Distributing and trusting proof checking: a preliminary report

Kaustuv Chaudhuri, Dale Miller, Farah Al Wardani

Distributing and trusting proof checking: a preliminary report

Kaustuv Chaudhuri
Inria Saclay, LIX, IPP
Palaiseau, France

Dale Miller
Inria Saclay, LIX, IPP
Palaiseau, France

Farah Al Wardani
Inria Saclay, LIX, IPP
Palaiseau, France

Abstract

When a proof-checking kernel completes the checking of a formal proof, that kernel asserts that a specific formula follows from a collection of lemmas within a given logic. We describe a framework in which such an assertion can be made globally so that any other proof assistant willing to trust that kernel can use that assertion without rechecking (or even understanding) the formal proof associated with that assertion. In this framework, we propose to move beyond autarkic proof checkers—i.e., self-sufficient provers that trust proofs only when they are checked by their kernel—to an explicitly non-autarkic setting. This framework must, of course, explicitly track which agents (proof checkers and their operators) are being trusted when a trusting proof checker makes its assertions. We describe how we have integrated this framework into a particular theorem prover while making minor changes to how the prover inputs and outputs text files. This framework has been implemented using off-the-shelf web-based technologies, such as JSON, IPFS, IPLD, and public key cryptography.

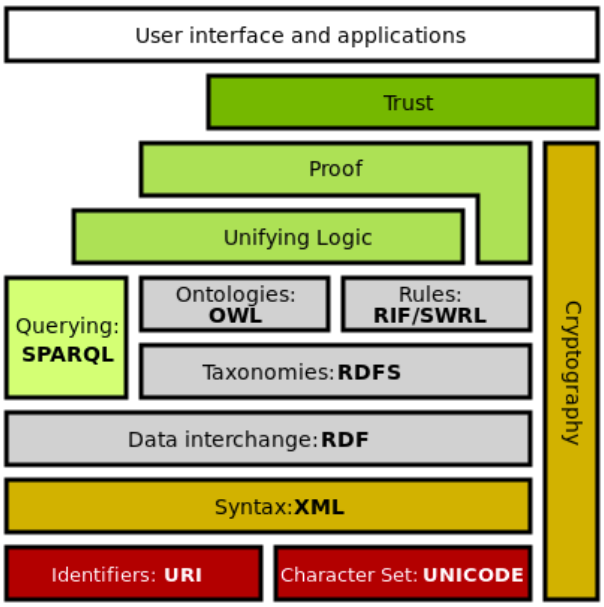

Figure 1. The semantic web stack [Source].

1 Introduction

A triumph of the World Wide Web is the ease at which anyone can access a great deal of diverse information. A glaring flaw of the web is that it provides few tools to help consumers of information trust the assertions that may be made in documents retrieved from the web. While techniques such as digital signatures can be used to determine the author of signed information and blockchains can be used to capture the provenance and timing of some sources of information, few techniques are available to provide trust in what is claimed in information sources.

One approach to providing the web with some aspects of trust was the proposal for the *semantic web* [BL98, BLHL01]. In that proposal, trust would be achieved at the top of a stack of various protocols and specification languages (see Figure 1). While the semantic web addresses a different set of problems than what we address here, some similarities are worth pointing out. In both cases, cryptography and low-level web-based technologies form the foundation on which most other elements are built. Near the top, a universally accepted logic and a notion of formal proof support trust. It is mainly the middle layers where the semantic web and our framework disagree since the domains of computational logic and formalized mathematics have replaced concepts such as taxonomies, ontologies, rules, and queries with other mathematical notions. At the top of this stack are the application programs that can be built on trust: in our case, these programs are proof checkers and tools associated with them.

There are, of course, significant challenges to establishing truth and trust in a distributed publishing platform like the World Wide Web. We now discuss our plans to find a means to maintain trust in proof checking when we move it into such a distributed setting.

When speaking of distribution, we mean in *space* and *time*. That is, we might have many active users spread across the globe, all working on producing and consuming proofs of theorems. Furthermore, we wish that items produced (and checked) today can be used as they are in years to come.

2 Two styles of proof checking

2.1 Autarkic proof checking

An *autarkic* proof checker[1] is simply a proof checker that only trusts its kernel. If proof checker A needs help from a second proof checker B, there are roughly two ways to

[1] In [BB02], this adjective was applied to computational components of a proof checker but not to a full proof checker.

proceed in the autarkic setting [AFG+11, FMM+06]. Checker B might output a proof witness that checker A can use to reconstruct a proof for its kernel: in this case, checker B is a *certifying* proof checker for A. Alternatively, the correctness of the checker B might be provable in checker A. Thus, if checker B succeeds in proving a theorem, then checker A immediately accepts that theorem. In this case, checker B is a *certified* proof checker by A. In either of these cases, the autarkic proof checker only trusts its own kernel.

The Dedukti proof system [ABC+16] has added a new dimension to autarkic provers. The logical framework behind Dedukti is a dependently typed λ-calculus augmented with rewriting. Since this logical framework is relatively simple, proof checkers for it can be compact, increasing one's willingness to trust it. At the same time, Dedukti is expressive and has been shown to capture proofs in many proof checkers. Given that a client proof system of Dedukti can output its proofs in a format Dedukti can check, that client prover can feel more confident that some other proof checkers also certify its proofs. While that is a non-autarkic conclusion, it seems that Dedukti is taking a central stage in the world of autarkic proof checkers. In particular, "[s]ome proofs expressed in some Dedukti theories can be translated to other proof systems, such as HOL Light, HOL 4, Isabelle/HOL, Coq, Matita, Lean, PVS, . . ." [Log22]. Thus, if Coq needs a theorem that was proved in, say, Lean, it might be possible for Lean to output a proof in the Dedukti format and then have that proof output into a format suitable for Coq to check. Thus, while Dedukti can be at the hub of exchanges of proofs between different proof checking systems, Dedukti does not need to be trusted by these proof systems.

We identify the two problems with this class of provers.

1. Such provers are hard to use in distributed proof development. Consider, for example, the HOL Light proof checker [Har17], where all theorems and their proofs are rechecked every time the system is initiated. Using checkpoints to save proof-checking sessions once a collection of proofs has been checked makes such systems more manageable for a single user. However, such techniques do not help in a distributed setting since remote users cannot share checkpoints and must, instead, recheck entire theories.
2. There are forces that cause kernels to grow in size and complexity over time. For example, since libraries of theorems and proofs tend to grow in size and complexity, kernels need to be optimized to be more efficient in using time and memory. Of course, such optimization can complicate the kernel, making it harder to check it for correctness. In addition, proofs can cover many different computational paradigms. For example, (functional programming-style) computation is often part of checking a proof. Proving theorems about reachability and simulation can require logic programming and model checking-style search. Requiring a kernel to be effective in all of these aspects of proof checking complicates the design of a kernel.

2.2 Non-autarkic proof checking

A typical place to find a non-autarkic proof checker is in theorem provers that use a range of special purpose provers, such as static analyzers, model checkers, SAT solvers, and SMT provers. Such special-purpose systems may not be certified or certifying, but they may still be important to use in certain settings. In 2002, Shankar [Sha02] argued that "little engines of proof" would need to be aggregated to yield the next generations of proof systems. The "evidential tool bus" [Rus05, CHOS13] was proposed as a means to integrate such specialized inference systems into a unified formal method system.

Besides the fact that it can be useful to employ many different theorem proving tools, two obvious questions lead us to consider non-autarkic proof checkers.

1. A proof of a theorem needs to be checked only once by a trustworthy agent. If we trust that prover, why recheck it ever again?
2. How can we develop a framework in which we only need to recheck proofs when we reconsider the agents we trust?

Of course, we must directly deal with *trust* to respond to these questions. In the next section, we present the design of a framework that does that explicitly.

3 An abstract design

3.1 The key elements of the design

The key components of our approach to understanding distributing and trusting proof checking involve the following items, which we list briefly and then expand on below.

- We choose one logic (say, higher-order logic) and parameterize it with logical signatures used to type-check formulas for syntactic correctness.
- We define a *justified sequent* as a pairing of a *justification* with a sequent that connects a list of hypotheses to a conclusion.
- We formally introduce agents who can assert the correctness of a justified sequent.
- We introduce a simple first-order logic, called the says-logic, which can deduce trusting relations for an agent.

3.2 Declaring parameters and namings

We assume that all provers will use the same logic. While the exact nature of this logic is not an issue for this paper, we pick the intuitionistic logic variant of Church's *Simple Theory of Types* [Chu40] (excluding the mathematical axioms of choice, infinity, and extensionality). This choice of logic captures propositional logic, multi-sorted first-order logic,

and higher-order logic. With the addition of suitable axioms, it also easily accommodates classical logic.

A key feature of such a simply typed logical language is the notion of *logical signature*.[2] To be concrete, such signatures contain two kinds of declarations. The specification of *parametric symbols* are made with *kind* and *type declaration*.

Kind declarations introduce primitive types and type constructors. In particular, we write τ/n to introduce a type constructor of arity n. If n is 0, τ is a primitive type. Simple types are built using the arrow type constructor in addition to these primitive types and type constructors.

Type declarations, such as $c\colon \tau$, introduces the constant c with simple type τ.

The second kind of declaration allowed in signatures is used to *name symbols*, of which there are two kinds.

Definition naming, written as $n \overset{\sigma}{:=} t$, declares that n (the *definiendum*) names the term (or formula) t (the *definiens*), which needs to have type σ. The symbol n is treated as a symbol of type σ from the point-of-view of a parser and type checker. From the point-of-view of the theorem prover, it should always be possible to replace the definiendum with its definiens. We do not accept recursive definitions at the level of declarations: rather, a definition can name a recursive definition, one defined by, say, a least or greatest fixed point expression.

Theorem naming declarations attribute a name to a formula that is expected to be eventually justified as a theorem. A formula may have multiple names but one name can denote at most one formula. The names for theorems might be used in proof scripts and proof certificates (collectively called *justifications* below).

As is usually done with logical signatures, we shall assume that (1) kinds and types belong to two different namespaces (e.g., it is possible for the token `nat` to be a kind of zero arity and a constant of type $\texttt{nat} \to o$), and (2) declarations in the same namespace must be functional (i.e., a name cannot be given two different kind arities, cannot be given two different types, and cannot be give two different definitions).

Note that the underlying structure of a logical signature is as a *set* of declaration: since we are not working with dependently typed λ-terms, we do not need these structures to be lists.

We shall assume that there is one type reserved for designating formulas. Church used omicron o, Coq uses `Prop`, and other systems reserve `bool`. In any case, we shall assume that the propositional connectives and quantifiers are given suitable typings over such types. For example, in [Chu40], we find typings such as $\wedge : o \to o \to o$ and $\forall_\sigma : (\sigma \to o) \to o$.

[2]Not to be confused with *cryptographic signatures*, which we encounter later.

Note that while we are fixing one universal logic, we are not fixing any particular format for proofs in that logic.

3.3 Justifications and assertions

Let Σ be a given logical signature. A Σ-*sequent* is a triple

$$\Sigma\colon \Gamma \vdash N,$$

where Γ is a list of theorem names and N is a theorem name: all these named theorems must be declared in Σ and they name Σ-formulas. A Σ-*justified sequent* is a quadruple

$$J :: \Sigma\colon \Gamma \vdash N,$$

where $\Sigma\colon \Gamma \vdash N$ is a Σ-sequent and J is a *justification*, which is meant to indicate some proof object or proof script. The justification J may or may not contain terms and formulas, but if it does, we also assume that they are Σ-terms and Σ-formulas, respectively. In general, we do not need to know about the structure of justifications: instead, we assume that that structure is known to specific proof checkers. In general, proof structures play the sole role of convincing a proof checker that a sequent is a genuine entailment.

We use the schematic variable K to range over agents. For now, we can think of agents as just names, but later we will allow them to be public keys within a public key infrastructure (PKI). Roughly speaking, an agent will be identified with a user employing a proof checker. We also introduce a simple multi-sorted first-order logic with just one predicate, namely, K *says* S, where S is a Σ-sequent. We shall refer to this logic (which is not the logic of the formulas for which we are checking theoremhood) as the *says*-logic.

Assume that agent K uses a proof checker to check the justified sequent

$$J :: \Sigma\colon N_1, \ldots, N_n \vdash N,$$

and that she concludes that the formula named by N is a theorem if the formulas named by $N_1, \ldots, N_n$ are theorems. (Presumably, the justification J uses these formulas as lemmas.) An *assertion* of this fact will be written as

$$K \textit{ says } (\Sigma\colon N_1, \ldots, N_n \vdash N_0).$$

Implicitly, this statement also asserts that all the formulas named by $N_0, \ldots, N_n$ are Σ-formulas. In this case, we say that $K \textit{ says } (N_1, \ldots, N_n \vdash N_0)$ is a Σ-formula in the says logic (we often drop explicit reference to Σ when it can be determined from context).

Eventually, assertions of the form K *says* S will be implemented using public-key cryptography. As a result, we shall also refer to assertions as being signed judgments.

3.4 The logic of agent assertions

Since inferences within the says-logic will only involve Horn clause assumptions, we can describe its provability logic as either classical or intuitionistic. We assume the following

says-logic formula, called Lemma, is our only non-logical axiom.

$$K\ says\ (\Gamma_1 \vdash N) \wedge K\ says\ (N, \Gamma_2 \vdash M) \supset K\ says\ (\Gamma_1, \Gamma_2 \vdash M),$$

(Here, M, N, and the members of Γ_1 and Γ_2 are all names of formulas.) A simple consequence of Lemma is that if agent K is willing to assert the hypothetical $N \vdash M$, then that agent must accept the provability of M whenever it accepts the provability of N.

Other logics that consider access control or authorization among agents often include other axioms, such as

$$\forall A \forall K.\ A \supset K says \vdash A$$
$$\forall A \forall K.\ K says \vdash A \supset K says \vdash (K says \vdash A)$$

Not only do we not want these entailments to hold here, they are, formally speaking, syntactically ill-formed.

If K and K' are two agents, then we write $[K \mapsto K']$ to denote the formula

$$\forall x \forall \Gamma.\ K\ says\ (\Gamma \vdash x) \supset K'\ says\ (\Gamma \vdash x).$$

Note that from the assumptions $[K_1 \mapsto K_2]$ and $[K_2 \mapsto K_3]$, the formula $[K_1 \mapsto K_3]$ follows.

Let $\mathcal{K}$ be a finite list of agent names. We write $[\mathcal{K} \mapsto K]$ to denote the set of says-formula

$$\{[K' \mapsto K] \mid K' \in \mathcal{K}\}.$$

If we think of $\mathcal{K}$ as the set of agents that K is willing to trust (i.e., K's *allow-list*), then the set $[\mathcal{K} \mapsto K]$ contains all the formula needed to transfer an assertion from an allow-list agent to an assertion of agent K.

If J is the formula $K\ says\ (N_1, \ldots, N_n \vdash N_0)$, let $\ulcorner J \urcorner$ be

$$K says \vdash N_1 \supset \cdots \supset K says \vdash N_n \supset K says \vdash N_0.$$

It is easy to see that J is stronger than $\ulcorner J \urcorner$ in the sense that from J and the axiom Lemma, $\ulcorner J \urcorner$ follows while the converse is not true. To illustrate why the converse entailment does not hold, let J be $K_1\, says (N_1 \vdash N_0)$. From J and the additional assumptions $[K_1 \mapsto K_2]$ and $K_2 says \vdash N_1$ it follows that $K_2 says \vdash N_0$. The same conclusion is not possible when J in this syllogism is replaced with $\ulcorner J \urcorner$.

For a discussion about similar logics involving the $\cdot\ says\ \cdot$ predicate, see [Aba08].

4 Comments on this design

4.1 Replacing proofs with assertions

The heart of the design described above is the step where a justified sequent of the form

$$J :: \Sigma\colon N_1, \ldots, N_n \vdash N_0$$

is replaced with an assertion of the form

$$K\ says\ (\Sigma\colon N_1, \ldots, N_n \vdash N_0).$$

The justified sequent is a claim that some proof structure J convinces a proof checker used by K that the formula named by N_0 follows from the named formulas $N_1, \ldots, N_n$. Using a proof checker can be resource intensive. For example, the structure J might be a large structure containing many low-level details. On the other hand, J might have fewer explicit details, which could leave the reconstruction of missing details (for example, unifiers) to be constructed by the checker: such reconstruction could involve a lot of computation time. If one is willing to trust agent K, then one does not need to recheck the entailment encoded by the sequent $N_1, \ldots, N_n \vdash N_0$.

As a result of our prominent use of assertions (in the sense above), we are not providing formal proofs any role in our framework beyond convincing a proof-checking kernel to make an assertion in the first place. Formal proofs themselves do not need to be communicated, transformed, and checked by another proof system. Our approach stands in contrast to projects such as Dedukti [ABC+16] and Logipedia [DT19] in which communicating formal proofs plays a central role. The framework we are proposing here can be extended with additional notions of assertions in which formal proofs are a part: their role can be used to limit the number and kind of agents that are trusted.

4.2 Users, proof checkers, and agents

We use the term agent to mean the combination of a user and a proof checker. An agent is the entity that signs a sequent. In particular, we do not speak of a proof checker making an assertion because a vector of attack against proof checking is a malicious user who might compromise the code of a proof checker. Such attacks are a recognized threat with, for example, the Coq system where a compiled `.vo` file can be maliciously modified by the user of the Coq system [ANS21].

In general, users can be identified with humans, but it also makes sense to identify them also with some specific *container instance* (e.g., Docker) that invokes a proof checker as part of some continuous integration activity. Furthermore, a given person might make use of multiple proof checkers: in that case, we could say that that person has several *profiles* and each of these profiles is associated with a different agent (or public-private key).

4.3 Limited closure properties are assumed for agents

As we have seen in Section 3.4, an agent that is willing to make a hypothetical assertion must also satisfy the closure property expressed by Lemma. We do not assume that agents have any additional closure properties. For example, it does not follow that the formula

$$K\ says\ N \wedge K\ says\ NM \supset K\ says\ M$$

holds, where NM names the formula $A \supset B$ (in some extension to the ambient logical signature) and where A and B are formulas named by N and M, respectively. 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 the substitution of parameters. For example, we might expect that most proof checkers have a justification for the formula $p \supset p$ where p is some parameter declared of type o. That same proof checker might not be able to prove the result of instantiating p in that formula with a potentially large formula B: that is, some proof checkers might not succeed to check $B \supset B$ if B is, say, a formula with a million occurrences of logical connectives.

While a proof checking agent may not be closed under modus ponens or the instantiation of parameters, it is possible to employ other agents that can look for opportunities to apply such inference rules on the results of trusted agents. Thus, there could easily be an agent K' for which

$$K' \; says \, N \wedge K' \; says \, NM \supset K' \; says \, M$$

is, say, the only kind of entailment it is willing to validate. Hence, if K is in the allow-list of K', then K' can help provide some closure on the assertions made by K.

4.4 Reasoning with many agents, lemmas, and proofs

We have described the result of using a proof checker as generating assertions involving the names of agents and theorems. In the end, however, a user of this system will want to know the answer to a question such as "Is Thm12 proved by an agent I trust?". Given the signed judgments, such questions can be answered using deduction based on building proofs using Horn clauses. Note that in the says-logic, atomic formulas of the form $K \; says \, \Sigma \colon N_1, \ldots, N_n \vdash N_0$ will be either *axioms*, if they are correctly signed by the private key for K, or as *derived*, if they follow from axioms, the allow list, and Lemma.

5 A concrete design

We now present a more concrete design by describing how we have applied this abstract design to the Abella theorem prover [BCG+14]. Abella was originally designed to test a particular approach to meta-theoretic reasoning using some new, proof-theoretically motivated mechanism for reasoning directly with bound variables (in particular, the ∇-quantifier and a treatment of equality based on higher-order unification). While the current implementation of Abella has succeeded with those meta-theoretic tasks [FMP15, Tiu08], the prover has not grown much beyond that domain. As a result, Abella has remained a small prover that lacks automation, a proof library, web interaction, and any sophisticated notion of persistence (e.g., a file system). As a result, Abella could stand to benefit greatly from being part of a distributing and trusting proof checking environment. Furthermore, the area of meta-level reasoning that Abella treats declaratively is also an area where many conventional proof assistants do not deal with well (in part, because of the need to encode and manipulate bindings [ABF+05]). As a result, other provers might be willing to delegate such meta-level reasoning to Abella and simply trust the meta-theorems it is able to prove.

```
Kind nat type.

Type z nat.
Type s nat -> nat.

Define nat : nat -> prop by
  nat z;
  nat (s N) := nat N.

Define plus : nat -> nat -> nat -> prop by
    plus z N N ;
    plus (s M) N (s K) := plus M N K.

Theorem plus_zero :
 forall N, nat N -> plus N z N.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.
```

Figure 2. The Abella theorem file `fileA.thm`.

```
Import "fileA".

Theorem plus_succ :
 forall M N K, plus M N K -> plus M (s N) (s K).
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem plus_comm :
 forall M N K, nat K -> plus M N K -> plus N M K.
induction on 2. intros. case H2.
  apply plus_zero to H1. search.
  case H1. apply IH to H4 H3.
  apply plus_succ to H5. search.
```

Figure 3. The Abella theorem file `fileC.thm`.

5.1 One agent using a local file system

A theorem file (using filename extension `.thm`) of Abella contains the elements needed to build justified sequents, as defined in Section 3.2. Consider, for example, the theorem file `fileA.thm` in Figure 2. This file declares one primitive type `nat`, two constructors (for zero and successor), two defined predicates, and a named theorem. The last three lines of this file contain a proof script that is used by Abella to prove the named theorem. This proof script, which is an example of what we called a justification in Section 3.3, is meant to be meaningful to Abella but the exact nature of such proof scripts will not concern us much in this paper. As illustrated in Figure 3, it is also possible to have a preamble to a theorem file that uses the `Import` keyword to explicitly list the names of other `.thm` files. Note that proof scripts may or may not contain the name of other theorems.

The name of a theorem file is translated into a logical signature by the following steps.

1. By a *local file system (LFS)* we mean the familiar Unix-based mechanism for translating a file name to a (possibly large) string containing the contents of that file. The *current directory* is some prefix that the LFS might need to locate a given filename: this is usually the directory of the shell running the Abella proof checker.
2. A file name mentioned in an `Import` statement is looked up in the LFS at the current directory. If that file is found, the contents of that file are assumed to have a preamble (possibly empty) of `Import`s followed by the `Kind`, `Type`, `Define`, and `Theorem` keywords.
3. The imported files are processed in a recursive fashion. As imported files are parsed, their contents are accumulated into a single logical signature. This accumulation is simply set union.

For example, the logical signature associated with the filename `fileC.thm` is the signature associated with `fileA.thm` plus the two named theorems. Note that the proof scripts in those files are not part of their associated logical signature. We also assume that the dependency graph described by the `Import` keywords is acyclic: a condition that Abella must check explicitly.

When given a filename, the proof checker attempts to check all justifications in that file: the result of that checking phase is a list of sequents representing what those justification have proved. For example, let Σ_0 be the logical signature containing the declarations for `z` and `s` and the two declarations for `nat`. Let Σ_1 be Σ_0 extended with the naming declarations for the definition `plus` and the two theorem declarations for `plus_zero` and `plus_succ`. The result of checking `fileC.thm` (Figure 3) are the following sequents.

$$\Sigma_0 : \vdash \texttt{plus\_succ}$$
$$\Sigma_1 : \texttt{plus\_zero}, \texttt{plus\_succ} \vdash \texttt{plus\_comm}$$

Note that if the `Theorem` keyword is not followed by a justification, then that keyword occurrence does not produce a corresponding sequent. Also, if the `Theorem` keyword is used more that once and each occurrence is followed by a justifications, then a sequent is produced for each such occurrence (the collection of lemmas might be different in these different occurrences).

In the setting of one person using a proof checker on a local file system, the assertions that such sequents are proved is only known to this one user. The next steps we take will make it possible for others to learn and possibly trust those assertions.

5.2 Distribution via content-addressed storage

In a distributed setting, we should like to share objects with other people and programs. These objects could represent structured objects such as formulas, signatures, sequents, assertions, etc. In our discussion above, we have employed a logical signature to dereference a name for the object it denotes. For example, in the example sequents displayed in the previous subsection, the signature Σ_1 is used to dereference the symbolic name `plus_succ` into the formula

```
forall M N K, plus M N K -> plus M (s N) (s K)
```

In a distributed setting, we will need a more global method for dereferencing the name of an object into that object.

A first attempt to dereference names in a global setting might be to use the web protocol `https:`. For example, we might use a url such as

```
https://proof-checker-server.org/library/plus_succ
```

and a number of common tools could then be used to dereference the name `plus_succ` into its associated string. Although such a scheme is frequently used, there are serious problems with it. For example, there is no guarantee that the contents of the associated web page have stayed the same between two downloads. Also, the server cited in the URL might have disappeared, or it might be malicious and deliberately sent the wrong object. (We discuss attacks against distributing proof checking in Section 7.3.)

Content-addressable storage provides a solution to these kinds of problems. In such systems, an object is given a unique name via a *hash* function. To the extent that it is extremely difficult to discover collisions (two different files with the same hash), we can proceed to view such a hash as **the** name associated to that content.

In our implementation of distributing proof checking, we use the IPFS (InterPlanetary File System) systems [Ben14]. For example, by using one of several implementations of IPFS that are available, we can compute the name for a formula as follows.

```
> cat t
forall M N K, plus M N K -> plus M (s N) (s K)
> ipfs add t
added QmdqKdoddW1wkUWB9Qoaem83Sq9gvTejwNxU6VSKMJZdhg t
>
```

The formula mentioned in the first line is given a unique (multi)hash name (the token starting with `Qm`). This name is global: no logical signature or local declarations are needed to dereference that name. In fact, anyone who has installed IPFS on their local machine should be able to access that formula-as-string. For example, the following command

```
> ipfs cat QmdqKdoddW1wkUWB9Qoaem83Sq9gvTejwNxU6VSKMJZdhg
forall M N K, plus M N K -> plus M (s N) (s K)
>
```

should function from the command-line on any internet connected computer with the proper IPFS software installed. By simply computing the hash on the file that is downloaded will reveal whether or not that download is really the file associated to its name. Such checks are automatically done within the IPFS software. (The filename `t` is a temporary file only used to get a string accepted by the `ipfs add` command.)

We follow the IPFS convention of using the term `cid` (short for *content identifier*) to refer to these hash names.

An important aspect of using `cid`s to name theorems is that logical signatures will not need to contain the declarations that tie theorem names to the formulas that they name. As a result of using a global naming mechanism for theorems, we can view signatures, referred to by the schematic variable Σ as only needing to contain kind, type, and definition declarations. The use of `cid`s also means that the same theorem that might have had two different local names will now get a single name via its global `cid`.

Another important technology that accompanies IPFS is IPLD, an extension of IPFS that accommodates various kinds of linked data. Using IPLD, it is possible to represent structured objects, such as sequents, as JSON objects, and for these to be given an identifying name based solely on their content.

5.3 Explicit signing of a sequent

Given the use of immutable storage and a global addressing scheme, the assertion that an agent claims that a given Σ-sequent is a valid entailment can be made globally known using two steps.

1. We first encode named formulas as IPLD objects. Then sequents, such as $\Sigma\colon N_1, \ldots, N_n \vdash N_0$, can be encoded by replacing the names N_i $(0 \leq i \leq n)$ with their `cid` within a JSON object that collects these various `cid`s into the encoding of that sequent. In this way, such sequents can be given their own `cid`.
2. We identify an agent with a public/private key pair and assume that public keys can be discovered in a distributed fashion. The assertion K *says* $\Sigma\colon N_1, \ldots, N_n \vdash N_0$ is an IPLD object that contains the signing by K's private key of the `cid` for $\Sigma\colon N_1, \ldots, N_n \vdash N_0$.

Both of these steps are elaborated in Section 6.1.

Although we will give more details about our use of IPFS, IPLD, JSON, and public key signing in the following section, we note here the following high-level features of this design. By simply having the `cid` for the assertion K *says* $\Sigma\colon N_1, \ldots, N_n \vdash N_0$, it is possible to access all components of the underlying objects: in particular, the agent's public key K as well as all the names $N_0, \ldots, N_n$ and what formulas they name.

6 The Dispatch tool

Our design consists of three components. The **Prover** (currently Abella), the **Dispatch** tool (new software), and the **Global Storage** (implemented using IPFS). From another perspective, this design consists of three processes or scenarios: **Publishing**, **Getting**, and **Proof Checking**. The main goal of the **Dispatch** tool is to mediate between the Prover and the Global Storage as follows:

```
Import "ipfs:bafyreigm32vqfhsqxrpodk6ynuehe3nite
qhp7gu65ie2dlltn6dew7pwu".

Theorem plus_succ :
 forall M N K, plus M N K -> plus M (s N) (s K).
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem plus_comm :
 forall M N K, nat K -> plus M N K -> plus N M K.
induction on 2. intros. case H2.
  apply plus_zero to H1. search.
  case H1. apply IH to H4 H3.
  apply plus_succ to H5. search.
```

Figure 4. The Abella theorem file `fileB.thm`.

1. `Dispatch` reads specific structures produced by the Prover, parses them, and constructs specific JSON objects which will be published through IPFS.
2. `Dispatch` reads published objects existing in *Global Storage* (identified by an IPFS `cid`), parses them, and constructs specific structures describing the global objects for the Prover to read and consume.

6.1 From Prover to Dispatch

6.1.1 Scenario: Publishing. After checking the theorems in a given file, the prover enters publishing mode in which a collection of sequents (one sequent per occurrence of the `Theorem` keyword) are printed. These sequents indicate which lemmas were used in a theorem's proof script. This collection of sequents, the *sequence object* as we will call it in the global context, will be the input for the `dispatch` tool.

6.1.2 Data Input to Dispatch. Consider `fileB.thm` file of Figure 4: where the imported `cid` is the global name for the sequence object of `fileA.thm` of Figure 2.

The input that `dispatch` expects from a prover is a JSON file with a standard format. To illustrate this format, consider the JSON file of Figure 5 as the input for `dispatch` corresponding to `fileB.thm`. As you can notice, the main entries correspond to theorems. Each of these entries have three attributes:

1. `"Sigma"`: Contains the necessary information needed to make sense of `"conclusion"`; the logical signature. In the current implementation, it is a list of strings corresponding to each `Kind`, `Type`, and `Define` declarations.
2. `"conclusion"`: Contains a string representation of the conclusion formula itself.
3. `"lemmas"`: Contains a list of lists of lemmas: each corresponding to a theorem's occurrence in the main `.thm` file or *a sequent object*.

Note that names of lemmas are used by `dispatch` to indicate what formulas should be linked in the sequent. For example, `"plus_succ"` indicates to `dispatch` to create a link

```
{ "plus_succ": {
        "Sigma": [
            "Kind nat      type",
            "Type zero     nat",
            "Type succ     nat -> nat",
            "Define nat : nat -> prop by nat zero ;
                 nat (succ N) := nat N" ],
        "conclusion": "forall M N K, plus M N K ->
             plus M (s N) (s K)",
        "lemmas": [ [] ] },
  "plus_comm": {
        "Sigma": [
            "Kind nat      type",
            "Type zero     nat",
            "Type succ     nat -> nat",
            "Define nat : nat -> prop by nat zero ;
                 nat (succ N) := nat N" ],
        "conclusion": "forall M N K, nat K -> plus M
             N K -> plus N M K",
        "lemmas": [ [ "ipfs:bafyreid6ivhia73..",
             "plus_succ" ] ]  }  }
```

Figure 5. Standard format input from prover to `dispatch`

to the *formula object* of `"plus_succ"` as a lemma in this *sequent object*. This lemma in the example is referred to by the theorem's name `"plus_succ"` as this theorem occurs in this same file. In contrast, the second lemma is referred to by a `cid` since that lemma exists in an imported structure and not in this same file. In this example, the `cid` in the `"lemmas"` attribute of the theorem entry `"plus_comm"` refers to the *formula object* of `"plus_zero"`. Note that it is not the same `cid` as the imported one; the imported `cid` refers to a *sequence object*, for example.

These are implementation choices for convenience, but it was necessary to illustrate the difference, which will become clearer in later sections.

6.1.3 Dispatch publish command.

In the current `dispatch` implementation, an *agent* is expected to call the `publish` command to invoke the parsing of the given JSON input file, creating, and publishing the objects describing it.

Consider the following call of this command:

```
> dispatch publish fileB Abella-v1 examples/input cloud
```

The result of this command is a single `cid` referring to the final *sequence object*. This `cid` is enough to retrieve the following structure.

```
> Input from Prover Published: The root cid of the published
  sequence of assertions by profile: Abella-v1 is
  bafyreialidp34p7w4vabe62qybxhexzeq5wjouuxiyhp2bouqot4tj2ruy
  DAG successfully published to web3.storage!
```

You can try to read the object referred to by this `cid` through this link. It should have the following format:

```
{ "format": "sequence", "name": "fileB",
  "sequents": [ { "/": "bafyreiapqrxflyeozgbsxf4aa.." },
                { "/": "bafyreigviuefn34wcwveylog5.." } ] }
```

The command's arguments, namely `fileB`, `Abella-v1`, and `examples/input`, refer to the JSON input file name, the public/private key profile name (for signing the sequents), and the `directory` in which the file exists starting from the directory of execution respectively. The final argument of value `cloud` is an indicator for the tool to publish the final structure through the `web3.storage api` which provides one solution to making published data more quickly discoverable by storing or *pinning* (using IPFS terminology) it on other remote IPFS peers. If the value `local` is used, the published data is only pinned initially on the agent's own machine until another agent chooses to get and pin it.

```
type cid = string

type ipfsLink = { "/": cid }

type formula = {
    "format": "formula",
    "formula": string,
    "Sigma": [string] }

type namedFormula = {
    "format": "named-formula",
    "name": string,
    "formula": ipfsLink }

type sequent = {
    "format": "sequent",
    "lemmas": [ipfsLink],
    "conclusion": ipfsLink }

type assertion = {
    "format": "assertion",
    "agent": string,
    "sequent": ipfsLink,
    "signature": string }

type sequence = {
    "format": "sequence",
    "name": string,
    "sequents": [ipfsLink] }
```

Figure 6. Type declaration for global objects

6.2 In Global Storage

The `TypeScript` declaration in Figure 6 defines the five main types of objects that exist in the global context (i.e., the objects published and retrieved by `dispatch`). The `formula` type contains the `"formula"` attribute, which is a string representation of a formula, and the `"Sigma"` attribute which is a list of strings denoting a logical signature. The `namedFormula` type contains the `"name"` attribute that provides the formula's name originally given by the agent, and the `"formula"` attribute that links to the actual formula object defined first, which is convenient to have. (In the rest of the paper, we shall refer to both of these types simply as formula objects.) In the `sequent` type, both the `"lemmas"` and `"conclusion"` attributes employ links (`cids`) to formula objects. In the `assertion` type, the `"agent"` attribute denotes the fingerprint of the signing agent's public key, the `"sequent"` attribute links to the sequent object through its `cid`, and

`"signature"` attribute contributes the cryptographic signature of the linked sequent object's `cid` by the private key associated with `"agent"`. In the `sequence` type, the `"name"` attribute contains the originally given name of the input

JSON file, and the "sequents" attribute contains links to the included sequent objects.

Note the usage of the type ipfsLink in Figure 6. As the goal is to publish structured objects that can link to each other and where each object's cid is a global name, we take advantage of using both IPFS and IPLD through the ipfs dag api. IPLD (*InterPlanetary Linked Data*) aims to be the data layer for content-addressed systems, meaning that its general goal is to provide a common ground for serializing structured data. To illustrate the interplay between IPFS cids and JSON objects (as provided by the IPLD extension to IPFS), let cidA be the cid for some sequent object. The command

```
ipfs dag get cidA/conclusion
```

will return the formula object for the conclusion of this sequent. This way, you can explore the DAG associated with a cid. For example, by using the path

bafyreialidp34p7w4vabe62qybxhe.../sequents/0

you will get the first assertion object (as the current implementation of the publish command publishes a sequence of signed sequents). By adding the suffix **/sequent** you will get the sequent object, and so on.

6.3 From Dispatch to Prover

Similar to how a standard format input was provided to dispatch from a Prover, another standard format input is produced by dispatch for a Prover to consume objects from the Global Storage. Since dispatch mediates between the Prover and the Global Storage, the file produced by dispatch is constructed so that the Prover can consume the objects in that file without having to deal directly with IPFS.

6.3.1 Dispatch get Command. An *agent* is expected to invoke the get command of dispatch to retrieve published objects starting from a given cid. The result of this command is a JSON file that the Prover can consume. Consider, for example, the following command.

```
> dispatch get bafyreialidp34p7w4vabe62qybxhexz.. examples/output
```

The result of this command is a single JSON file that results from flattening (dereferencing all the cids) starting with the given argument root cid.

```
Input to Prover Constructed: DAG referred to by this cid
is in the file named bafyreialidp34p7w4vabe62qybx....json
```

The first argument following the get command is the cid of the DAG to be retrieved, and the second is the directory in which the output JSON file shall be stored (relative to the directory of execution).

The structure of the output JSON file is presented next.

6.3.2 Data Input to Prover. The input that a Prover should expect from dispatch is a JSON file. As an example, the JSON file in Figure 7 was produced by the dispatch get command when given, as its first argument, the cid of the sequence object corresponding to fileB.thm (Figure 4).

```
{ "plus_succ": {
    "cidFormula": "bafyreidk2fl2j5gpa6sk4t..",
    "formula": "forall M N K, plus M N K -> plus M (s
        N) (s K)",
    "SigmaFormula": [
        "Kind nat      type",
        "Type zero     nat",
        "Type succ     nat -> nat",
        "Define nat : nat -> prop by
            nat zero ; nat (succ N) := nat N" ],
    "sequents": [
        { "lemmas": [],
          "signer": "22f363b82059249fdbd1a09.." }  ]},
  "plus_comm": {
    "cidFormula": "bafyreihktm54xqmbquuwk3..",
    "formula": "forall M N K, nat K -> plus M N K ->
        plus N M K",
    "SigmaFormula": [
        "Kind nat      type",
        "Type zero     nat",
        "Type succ     nat -> nat",
        "Define nat : nat -> prop by
            nat zero ; nat (succ N) := nat N" ],
     "sequents": [
        { "lemmas": [
          { "name": "plus_zero", "cidFormula":
              "bafyreid6iv..",
            "formula": "forall N, nat N -> plus N z
                N",
            "SigmaFormula": [ "..." ] },
          { "name": "plus_succ", "cidFormula":
              "bafyreidk2f..",
            "formula": "forall M N K, plus M N K ->
                plus M (s N) (s K)",
            "SigmaFormula": [ "..." ] } ],
          "signer": "22f363b82059249fdbd1a.." }  ]}  }
```

Figure 7. Standard format input from dispatch to prover

This JSON format is designed to satisfy two main goals.

1. The JSON file removes all references to IPFS cids. This way, the Prover does not need to deal directly with IPFS.
2. The JSON file should provide a structure that is convenient for the Prover to consume. The Prover will probably need to make some proper checks, such as not allowing to import two theorems having the same name (descriptive name), and the JSON structure should aid in making such checks.

These JSON files consist of an entry per theorem (i.e., these are named formulas). Each entry contains the cid of the formula object, the formula itself, its logical signature, and a list of sequents having this formula as their conclusion. An element of this sequent list contains the details specific to each sequent's lemmas along with the signer of the sequent.

While the example in Figure 7 represents the result of using the dispatch get command starting from a sequence object cid, the same command starting from the cid of a sequent or an assertion object is similar. In contrast, calling the dispatch get command on the cid of a formula object yields no JSON file since we do not import a simple formula object.

How the Prover consumes such JSON files will be discussed next.

6.4 In Prover

We mentioned earlier that our described design consists mainly of three interacting components. The impact of the design of our framework on the Prover is discussed in this section.

6.4.1 Scenario: Proof Checking. Proof checking is the main functionality of a Prover (a.k.a. a Proof Checker). Our primary concern has been: What will need to change within a Prover when global objects with global addresses (`cids`) are introduced? We decided earlier that a Prover shall only consume the specified output standard format JSON file as produced by `dispatch`. Thus, controlling what objects are allowed to be consumed by a prover (and their meaning) is decided at the `get` command level by `dispatch` and not by the prover. On the other hand, the prover decides how to consume the provided JSON file. So, the core meaning of imported structures shall be the same for all provers, and only minor details might differ between them (for example, in printing the results of proof-checking a file).

6.4.2 Application to Abella. In Abella, the `Import` command is used to import the logical signature of some previously compiled `.thm` file into a new file. From our framework's perspective, this corresponds to generating a list (a sequence) of signed sequents generated from the imported file by the executing agent. The signing is done using this agent's private/public key (a local-only key not known to others).

Abella allows the `Import` command to bring in previously checked theorems from a local `.thm`. The first change that was needed with Abella was extending the `Import` command to allow an argument of the form "ipfs:cidA". In this case, Abella has new code that calls the `dispatch get` command to retrieve the corresponding JSON file. In such a JSON file is contained information about (1) what sequents are imported, (2) which agents signed each sequent, and (3) all the information needed to make proper sense of these sequents and the formulas they include.

The JSON that results from a `dispatch get` command contains, within objects of the `assertion` type, the fingerprint of an agent's public key. We do not expect Abella to do anything directly with this item: instead, we can make some simple logical inference in the says-logic outside of Abella. For example, the trust information related to the assertions of a sequence of sequents can be represented as a list such as

$$[\text{"}K_1 \; says \; (A, B \vdash C)\text{"},\ \text{"}K_2 \; says \; (\vdash D)\text{"},\ \text{"}D \vdash F\text{"}].$$

Here $A, B, C, \ldots$ are names of formulas and K_1, K_2 are names of agents. The elements having a *"says"* in them refer to signed sequents (assertions), and the elements without it refer to unsigned sequents where they are considered as assumed "axioms" once imported. Some form of the elements of this list could be processed further in order to retrieve additional information. For example, if such a list contains $K_1 \; says A \vdash B$ and $K_2 says \vdash A$, and agent K_2 is willing to trust agent K_1 (encoded by the formula $[K_1 \mapsto K_2]$), then a simple says-logic deduction is $K_2 says \vdash B$.

6.4.3 In publishing mode. After the successful proof-checking of a collection of theorems and their proofs, the Prover needs to be modified to have a new output mode, called here the *publishing mode*. In this mode, the JSON file needed to call the `dispatch publish` command must be composed (Section 6.3.2). The following call to the Abella prover illustrates this publishing mode.

```
> abella --ipfs-imports --ipfs-publish cloud fileB.thm
  Import "ipfs:bafyreiafar6l6j4orqro5kvfwoaemrpc367hooe..".
  Theorem plus_succ : forall M N K, plus M N K -> .. .
  induction on 1. intros. case H1. search. ..
  Proof completed
  Theorem plus_comm: ... Proof completed
  Published as ipfs:bafyreid5jg35p43cnjx3fpe5nmf6mrjejq..
```

Given this framework, it is easy to make some simple extensions. For example, if a local file is imported into a file being published, the local file can be published but with its assertions signed by a "local-only" public key that only the publishing agent knows and trusts. This signing resembles the normal compilation process of Abella where a compiled theorem file (extension `.thc`) file is produced.

7 Comments on this framework

7.1 Users and profiles

As mentioned in Section 4.2, public and private key pairs are probably best associated not with proof checkers alone but with a pairing of a user and a *profile*. For example, a person using Abella versus the same person using Coq could result in different trust relations with others. Similarly, using different versions of a proof checker can be encoded as different profiles. It also seems sensible for users to have a profile, say `open`, that can be used to sign sequents for which no proof is known for the signed sequent. Such a profile would allow a user to publish a collection of conjectured theorems or a proposed set of axioms. By the use of signing, it would be possible to track which theorems have been proved assuming such conjectures or axioms.

7.2 Using simpler and specialized kernels

Modern autarkic provers routinely recheck proofs, often after every invocation of a new instance of the proof checker and certainly after every change in the version number of the prover. As a result of needing to recheck proofs, there is a tendency for implementers of proof checkers to optimize such kernels to be more efficient. However, such optimizations can add greater complexity to a kernel, which, in turn, makes errors in the kernel more likely to occur. With the framework described in this paper, once a trusted kernel

checks a proof, it does not need to be rechecked. As a result, there will be less pressure for kernels to be optimized, and, as a result, they might be allowed to remain simpler and, hopefully, less prone to errors.

Abella can benefit from proofs from λProlog and the Bedwyr model checker, both of which work in subsets of the logic used by Abella. These systems could do what they are specialized to do and then output their theorems without any proof evidence. We need to sign their output as being performed by one of these systems and explicitly trust such signed theorems.

7.3 The nature of attacks against this framework

The immutability of IPLD objects means that no matter how a `cid` is resolved to a file or JSON object—via the IPFS network, HTTP, or a local cache—no malicious actors can corrupt that dereferencing process undetected: one needs to compute the hash of the received object and compare it to the `cid` used to fetch it.

Signing is associated to both a user and a proof checker: signing along by a proof checker might miss the fact that a malicious users can corrupt the proof checker and its outputs.

Agents might be large corporations or large libraries of checked proofs. Given the global and transparent nature of this framework, if such a corporation or library signs a sequent later demonstrated to be false, there is no way to hide that error and that company's economic value might be adversely affected. Thus, for any agent with an economic interest in proof checking, such transparency should lead them to be careful in what they sign.

When using this framework, there are several software packages that need to be trusted. Besides the obvious need to have some trust in operating systems, network software, etc, we must trust the off-the-shelf software we used (IPFS, IPLD, and public-private key signing). We must also trust the implementation of the `dispatch` software as well as the new code for printing and parsing JSON objects within Abella. Whether or not one wishes to actually trust the Abella kernel is an option that is explicitly enabled by our framework.

7.4 Two lessons from the World Wide Web

In a couple of decades, the World Wide Web changed everything about how documents were produced and shared and how correspondence was done. By designing some protocols and standards (e.g., `HTTP` and `HTML`) as well as some tools (e.g., web browsers), the sharing of documents created a revolution in the way people access information. We point out two lessons from the World Wide Web that are particularly relevant to us here.

Standardization versus emergence. One of the strengths of the early web was that it developed some standards but did not standardize too much. Instead, many features we now see as integral to our use of the web—ranging from curated sites like Wikipedia to programmable web content based on JavaScript—were left to evolve along their trajectory. Thus a goal of early standards should allow for a diversity of new structures to *emerge* later. In particular, we have left formal proofs out of our current framework. Of course, we can eventually integrate formal proofs into our framework, but since they vary a great deal and can be quite large and resource-intensive objects to manage, we felt it was better to leave them as an emergent feature of this framework. The existence of proof objects, especially if multiple provers can check such objects, could have a big impact on the kind of agents one might be willing to trust.

Moving from a cooperative to an adversarial environment. At the beginning of the world wide web, most web users were academics who mainly used this distributed information system in a *cooperative* fashion. If someone found a bug in a protocol or implementation, that bug was reported so it could be fixed; back-doors existed to allow for testing; and information sources such as academic papers, user manuals, etc., communicated true and fact-based information. In the decades since the invention of the World Wide Web, however, we have seen that the web must now defend against adversarial behaviors. Today, if a bug in a protocol or implementation is found, that bug can be sold on a market to people interested in exploiting it; back doors are placed in systems to allow bad actors to infiltrate or remotely monitor those systems; and information sources can be lies meant to manipulate voters. Many new techniques - cryptographic in nature - have been invented and deployed to provide for authentication, privacy, and transparency. Given that we are seeing formal proofs moving out of the academic and cooperative setting, we can expect to see adversaries attempting attacks on provers. For example, attempting to prove a theorem (like Kepler's conjecture) might cause many lemmas to be conjectured, and the provers appropriate for those lemmas might vary significantly. Money might also be paid to anyone who can prove some of these lemmas. In that setting, anyone who can find a bug in a proof-checking kernel and convert that bug into a proof of false could make money by proving all the conjectured lemmas. Our use of cryptographic signatures to track users and the provers they employ is a low-level device to keep track of which kernels are being trusted so that when bugs in kernels are found, we know which previously trusted theorems might be tainted.

7.5 Future development

We briefly describe several additional developments that can accompany our framework.

Other proof checkers. A clearly desirable next step in the framework is to use it with other proof systems. For this, a natural early step would be to support the TPTP format [Sut09]. A variant of our dispatch tool that parses the TPTP format could be helpful for publishing the results of the

CADE ATP competition [Sut16] since the competing provers will be signing—in a transparent and global setting—any theorems for which they wish to take credit.

Dealing with more realistic declarations. Most provers place more things into their signatures and preambles than what we assume here. Examples of such additional declarations are infix declarations, pragmas for the kernel's operation, descriptions of how equations might be used within deductions, etc. Even Abella has additional declarations that are not addressed explicitly here, most notably the `Specification` declaration. We need to find ways to accommodate at least some such declarations.

Libraries. Although our framework does not provide any *a priori* notion of hierarchical structure on collections of definitions and theorems, it is natural for such organizations to be built. Such structures can be seen as emergent structures enabled by this framework. Of course, the same definitions and theorems could be organized and used in rather different ways in different curated libraries. It also seems wrong to insist on imposing the top-down structure found in libraries as part of our framework, since this framework enables is a more bottom-up building of collections of definitions and theorems.

Meta-data. We currently store almost nothing that is usually considered meta-data in our JSON objects, even though such information can be extremely important for activities beyond simply trusting kernels. For example, attributing authors, a date, and some keywords to a theorem and proof could prove valuable for placing certain objects into curated libraries.

Web-based proof checking. Given the fact that Abella is implemented in OCaml and that OCaml can be compiled into JavaScript, Abella has been deployed via a web browser (try this link) instead of via more complex installation steps. A significant problem with such web-based deployment of proof checkers is that there needs to be some approach to the *persistence* of collections of definitions, theorems, and proofs. Our use of IPFS can provide a natural and robust solution to that problem.

7.6 Deploying this framework beyond proof checking

We have not made proof objects a central theme of our framework. One consequence of this choice is that the connection between *justification* and *trust* is open to other interpretations: it does not need to rely on a proof-checking kernel. There seems no specific reason why a framework such as the one that we describe here—that tracks assertions made by agents who we may or may not trust—could not be applied in a much wider setting than that involving the checking of formal proofs. For example, the problem of reproducing scientific results [FW21] has gained a great deal of interest in recent years. Some websites, such as the Open Science Foundation provide a top-down organization for storing, sharing, and analyzing data collected from scientific experiments. Parts of the framework described here could be used to support such efforts but in a more bottom-up setting by tracking the provenance of databases and analysis tools.

Of course, one can also return to the issue of merging the upper layer and the middle layer of the semantic web (see the discussion in Section 1).

8 Conclusion

We have described a framework in which assertions that one proof-checking kernel makes about a theorem can be made available to any other proof-checking kernel willing to trust it. We have given a high-level overview of how this framework was designed, and we discussed our prototype implementation of it using the off-the-shelf, web-based technologies JSON, IPFS, IPLD, and public key cryptography. Finally, we have described how this framework can be used with the Abella theorem prover: in particular, the only things that need to be changed in the Abella system is some facilities to input and output certain JSON objects. No changes to the underlying logic and kernel needed to be considered. We plan to explore next how other provers can use essentially the same tools and technologies to explicitly allow trust relations with other provers in a highly distributed setting.

References

[Aba08] Martín Abadi. Variations in access control logic. In Ron van der Meyden and Leendert W. N. van der Torre, editors, *Deontic Logic in Computer Science, 9th International Conference, DEON 2008, Luxembourg, Luxembourg, July 15-18, 2008. Proceedings*, volume 5076 of *LNCS*, pages 96–109. Springer, 2008.

[ABC+16] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory, 2016.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP 2011)*, volume 7086 of *LNCS*, pages 135–150, 2011.

[ANS21] French National Cybersecurity Agency ANSSI. Requirements on the use of coq in the context of common criteria evaluations. https://www.ssi.gouv.fr/uploads/2014/11/anssi-requirements-on-the-use-of-coq-in-the-context-of-common-criteria-evaluations-v1.1-en.pdf, December 2021. v1.1.

[BB02] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.

[BCG+14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[Ben14] Juan Benet. IPFS-content addressed, versioned, P2P file system, 2014.

[BL98] Tim Berners-Lee. Semantic Web road map. Technical report, W3C Design Issues, 1998.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American Magazine*, May 2001.

[CHOS13] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013*, volume 7737 of *LNCS*, pages 275–294. Springer, 2013.

[Chu40] Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[DT19] Gilles Dowek and François Thiré. Logipedia: a multi-system encyclopedia of formal proofs. http://www.lsv.fr/ dowek/Publi/logipedia.pdf, 2019.

[FMM+06] 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 H. Hermanns and J. Palsberg, editors, *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.

[FMP15] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2–A survey. *J. of Automated Reasoning*, 55(4):307–372, 2015.

[FW21] Fiona Fidler and John Wilcox. Reproducibility of Scientific Results. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.

[Har17] John Harrison. *The HOL Light tutorial*, 2017.

[Log22] Logipedia in a nutshell. http://logipedia.inria.fr/about/about.php, 2022.

[Rus05] John M. Rushby. An evidential tool bus. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *LNCS*, pages 36–36. Springer, 2005.

[Sha02] Natarajan Shankar. Little engines of proof. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, Proceedings of the International Symposium of Formal Methods Europe (Copenhagen, Denmark)*, volume 2391 of *LNCS*, pages 1–20. Springer, July 2002.

[Sut09] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Sut16] Geoff Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[Tiu08] Alwen Tiu. On the role of names in reasoning about λ-tree syntax specifications. In Andreas Abel and Christian Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, pages 32–46, 2008.