to this translation, with the `"mode"` of production set to `null`, can be easily generated and published by Abella.

Given these external lemmas from λProlog and Coq, the final desired theorem is fairly straightforward to assemble. The interested reader can find the full details in the online walk-through [3].

8 A DAMF-aware prover: Costs and Benefits

We list three potential costs of adopting DAMF within the theorem-proving community and comment on why they are relatively inexpensive.

1. *New standards and processes must be adopted.* Any exercise in communicating between different software systems must rely on some standards that define the structure of the data communicated. Here, we have focused on keeping those standards close to a minimum, keeping close to generic requirements for a range of theorem provers. The actual communicated artifacts use a standardized structure (here, JSON).
2. *New software is needed.* Adding a new feature to a theorem prover requires new code. Our implementation of DAMF factors that new software into two parts: (1) the Dispatch tool that is written in easily deployable JavaScript using off-the-shelf technology and (2) new code located in the printing and parsing subsystem of individual provers. As illustrated in Section 7, the logical kernel of provers does not need to be touched to employ DAMF.
3. *Agents must manage public and private keys and allow lists.* While this is a new feature for the theorem-proving community, it is standard technology used by online service providers, ranging from banking to cloud computing.

Next, we highlight the possible benefits a framework such as DAMF could have for the general proof-checking community and also for an individual prover.

8.1 Potential *benefits* for the community

One benefit of using a framework that explicitly addresses trust within the proof-checking community is that it helps to document the possible roles of special-purpose theorem provers within general-purpose proof-checkers. Most proof assistants, such as Coq and Isabelle, are general-purpose in the sense that they can be used in a wide array of areas in mathematics and computer verification. Many other formal method tools, such as SAT solvers, work in very restricted domains: as a result, proof checkers in those more narrow domains can be significantly optimized for modern operating systems and computer hardware to improve their usage of time and memory. If we are committed to being autarkic, the proof certificate generated by UNSAT (using, for example, the DRUP format [22]) would need to be rechecked by the one proof checker we trust. While such rechecking can be done in practice [13], a general purpose kernel is not likely to be able to recheck successfully the large proof certificates that have been generated recently for open mathematical problems. For example, DRUP-based proof certificates that the Erdös discrepancy conjecture is true when its parameter is set to 2 has been reported as being 1.88 gigabytes in size [26], while a DRUP-based proof certificate answering the Schur Number Five problem is much larger, weighing in at 2 petabytes [23]. If these results were to be needed in conventional and general-purpose proof assistants, it is unlikely that their kernel rechecks them: the only other options would be to *explicitly trust* another, specialized proof checker or to find another, presumably smaller proof. Using DAMF, of course, we would not require this rechecking to be done; instead, that framework would track the explicit trusting of such a specialized prover.

As we have described in Section 6.3, adapters are used to translate theorems and their context in one language to theorems and contexts in another language. In essence, adapters are a pairing of a parser for one language and a printer for another language. Given that the correctness of both printing and parsing is a significant problem in the setting of theorem provers (see, for example, [25,38]), our use of adapters explicitly elevates this pairing to be a named tool that one can choose to trust or not. The considerable work that has gone into representing logics and proof systems in generic ways, such as Dedukti [6] and MMT [42], can be used to construct high-quality and broadly applicable adapters.

In contrast, many modern theorem provers come with a central repository containing a structured library of theories: see, for example, libraries associated to Lean [43] and Coq [31]. Given that the underlying file structure of DAMF is based on IPFS, there is no *a priori* hierarchy imposed on the structure of theories (i.e., collections of definitions, lemmas, and theorems). As a result, the theorems that appear in the DAMF *global store* or *information layer* can be curated into any number of collections to suit different needs. For example, a textbook in one area of mathematics might organize its collection of theorems differently than those used by software developers attempting to prove correct some safety-critical computer components.

A final benefit for the proof checking community is the "let a thousand flowers bloom" principle: efforts to prove theorems in many different domains with many different methods should be encouraged. If these efforts yield results that can be trusted, their results should be available directly to any other prover willing to, at least, tentatively trust them. Specialized proof languages for specialized settings could then be incorporated into the community activity of establishing proofs: examples of such systems are the geometrically-based prover GeoGebra [18] and a graphic presentation of commuting diagrams [27].

8.2 Potential *benefits* for an individual theorem prover

While the structure of DAMF has been motivated around issues involved with a community of different theorem provers, there are several reasons that an individual theorem prover might adopt elements of DAMF for internal reasons.

Version control tracking yields an immediate and natural use of DAMF. If a theorem has been checked by version n of a prover, can we trust that theorem in version $n + 1$ or insist that it be rechecked by version $n + 1$? Certain features of the newer proof checker may have stayed the same, and, as a result, one might be willing to accept certain theorems checked by the earlier version of the prover.

Since deploying a framework like DAMF removes the emphasis on rechecking proofs, it removes the need for kernel implementations to chase performance by means of complicated techniques whose correctness conditions may be unclear. As a result of avoiding such optimizations, the kernel can be simpler and, hence, more accessible to inspect, maintain, and trust.

Adapters can provide features and logical expressiveness without incorporating them into the kernel's logical core. For example, the logic behind both λProlog and Abella is based on Church's Simple Theory of Types [12]. While this logic allows for quantification at all simple types, it does not allow for *type variables* and type instantiation, i.e., polymorphism. Some form of polymorphism is however immensely useful and it is therefore demanded by users of these two systems. One approach to providing polymorphism can be to extend the foundations of Church's logic to include type variables (for example, by moving to a much more expressive logic like System F [19]). Enriching the logic in this way can make the kernels of these systems much more involved (especially since the kernels of both λProlog and Abella involve unification of λ-terms). Another approach that does not involve any changes to the kernel would be to write an adapter that takes as input definitions and theorems that mention a generic type constant, say `a`, and then outputs versions of those definitions and theorems in which `a` is instantiated with different types. For example, the Abella file in Figure 2 defines the append relation on lists of simple type `list a`. As far as Abella's kernel is concerned, the type `a` is basic type, and this file checks in the (unmodified) kernel. A rather simple adapter can be written that specializes the type `a` to arbitrary types, such as `nat` or `bool`, and then create specialized versions of the `append` predicate and associated theorems. The adapter must of course be trusted to perform this specialization correctly, but the kernel does not need to be touched.

Finally, web browsers are emerging as possible interfaces for theorem provers; for example, a version of Abella is available as a client-side JavaScript program [2]. Because web browsers in a browser context usually run in a sandbox, there is the problem of having a persistent store (such as a file system). IPFS and DAMF readily solve this problem by persisting developments to the global cloud. Note that because of its content-addressed nature, unless some other node in the network accesses these files by their `cid`s, they will not propagate; thus, private developments will remain private as long as the `cid`s are not published.

9 Conclusion

We have focused on describing a framework that explicitly tracks the trust that one must have when using multiple proof checkers. The Distributed Assertion

```
Kind  list   type -> type.
Kind  a      type.
Type  empty  list a.
Type  cons   a -> list a -> list a.

Define append : list a -> list a -> list a -> prop by
  append empty L L ;
  append (cons X L) K (cons X M) := append L K M.

Theorem append_associative : forall A B C AB ABC,
   append A B AB -> append AB C ABC ->
     exists BC, append B C BC /\ append A BC ABC.
induction on 1. intros. case H1.
  search.
  case H2. apply IH to H3 H4. search.
```

Fig. 2. An example of accommodating type instantiation.

Management Framework (DAMF) uses the InterPlanetary FileSystem to distribute proof-checking assertions using public key cryptography. DAMF provides for declaring and managing the kinds of dependencies that occur within theorem provers without an explicit reference to a centralized library structure. This approach can be implemented in provers with minimal modifications using the Dispatch tool. We have illustrated how the Abella theorem prover benefits from including DAMF features, and we have argued that other provers might expect similar benefits. Lifting the DAMF framework to handle other domains, such as journalism and experimental science, will require the treatment of new dimensions of trust and mistrust, such as observational data and computations based on such data. Whether or not such a lifting is possible is still open.

References

1. Abadi, M.: Variations in access control logic. In: van der Meyden, R., van der Torre, L.W.N. (eds.) Deontic Logic in Computer Science (DEON 2008). LNCS, vol. 5076, pp. 96–109. Springer (2008). https://doi.org/10.1007/978-3-540-70525-3_9
2. Abella in your browser (2015), https://abella-prover.org/tutorial/try
3. Al Wardani, F., Chaudhuri, K., Miller, D.: The distributed assertions website, May 2024 archived version, https://doi.org/10.5281/zenodo.11163505
4. Al Wardani, F., Chaudhuri, K., Miller, D.: Formal reasoning using distributed assertions. In: Sattler, U., Suda, M. (eds.) FroCoS 2023. LNAI, vol. 14279, pp. 176–194 (2023). https://doi.org/10.1007/978-3-031-43369-6_10
5. Apéry, R.: Irrationalité de $\zeta 2$ et $\zeta 3$. Journées Arithmétiques de Luminy, Astérisque **61**, 11–13 (1979)
6. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory (2016), https://theses.hal.science/INRIA-SACLAY-2015/hal-04281492v1

7. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: A system for reasoning about relational specifications. J. of Formalized Reasoning **7**(2), 1–89 (2014). https://doi.org/10.6092/issn.1972-5787/4650
8. Benet, J.: IPFS-content addressed, versioned, P2P file system (2014). https://doi.org/10.48550/arxiv.1407.3561
9. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. In: Linking the World's Information, pp. 91–103. ACM (2023). https://doi.org/10.1145/3591366.3591376
10. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
11. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979)
12. Church, A.: A formulation of the Simple Theory of Types. J. of Symbolic Logic **5**, 56–68 (1940). https://doi.org/10.2307/2266170
13. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: TACAS 2017: Tools and Algorithms for the Construction and Analysis of Systems. pp. 118–135. Springer (2017). https://doi.org/10.1007/978-3-662-54577-5_7
14. Debian's SecureApt, https://wiki.debian.org/SecureApt
15. Dowek, G., Thiré, F.: Logipedia: a multi-system encyclopedia of formal proofs. Tech. Rep. abs/2305.00064, ArXiV (2023). https://doi.org/10.48550/ARXIV.2305.00064
16. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable, λProlog interpreter. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-20. LNCS, vol. 9450, pp. 460–468. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_32
17. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Foundations and Trends in Theoretical Computer Science **9**(3-4), 211–407 (2014). https://doi.org/10.1561/0400000042
18. Geogebra for teaching and learning math. https://www.geogebra.org/
19. Girard, J.Y.: The system F of variable types: Fifteen years later. Theoretical Computer Science **45**, 159–192 (1986). https://doi.org/10.1016/0304-3975(86)90044-7
20. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979). https://doi.org/10.1007/3-540-09724-4
21. Gordon, M.: HOL: A machine oriented formulation of higher-order logic. Tech. Rep. 68, University of Cambridge (Jul 1985), https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-68.pdf
22. Heule, M., Jr., W.A.H., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2013. pp. 181–188. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679408
23. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second Conference on Artificial Intelligence (AAAI-18). pp. 6598–6606. AAAI Press (2018). https://doi.org/10.1609/AAAI.V32I1.12209
24. Jones, C.B.: VDM proof obligations and their justification. In: Bjørner, D., Jones, C.B., Mac an Airchinnigh, M., Neuhold, E.J. (eds.) VDM 1987 – A Formal Method at Work. LNCS, vol. 252, pp. 260–286. Springer-Verlag (1987). https://doi.org/10.1007/3-540-17654-3_15

25. Jourdan, J.H., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: Seidl, H. (ed.) ESOP: Programming Languages and Systems. pp. 397–416. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
26. Konev, B., Lisitsa, A.: Computer-aided proof of Erdös discrepancy properties. Artificial Intelligence **224**, 103–118 (2015). https://doi.org/10.1016/j.artint.2015.03.004
27. Lafont, A.: A diagram editor to mechanise categorical proofs. In: JFLA 2024: Journées Francophones des Langages Applicatifs. Saint-Jacut-de-la-Mer, France (Jan 2024), https://hal.science/hal-04407118
28. The Lean Reference Manual, https://leanprover.github.io/reference/
29. Mahboubi, A., Sibut-Pinote, T.: A formal proof of the irrationality of $\zeta(3)$. Logical Methods in Computer Science **17**(1), 1–25 (2021). https://doi.org/10.23638/LMCS-17(1:16)2021
30. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press (Jun 2012). https://doi.org/10.1017/CBO9781139021326
31. Müller, D., Rabe, F., Coen, C.S.: The Coq library as a theory graph. In: Intelligent Computer Mathematics: 12th International Conference, CICM 2019, Prague, Czech Republic, July 8–12, 2019, Proceedings 12. pp. 171–186. Springer (2019). https://doi.org/10.1007/978-3-030-23250-4_12
32. Necula, G.C.: Proof-carrying code. In: 24th Symposium on Principles of Programming Languages 97. pp. 106–119. ACM, Paris, France (1997). https://doi.org/10.1145/263699.263712
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. No. 2283 in LNCS, Springer (2002). https://doi.org/10.1007/3-540-45949-9
34. Nystrom et al: UEFI networking and pre-OS security. Intel Technology Journal - UEFI Today: Boostrapping the Continuum **15**(1), 80–101 (Oct 2011)
35. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th Conf. on Automated Deduction (CADE). LNAI, vol. 607, pp. 748–752. Springer (Jun 1992). https://doi.org/10.1007/3-540-55602-8_217
36. Paul, C., Matthews, M.: The Russian "Firehose of Falsehood" Propaganda Model. Rand Corporation **2**(7), 1–10 (2016), https://www.rand.org/pubs/perspectives/PE198.html
37. Paulson, L.C.: Isabelle: A Generic Theorem Prover. No. 828 in LNCS, Springer (1994). https://doi.org/10.1007/BFb0030541
38. Pollack, R.: How to believe a machine-checked proof. In: Sambin, G., Smith, J. (eds.) Twenty Five Years of Constructive Type Theory. Oxford Uni. Press (1998)
39. van der Poorten, A.: A proof that Euler missed .... In: Berggren, L., Borwein, J., Borwein, P. (eds.) Pi: A Source Book, pp. 439–447. Springer New York (2000). https://doi.org/10.1007/978-1-4757-3240-5_49
40. Portoraro, F.: Automated Reasoning. In: Zalta, E.N., Nodelman, U. (eds.) The Stanford Encyclopedia of Philosophy. Spring 2024 edn. (2024), https://plato.stanford.edu/archives/spr2024/entries/reasoning-automated/
41. Qi, X., Gacek, A., Holte, S., Nadathur, G., Snow, Z.: The Teyjus system – version 2 (2015), https://teyjus.cs.umn.edu/
42. Rabe, F.: The future of logic: Foundation-independence. Logica Universalis **10**(1), 1–20 (2016). https://doi.org/10.1007/s11787-015-0132-x
43. The mathlib Community: The Lean mathematical library. In: CPP 2020: Intern. Conference on Certified Programs and Proofs. pp. 367–381. ACM (Jan 2020). https://doi.org/10.1145/3372885.3373824