Formal Reasoning using Distributed Assertions

Farah Al Wardani, Kaustuv Chaudhuri, and Dale Miller

Inria Saclay & LIX, Institut Polytechnique Paris

Abstract. When a proof system checks a formal proof, we can say that the kernel *asserts* that the formula is a theorem in a particular logic. We describe a general framework in which such assertions can be made global so that any other proof assistant willing to trust the creator of the assertion can use that assertion without rechecking any associated formal proof. This framework is heterogeneous and allows each participant to decide which tools and operators they are willing to trust in order to accept external assertions. This framework can also be integrated into existing proof systems by making minor changes to the input and output subsystems of the prover. It achieves a high level of distributivity using off-the-shelf technologies: IPFS, IPLD, and public key cryptography. We illustrate the framework by providing implementations of an intermediate tool for validating and publishing assertion objects and a modified version of the Abella theorem prover that can use and publish such assertions.

1 Introduction

In order to communicate a result from one formal reasoning system to another, a common technique is to transfer a formal proof certificate from the source system to the target system. This technique is usually required when the target system is *autarkic*,[1] wherein the system only trusts its own components, of which a particularly trusted component may be an implementation of a proof checking *kernel*. To transfer a formal proof to an autarkic target system, either (a) the proof has to be *translated* from the source system, or (b) the verifier for the proof must be re-implemented as a *certified* procedure in the target system [5,25]. Both kinds of transferral are complicated for a variety of reasons: (1) The source and target system may not be syntactically, semantically, or foundationally compatible. (2) The source proof language can have complex operational semantics that is cumbersome to encode in the target system. (Note that no universal standard has yet emerged for encoding the formal semantics of arbitrary proof languages; cf. sec. 5.) (3) As systems change and mature, older versions of proof certificates can become stale and unmaintained. (4) Perhaps most importantly, there are many popular reasoning systems that do not produce proof certificates at all. Prominent examples of that latter are SMT solvers that are not certifying when memory size and execution time are critical [32] and the specification tool Twelf [43] when using non-certifying procedures (e.g., totality checking).

[1] In [11], the adjective *autarkic* was applied to computational components of a proof checker but not to an entire proof checker.

Formal reasoning systems that are *non-autarkic* have an additional way to interact with external provers that addresses many of the above issues. In such systems, a host system is designed to build *proof obligations* that are then dispatched to external systems to solve. While these external systems may produce proofs, the host system usually does not check the proofs and instead *trusts the executions* of the external systems. This system architecture is most commonly used in program verification tools such as Dafny [28], Why3 [24], and TLAPS [15]. One issue not addressed with this enlarged view of trust is that the external dependencies tend to have unclear descriptions, especially from a third party perspective. To illustrate, Dafny may declare that it trusts "Z3 v.4.12.1", but what does this mean? Is this external dependency to be interpreted by name, in which case any tool called "Z3 v.4.12.1" can be used, or is it precisely identified by, e.g., (a cryptographic hash of) the source code (or better, an executable binary) of a particular tool called "Z3 v.4.12.1"? Even with a precise identification, an external executable dependency may not be practical to incorporate. For example, the HOL Light system [27] re-checks its entire standard library every time it is started, taking on the order of minutes. If a development involves a number of calls to an external HOL Light-based solver, how are the calls to be orchestrated?

In addition to these two bases of trust—autarkic based on proof certificates, and non-autarkic based on executions of external tools—there is at least one other basis of trust in any heterogeneous development: the *agents* that write and assemble the developments and execute the formal tools as required (checkers, solvers, etc.). An example of an agent is a user, but one individual user can have many *agent profiles* (see sec. 3.2), and entities such as a trustworthy central database can also correspond to an agent. Agents as trust bearers has been largely neglected in the formal reasoning world, but is common in other settings of high reliability such as security. Nevertheless, agents are at least implicitly present in any formal development: to claim that a result has been formally achieved is tantamount to saying that some trustworthy agent (e.g., peer reviewers) has *correctly and successfully* executed a specific collection of formal tools to convince themselves of that formal result. Furthermore, if one agent A *trusts* another B, there is no need for A to re-check B's proof scripts and re-execute any tools that B used to construct the result.

In this paper, we propose a framework where a *distributed* collection of agents can exchange formal results (called *assertions*), where the results have an unimpeachable *provenance*, and where each agent is in full control of their trust parameters. This *Distributed Assertion Management Framework* (DAMF) is:

– *Decentralized*: a global notion of *truth* is not imposed on every participant by the means of a privileged logic, language, system, or software. This linguistic independence makes DAMF different from formalisms such as the *evidential tool bus* [19,39] that have been proposed for integrating external reasoning agents into a unified formal system. Participants in DAMF are free to combine assertions from different sources if they believe the combination to be meaningful. Any participant can *retrieve* and use any assertion they understand, and this external import will be explicitly marked as a *dependency* if they choose to *publish* assertions they build with such external imports.

- *Reliable*: assertions have an *irrepudiable* provenance, i.e., the fact that an agent has published an assertion is locally verifiable and independent of any other aspect of DAMF. Assertions therefore need to be *immutably* and *eternally* available, even in the presence of intermittent infrastructure and nefarious users or tools.
- *Composable*: assertions are not rigidly constrained by their own *history*; new logical artifacts such as *theories*, *libraries*, *proof outlines*, etc. can be crafted by reorganizing existing assertions based on their declared dependencies.
- *Egalitarian*: the barrier to entry is low for participants who want to produce *or* consume such assertions.
- *Status Quo Compatible*: existing work that has already been done with currently mainstream systems is readily incorporated as assertions without needing to modify any existing system.

Concretely, DAMF provides JSON-based representations of a small number of concepts such as formulas, assertions, dependencies, etc. *without* any up front commitment to a formal syntax or any particular semantics. These objects are then added to a *global store* in terms of the *InterPlanetary File System* (IPFS) [12] using linked data in the *InterPlanetary Linked Data* (IPLD) format. An object in IPFS/IPLD is denoted by a *canonical* content identifier (`cid`) that is a cryptographic hash of its content. Knowing the `cid` is sufficient to retrieve the object by any participant of the IPFS network. Furthermore, the `cid`s serve as the only externally visible *names* in DAMF, and links between objects are made using these `cid`s by IPLD. Features specific to a particular language or system, such as constants, variables, definitions, notations, etc. are kept localized to particular *formula objects*. Assertions are built using (the `cid`s of) formula objects and *signed* by their creator agents using public key cryptography. IPFS is used to distribute DAMF objects transparently using a variety of technologies whose precise details are not relevant for this paper.

This paper is accompanied by two concrete implementations that illustrate DAMF. First, we provide a tool called Dispatch that can be used by users and systems to both produce and consume DAMF assertions. Dispatch is not a privileged tool in DAMF: users and systems can interact directly with DAMF objects in IPFS if they so choose. Dispatch is simply one *interface* to the DAMF *global store* making the integration of producers and consumers minimally demanding. It does tasks such as schematically validating the concrete JSON objects that are added to or retrieved from the global store. Dispatch also helps to analyze and modify the trust parameters for (compositions of) assertions.

Second, we implement a version of the Abella interactive theorem prover [9] that can produce and consume assertions in DAMF, mediated by Dispatch. As an example of its use, we show how Abella can use a lemma that was stated and proved using the automated linear arithmetic reasoning tactics of Coq (v. 8.16.1); this lemma is manually translated from the Coq to the Abella language, with an explicit dependency on its Coq development, and added to the global store by the present authors. A user can accept this heterogeneous development as long as they trust Coq, Abella, and our translation of the Coq lemma to Abella. Moreover, this assertion, which contains explicit links to the externally sourced DAMF imports, can be published back to DAMF for use by others.

Since dependencies are explicitly tracked in DAMF assertions, it is possible for any user to analyze various aspects of how it was composed from other assertions. Such analysis can form the basis of various kinds of *investigations*: for example, if a formula is found to be a non-theorem, an investigator can explore the compositions of the DAMF assertions that yield that formula in order to find the agents whose trust parameters may need to be modified. The Dispatch tool mentioned above comes with a command called *lookup* that explores combinations of known assertions that ultimately yield a desired result; for each such composition, the analysis extracts the collection of agents (and tools) that could be *trusted* in order to accept that composition.

In the next section, we describe the abstract design of DAMF and its underlying logic of assertions, which forms the basis of the investigations mentioned above. Section 3 describes our concrete implementation of DAMF, Section 4 discusses some of the design choices in DAMF, and Section 5 discusses some related work. The specific software tools (Dispatch and Abella-DAMF) accompanying this paper are fully documented at: `https://distributed-assertions.github.io/`.

2 Design of DAMF

2.1 Languages, contexts, and formulas

To transfer a theorem from a source proof system to a target proof system, we must be able to transfer the statement of the theorem, which we represent as a *formula* object in DAMF. To be as general as possible, we represent the content of such a formula as a *string*, i.e., in a format suitable as an input to a parser of the source proof system. 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.—that are 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 features that are globally available; 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 that is used in the formula is defined in the language or the context. From the perspective of DAMF, a formula object is an atomic entity and there are no reasoning principles on the language or context components. For instance, there is no mechanism in DAMF that 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.

2.2 Sequents 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. As an example, suppose the `Coq 8.16.1` theorem in sec. 2.1 has a proof that appeals to the lemma `lem : forall m n, m <= n -> S m <= n \/ m = n`. The sequent that is produced is conceptually of the form `lem` $\vdash$ `ex_coq`, though concretely we would have to build DAMF formula objects by packaging the language and contexts.

An *agent* is a globally unique name; we use the schematic variable K to range over agents. 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* S, where K is an agent and S a sequent, is an *assertion*. The *says* predicate is implemented in DAMF using public-key cryptography. In a DAMF-aware proof system, when an appeal is made—say as part of the proof of some other theorem—to an assertion K *says* $(N_1, \ldots, N_k \vdash N_0)$, the appeal is interpreted as follows:

- The agent K 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.
- The conclusion of the assertion, N_0, contains the formula representing the lemma that is being appealed to. Note, in particular, that the dependencies $N_1, \ldots, N_k$ do not participate as such in the appeal.

2.3 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.[2]

```
1 Import "nats". % some natural numbers library
2 Define lincomb : nat -> nat -> nat -> prop by
3   lincomb N J K := exists X Y U V,
4     times X J U /\ times Y K V /\ plus U V N.
5 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

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

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.

2.4 Composing assertions, trust

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

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

[2] This encoding of functions using relations is the usual one: see [16] for details.

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, in particular, not be a sequent that has been explicitly signed by the agent K even if both premises are. Rather, 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.

2.5 Producing assertions, formal reasoning tools

Conceptually, an agent constructs a DAMF sequent as a consequence of running formal reasoning tools such as proof checkers or theorem provers. DAMF includes *tool objects*, which are unconstrained JSON objects that can be used to describe such tools or tool collections. Like with languages in sec. 2.1, we compare tools for equality by means of the `cid`s of these tool objects. It is also possible for an agent to build a DAMF sequent manually, without running any tool. The agent may do this for a number of reasons: e.g., the assertion may be an *axiom* (i.e., no proof is expected) or a *conjecture* (i.e., a proof may be provided at some other time but is currently missing).

A DAMF *production* is a sequent that is annotated with a *mode* that describes how the sequent was produced; this mode can be the `cid` of a tool object mentioned above, or it can be one of a small number of keywords such as *axiom*, *conjecture*; this mode annotation can also be omitted. We use the schematic variable T for modes, and write a production of the sequent $\Gamma \vdash N$ with mode T as $\Gamma \vdash_T N$. Published DAMF assertions will be of the form K *says* $(\Gamma \vdash_T N)$, and we modify the TRUST rule to the following:

$$\frac{(K_1 \textit{ says } (\Gamma \vdash_T N)) \text{ is published}}{K_2 \textit{ says } (\Gamma \vdash N)} \ \textsc{Trust}\ [K_1/T \mapsto K_2]$$

where the side condition $[K_1/T \mapsto K_2]$ means that K_2 allows K_1's assertions in mode T. It may be tempting to think of K_1/T as an agent by itself, but, as we shall see in sec. 3.1, agents are implemented in DAMF using keypairs, so if K_1/T_1 and K_1/T_2 were separate agents then there would be no verifiable way to link them both to K_1. This use of modes makes it possible, for example, to trust an agent K using any version of Coq while not trusting K when using other proof systems.

2.6 Logical consistency of heterogeneous combinations

DAMF imposes no constraints on the composition of assertions, which can at first glance appear to be risky. For example, suppose the assertions come from incompatible logics, say an assertion in classical logic during the proof of an intuitionistic theorem. Without exceptional care, the result of a COMPOSE will only be classically, not intuitionistically, true. Similar problems exist if the imported assertion requires additional axioms that are incompatible with the user's setting (e.g. extensionality or UIP in the setting of univalence).

This issue highlights the fact that DAMF *does not* guarantee logical compatibility of assertions; rather, DAMF is more accurately seen as a *record* of compositions that have been made. To trust an agent's assertion is just to say that we trust that the agent indeed had good reasons (such as a proof) to make that assertion, *not* that the assertion may be arbitrarily composed. Moreover, DAMF assertions are intended to be read as *hypothetical statements* from dependencies to conclusions (where "*hypothetical*" is understood in the informal language of discourse rather than as a formal implication or entailment). If the dependencies cannot be met, the assertion is useless. To illustrate, if an agent K wants to use an assertion $\Gamma \vdash M$ in their proof of N, the assertion they will publish is K *says* $(M \vdash N)$, which is acceptable in isolation; if M is incompatible with the logic of N, then the assertion K *says* $(M \vdash N)$ is vacuous.

3 Implementation: Information, processes, and tools

3.1 The structures of the global store

A crucial design criterion of DAMF is that the assertions and their constituent objects are a globally shared commodity, existing independently of the tools that produce or consume them. To this end, DAMF requires well-defined basic structures that producers would produce and consumers would expect and know how to address.

The use of a content-addressing scheme is an essential part of seeing these structures as global. Each structure is identified and addressed by a unique global identifier in a common namespace in an independently verifiable and trusted way: the identifier is derived from the content itself and every alteration of the content produces a new identifier; at the DAMF level, *the content is the name/address*, and comparing two objects structurally at the DAMF level is reduced to comparing their `cid`s as strings. One way to handle differences in `cid`s between different forms of conceptually the same DAMF object is by curation and normalization of such structures at the level of producers or potentially other DAMF actors.

The structures we may want to specify in DAMF are built by composing several elements; for instance, a *sequent* contains *formula* structures, which themselves contain *context* structures. In DAMF, we make the design choice to treat all such structures as *first class* objects stored in a distributed network through IPFS, and use the linked data representation of IPLD to represent an object as being composed of other objects.

The core DAMF structures we define are *context*, *formula*, *sequent*, *production*, and *assertion*. Concretely, these structures are represented as JSON objects with a varying `format` property which has the type of the structure as its value. These structures are described as follows (Appendix A contains the full definitions):

- *Context*: contains a `language` field, which is an IPLD link to a *language* object, described in sec. 2.1, and a `content` field containing the body of the context.
- *Formula*: contains a `language` field, a `content` field for a string representation of the formula in the language, and a `context` field that is an IPLD link to a context object, as described in sec. 2.1.
- *Sequent*: a `dependencies` field mapped to a list of IPLD links to formula objects, and a `conclusion` field as an IPLD link to a formula object.

– *Production*: pairs a sequent object with a `mode` field denoting a *mode of production* of a sequent as described in sec. 2.5.
– *Assertion*: a `claim` field mapped to an IPLD link to a production (currently considered the main claim type in DAMF), an `agent` field mapped to a public key, and a `signature` field containing the result of signing the `cid` of the value of the `claim` field.

Given these schemata, the aspects of tracking and trusting become natural: a formula present as a `dependency` in some assertion could be matched with the same formula present as the `conclusion` of a different assertion.

It is also useful to annotate these core DAMF objects with additional metadata such as external names, proof objects, timestamps, etc. In DAMF, we have chosen to give the core objects a `cid` independent of the metadata; instead, for every core object, we define an *annotated* object that is composed of a link to the core object and a link to any additional metadata. DAMF follows the design principle that objects are to be considered equal at the DAMF level if they have the same `cid`: the content of the objects is not examined, and no IPLD-links are followed for such comparisons. Generally speaking, therefore, DAMF core objects will not link to annotated objects, since the annotations will factor into the `cid`s and force disequality when undesired, such as when building compositions (sec. 2.4). The sole exception to this rule of thumb are assertion objects which can use annotated production objects as their claims. Note that every assertion object will be globally unique when produced: it will have a different `cid` each time its claim is signed, even if signed by the same agent, because cryptographic signatures always include a nonce.

Another layer of structures that can aggregate global object references are *collections*. We currently define one generic *collection* format in our implementation: many other non-generic collection formats can easily be considered.

3.2 Processes in DAMF, and Dispatch as an intermediary tool

The two obvious processes in DAMF are the *production* and *consumption* of DAMF objects. In a *production* process, DAMF objects are constructed starting from local information, published, and then stored across the distributed network. The *consumption* process is in the opposite direction: locally consumable information are constructed from DAMF objects. The important point is that these DAMF objects are common and well-understood (as DAMF formats) for all consumers, and each consumer decides what to consume and how to consume it. For example, a consumer might only choose to read formulas that are of some specific language, and then decide how to process their internal structures based on its own criteria. Other than these two, other processes will be done on the published DAMF objects that will incorporate their combination, curation, and analysis. The process we consider first in our implementation is *lookup* which will be discussed further below. Individual producers and consumers, such as theorem provers, can choose to implement some or several of these DAMF processes. However, many aspects of dealing with linked data and IPFS will be common to such tools, so we describe an intermediary tool called Dispatch that simplifies the interactions between these producers and consumers and the DAMF global store. Of course, Dispatch would be considered part of the *trusted code base*, along with IPFS and any utilities used to manipulate JSON data and cryptographic signatures. If this is problematic, Dispatch can be completely foregone in preference to native implementations.

The Dispatch tool is distributed as an executable `dispatch` with three subcommands: `publish`, `get`, and `lookup`. The `dispatch publish` command operates on one of a collection of standard input formats that contains local information corresponding to DAMF types. After syntactically validating this input, the `publish` command will construct and publish the global objects. Dispatch can also optionally interact with a specific storage service in order to make that object widely discoverable in the IPFS network. As an example, consider the following input for an *assertion* object, where newly created formulas and contexts are placed in the same file and are referred by local names such as `plus_comm`, and previously existing objects are referred by their `cid`s usig the `damf:` flag, such as the first value of `"dependencies"` (line 10) which refers to a *formula* object `cid`, as well as `"language"` and `"mode"` values which refer to existing *language* and *tool* objects respectively.

```
1  { "format": "assertion",
2    "agent": "localAgent",
3    "claim": {
4      "format": "annotated-production",
5      "annotation": ...,
6      "production": {
7        "mode": "damf:bafyreihnx2...",
8        "sequent": {
9          "conclusion": "plus_comm",
10         "dependencies": [ "damf:bafyreihw6g...", "plus_succ" ] } } },
11   "formulas": {
12     "plus_comm": {
13       "language": "damf:bafyreidyts...",
14       "content": ": forall M N K, nat K -> ...",
15       "context": ["plus"] },
16     "plus_succ": {
17       "language": "damf:bafyreidyts.....",
18       "content": ": forall M N K, ...",
19       "context": ["plus"] } },
20   "contexts": {
21     "plus": {
22       "language": "damf:bafyreidyts.....",
23       "content": [
24           "Kind nat type.", "Type z nat.", "Type s nat -> nat.",
25           "Define plus : nat -> nat -> prop by ...." ] } } }
```

This example is based on an output from our Abella-DAMF prover described below. A prover using Dispatch tool only needs to be able to produce and consume JSON objects with this structure, without needing to interface with IPFS directly. The value of `"agent"` (line 2) refers to an *agent profile* in Dispatch; each profile maps a user-readable name to a cryptographic key-pair, created separately using the `dispatch create-agent` command.

The `dispatch get` command takes a `cid` as an argument, fetches the IPLD `dag` (the full JSON object) referenced by it from the global store, validates the types of all constituent IPLD linked objects, verifies any signatures, and finally outputs a JSON object that is similar in structure to that accepted by `dispatch publish`.

The consumer will have access to all the necessary DAMF objects referenced by the root `cid` without needing to interact with the global store or structurally validating any objects. The only difference between the output of `dispatch get` and the input of `dispatch publish` is that the local names that appeared in the input will be replaced by `cids` (i.e., *global names*) in the output. Input and output formats corresponding to other global types are described further at the site mentioned in the introduction.[3]

The `dispatch lookup` command, as mentioned earlier, is the starting process that we consider in our implementation regarding the combination and analysis of DAMF assertions. Given a formula `cid` and a collection of assertion `cids`, the output of this command is a list of potential sets of (agent, mode/tool) pairs that correspond to combinations of assertions that would yield the target formula. Any remaining unmatched dependency is also outputed along with the (agent, mode/tool) pairs. In our current implementation, Dispatch exhaustively generates all possible ways of constructing the target formula. A direct improvement is to change this aspect of the tool to allow for a more interactive and incremental exploration of such dependencies. In addition, filtering through allow-lists would reduce the number of assertion combinations generated by this command.

3.3 Edge systems example: Abella

We have implemented a DAMF-aware branch of Abella [9] as an example of a system that interacts with assertions in DAMF with the help of Dispatch as a mediator. Abella was originally designed to test a particular approach to meta-theoretic reasoning using a new, proof-theoretically motivated mechanism for reasoning directly with bound variables (in particular, the ∇-quantifier [30] and a treatment of equality based on equivariant higher-order unification [26]). While the current implementation of Abella has succeeded with those meta-theoretic tasks [22,42], the prover has not grown much beyond that domain. Indeed, Abella has some (mis)features that make it a good test case for DAMF: (1) it has no awareness of the file system and it is easy to replace the backing store from local files to objects stored in IPFS; (2) it has a feature-poor proof language with nearly no support for proof automation and hence an underdeveloped formal mathematical libraries; and (3) it uses *relational* specifications as opposed to the more common *functional programming* specifications. Furthermore, the area of meta-theory that Abella treats declaratively is also an area many conventional proof systems do not deal well, in part, because of the need to encode and manipulate bindings [8,23]. Such conventional systems might be willing to delegate such meta-theoretic reasoning to Abella.

Ordinary Abella developments (in `.thm` files) support a kind of *import* mechanism which loads in marshaled results from a different run of Abella. We extend *import* with a new kind of statement: `Import "damf:bafyr..."` that refers to a collection of DAMF assertions (i.e., a DAMF collection object whose elements are assertions). Dispatch is used to fetch all the referenced objects from IPFS as explained in the previous subsection.

[3] `https://distributed-assertions.github.io/`

To appeal to an assertion, the elements of the context of the conclusion of the assertion are *merged* using their internal names with the ambient context of Abella where the assertion is appealed to. An Abella declaration in the context is *mergeable* if it has both the same internal name and an identical (up to λ-equivalence) definition; thus, type and term constants are merged if they have the same kinds or types (respectively), and (co-)definitions are merged if they have the same definitional clauses. This is done to keep the implementation simple and mostly unchanged from the standard (non-DAMF) Abella, which also only allows an `Import` declaration when the imported objects can be merged.

When the proof of a theorem is completed in Abella, a sequent object is constructed with the dependencies being all the DAMF lemmas appealed to in the proof, and the conclusion being the statement of the theorem (the formula) in the context of all its necessary declarations, computed using a dependency analysis. We use only the necessary declarations to allow such DAMF sequents to have the widest possible uses, since a DAMF assertion can only be used in Abella if the *entire* context of the conclusion can be merged.

Appendix B contains a full example of an Abella development that makes use of imported assertions from Abella, Coq, and λProlog. In this example, Coq and λProlog are not modified at all, and Abella is only minimally modified to use Dispatch to interact with DAMF assertions. The total amount of modifications to Abella to interface with Dispatch amounts to about 100 lines of code, most of which deals with (un)marshalling JSON. We expect that making tools DAMF-aware would require negligible effort.

4 Discussion: Design choices and alternatives

4.1 The role of formal proofs

Autarkic theorem provers often exploit the existence of proofs for several reasons. Obviously, the ability to check a fully detailed proof object in their own kernel, following the *De Bruijn criterion* [10], is central. But proofs can also be used for various other roles. For example, they sometimes contain constructive content that can be extracted as executable programs, and they can be used as guides during the development and maintenance of other proofs. Given their central role in many proof assistants, a great deal of effort has gone into the formalization, manipulation, and transformation of formal proof objects; see, for example, MMT [36], Logipedia [20], and foundational proof certificates [17]. As a concrete matter, proof objects can be included in the annotations of annotated productions in the global store of DAMF. Sequents are linked in productions by their `cid`s, so it is possible for the same sequent to have multiple proof objects contributed by different agents in separate assertions.

4.2 Potential benefits to mainstream systems

The fact that proof objects are not central to DAMF and the example presented in Section 3.3 might lead the reader to believe that the only beneficiaries of DAMF are new systems that want to leverage existing developments in mainstream systems. This belief is not necessarily true for two reasons. First, there are certain logical systems and formalization styles that are inordinately complicated or impossible to do in mainstream systems. Good examples are nominal sets [34], λ-tree syntax (a.k.a. *higher-order abstract syntax*) [2,23], generic judgments [30], and nominal abstraction [26]. It is conceivable that a mainstream prover can use DAMF to import a formalization such as the proof of soundness of Howe's method done in the setting of higher-order abstract syntax and contextual modal type theory [31], which is at present not available in a mainstream proof system such as Coq or Agda.

A second benefit to mainstream systems is to enable more trustworthy refactoring of their existing implementations. For example, modern autarkic provers routinely recheck large collections of proofs, often after every invocation of a new instance of the proof checker and certainly after every change in the version of the prover. As a result of needing to recheck such 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, making errors in the kernel more likely to occur. With DAMF, once a trustworthy but slow kernel—e.g., a certified implementation of a kernel [40]—checks a proof, it rarely needs to be rechecked. This can even lower the pressure for kernel implementations to chase performance with increasing, error-prone complexity. Furthermore, the immutable nature of IPFS objects makes DAMF assertions resistant to malicious subversion of the proper execution of a tool – see, for example, the discussion in [4] concerning attacks on Coq's `.vo` object files

4.3 Other use cases

While it is common to view tools that perform pure computations (such as functional program execution or proof search a la λProlog) as producing assertions without proofs, there are various well-known reasoning systems that have been used a lot without being either certified or certifying: for example, Twelf [33]. DAMF would enable Twelf-based assertions to be exported to agents willing to trust its type and totality checkers.

The relationship of DAMF to the following topics is discussed in greater detail in the technical report [3]: libraries as curation on top of the DAMF model of global objects; attacks in the adversarial environment of the web; and possible uses of this framework in settings (such as journalism) where the lack of formal proof means increasing the need to explicitly track trust.

5 Related work

The *semantic web* [13,14] was proposed to enrich the web with aspects of trust and would rely on concepts and technologies such as cryptography, taxonomies, ontologies, and inference rules. While the semantic web and DAMF both use cryptographic signatures and low-level web-based technologies, DAMF differs from the semantic web by focusing on objects rather than documents and using richer notions of logic and compositional reasoning.

Dedukti [7] is a dependently typed λ-calculus augmented with rewriting. Dedukti can be used to produce adapters (Section 2.3): in particular, proofs in a source system can be transformed to Dedukti proofs and then transformed back into formal proofs in a different system. For example, the Logipedia documentation mentions that "some 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, ..." [29]. As a by-product, Dedukti can be used to build correctness-preserving translations of assertions for DAMF.

TPTP [41] provides a number of standards for the concrete syntax of first-order and higher-order logic along with tools for parsing and printing files that adhere to such standards. Deploying those tools for the production of the kind of multilingual adaptors that we have described in sec. 2.3 is a natural next step for tool development within DAMF.

The recognition that distributing some aspects of proof environments goes back to at least the systems described by Sacerdoti Coen, et al. [6,18]. In such systems, integration was meant to work between "near-peer" systems: that is, between systems that are both based on rich logics such as higher-order logic or on typed λ-calculi based on the Curry-Howard correspondence. A prerequisite for successful integration in such systems is the ability to connect the semantics of formulas, types, universes, proofs, etc. The wide spread use of such integration approaches has been delayed since it has only been in recent years that efforts, such as Dedukti [7] and MMT [37,38], are making it possible to form the necessary deep and sophisticated ties between the semantics of these objects arising from different implementations. In contrast, DAMF allows the composition of different assertions without an apriori assumption that there is a formal semantics that relates them. Of course, correctness is a concern in many (most) situations: in those cases, Dedukti and MMT encodings can be used to translate assertions between two provers with precise correctness assurances. Often, however, the integration is of a more asymmetric kind. For example, when integrating a system that only performs integer operations or reasons only with integer inequalities (operations that are available in SMT systems) with a system based on higher-order logic, producing adapters based on sophisticated encodings might be completely unnecessary. The DAMF system similarly allows such integration.

6 Conclusion

We have described a Distributed Assertion Management Framework (DAMF) designed to share assertions between agents while tracking dependencies with canonical content ids (`cids`). This framework endows assertions with reliable provenance using public key cryptography and distributes them globally using the IPFS network. We have given an example of using DAMF to import a Coq lemma into Abella. The biggest challenge for future work is to adapt existing work on language translation and proof translation (in, e.g., Dedukti) to create or derive adapters automatically. Another important matter for future consideration is whether to persist compositions (i.e., Compose-derivations, cf. sec. 2.4) to DAMF, which can serve as hints for post hoc investigations.

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, 9th International Conference, DEON 2008, Luxembourg, Luxembourg, July 15-18, 2008. Proceedings. LNCS, vol. 5076, pp. 96–109. Springer (2008). https://doi.org/10.1007/978-3-540-70525-3_9
2. Abel, A., Allais, G., Hameer, A., Momigliano, A., Pientka, B., Schaefer, S., Stark, K.: POPLMark reloaded: Mechanizing proofs by logical relations. Journal of Functional Programming **29** (2019). https://doi.org/10.1017/S0956796819000170
3. Al Wardani, F., Chaudhuri, K., Miller, D.: Distributing and trusting proof checking: a preliminary report. Tech. rep., Inria Saclay (2022), https://hal.inria.fr/hal-03909741
4. ANSSI, F.N.C.A.: Requirements on the use of Coq in the context of common criteria evaluations. URL (Dec 2021), v1.1
5. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.P., Shao, Z. (eds.) Certified Programs and Proofs (CPP 2011). LNCS, vol. 7086, pp. 135–150 (2011), http://hal.inria.fr/docs/00/63/91/30/PDF/cpp11.pdf
6. Asperti, A., Padovani, L., Coen, C.S., Guidi, F., Schena, I.: Mathematical knowledge management in helm. Ann. Math. Artif. Intell **38**(1-3), 27–46 (2003)
7. 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), http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf
8. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In: Theorem Proving in Higher Order Logics: 18th International Conference. pp. 50–65. No. 3603 in LNCS, Springer (2005). https://doi.org/10.1007/11541868_4
9. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: A system for reasoning about relational specifications. Journal of Formalized Reasoning **7**(2), 1–89 (2014). https://doi.org/10.6092/issn.1972-5787/4650
10. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Transactions A of the Royal Society **363**(1835), 2351–2375 (Oct 2005)
11. Barendregt, H., Barendsen, E.: Autarkic computations in formal proofs. J. of Automated Reasoning **28**(3), 321–336 (2002). https://doi.org/10.1023/A:1015761529444
12. Benet, J.: IPFS-content addressed, versioned, P2P file system (2014). https://doi.org/10.48550/arxiv.1407.3561
13. Berners-Lee, T.: Semantic Web road map. Tech. rep., W3C Design Issues (1998), http://www.w3.org/DesignIssues/Semantic.html
14. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American Magazine (May 2001)
15. Chaudhuri, K., Doligez, D., Merz, S., Lamport, L.: The TLA+ proof system: Building a heterogeneous verification platform. In: Cavalcanti, A., Déharbe, D., Gaudel, M.C., Woodcock, J. (eds.) Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC). LNCS, vol. 6256, p. 44. Springer, Natal, Rio Grande do Norte, Brazil (Sep 2010). https://doi.org/10.1007/978-3-642-14808-8_3, http://hal.archives-ouvertes.fr/inria-00521886/en/

16. Chaudhuri, K., Gérard, U., Miller, D.: Computation-as-deduction in Abella: work in progress. In: 13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. Oxford, United Kingdom (Jul 2018), `https://hal.inria.fr/hal-01806154`
17. Chihani, Z., Miller, D., Renaud, F.: A semantic framework for proof evidence. J. of Automated Reasoning **59**(3), 287–330 (2017). `https://doi.org/10.1007/s10817-016-9380-6`
18. Coen, C.S.: Mathematical libraries as proof assistant environments. In: International Conference on Mathematical Knowledge Management (MKM), LNCS. vol. 3 (2004)
19. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013. LNCS, vol. 7737, pp. 275–294. Springer (2013). `https://doi.org/10.1007/978-3-642-35873-9_18`
20. Dowek, G., Thiré, F.: Logipedia: a multi-system encyclopedia of formal proofs. `http://www.lsv.fr/~dowek/Publi/logipedia.pdf` (2019)
21. 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 - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. LNCS, vol. 9450, pp. 460–468. Springer (2015). `https://doi.org/10.1007/978-3-662-48899-7_32`, `http://dx.doi.org/10.1007/978-3-662-48899-7_32`
22. Felty, A.P., Momigliano, A., Pientka, B.: 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). `https://doi.org/10.1007/s10817-015-9327-3`
23. Felty, A.P., Momigliano, A., Pientka, B.: Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. Mathematical Structures in Computer Science pp. 1507–1540 (2017). `https://doi.org/10.1017/S0960129517000093`
24. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming, ESOP 2013, Rome, Italy. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
25. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.F.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 3920, pp. 167–181. Springer (2006). `https://doi.org/10.1007/11691372_11`
26. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. Information and Computation **209**(1), 48–73 (2011). `https://doi.org/10.1016/j.ic.2010.09.004`
27. Harrison, J.: The HOL Light tutorial (2017), `https://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf`
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
29. Logipedia in a nutshell. `http://logipedia.inria.fr/about/about.php` (2022)
30. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Trans. on Computational Logic **6**(4), 749–783 (Oct 2005). `https://doi.org/10.1145/1094622.1094628`

31. Momigliano, A., Pientka, B., Thibodeau, D.: A case-study in programming coinductive proofs: Howe's method. Submitted (2017)
32. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the Combined KEAPPA - IWIL Workshops. CEUR Workshop Proceedings, vol. 418, pp. 123–132. CEUR-WS.org (2008), `http://ceur-ws.org/Vol-418/paper10.pdf`
33. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) 16th Conf. on Automated Deduction (CADE). pp. 202–206. No. 1632 in LNAI, Springer, Trento (1999). `https://doi.org/10.1007/3-540-48660-7_14`
34. Pitts, A.M.: Nominal logic, A first order theory of names and binding. Information and Computation **186**(2), 165–193 (2003)
35. Qi, X., Gacek, A., Holte, S., Nadathur, G., Snow, Z.: The Teyjus system – version 2 (2015), `http://teyjus.cs.umn.edu/`, `http://teyjus.cs.umn.edu/`
36. Rabe, F.: The future of logic: Foundation-independence. Logica Universalis **10**(1), 1–20 (2016)
37. Rabe, F.: How to identify, translate and combine logics? J. of Logic and Computation **27**(6), 1753–1798 (2017)
38. Rabe, F.: The MMT Language and System. `https://uniformal.github.io/` (2022)
39. Rushby, J.M.: An evidential tool bus. In: Lau, K., Banach, R. (eds.) Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings. LNCS, vol. 3785, pp. 36–36. Springer (2005). `https://doi.org/10.1007/11576280_3`
40. Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The metacoq project. J. Autom. Reason. **64**(5), 947–999 (2020). `https://doi.org/10.1007/s10817-019-09540-0`
41. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning **43**(4), 337–362 (2009). `https://doi.org/10.1007/s10817-009-9143-8`
42. Tiu, A.: On the role of names in reasoning about λ-tree syntax specifications. In: Abel, A., Urban, C. (eds.) International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008). pp. 32–46 (2008)
43. The Twelf project (2016), `http://twelf.org/`

A Global types, or DAMF types

A.1 Types schema

The global types, as identified in our implementation, are illustrated below (in `TypeScript` syntax). A type of the form `(X)Link` is an `ipldLink` which has the form `{ "/": cid }`, where `cid` is a `string`. This is done only to illustrate the kind of object the link is referencing. Similarly, while both `publicKey` and `digitalSignature` are, in fact, strings, we use two different names to indicate their different roles. Finally, we write `damfLink` to denote that the defined generic `collection` type refers to any type of the defined DAMF types. Notice that in a `formula` object, the `context` is a list of `ipldLinks` to `context` objects: thus, previously stored contexts can be accumulated to form larger contexts. In this example, we have provided some `mode` values for illustration purposes: the actual collection of useful modes will certainly grow as a larger community uses this framework.

```
type language = {
  "format": "language",
  "content": ipldLink }

type context = {
  "format": "context",
  "language": languageLink,
  "content": ipldLink }

type formula = {
  "format": "formula",
  "language": languageLink,
  "content": ipldLink,
  "context": contextLink[] }

type sequent = {
  "format": "sequent",
  "dependencies": formulaLink[],
  "conclusion": formulaLink }

type tool = {
  "format": "tool",
  "content": ipldLink }

type production = {
  "format": "production",
  "sequent": sequentLink,
  "mode": null | toolLink | "axiom" | "conjecture" }

type assertion = {
  "format": "assertion",
  "claim": productionLink | annotatedProductionLink
  "agent": publicKey,
  "signature": digitalSignature }

type annotatedContext = {
  "format": "annotated-context",
  "context": contextLink,
  "annotation": ipldLink }

type annotatedFormula = {
  "format": "annotated-formula",
  "formula": formulaLink,
  "annotation": ipldLink }

type annotatedSequent = {
  "format": "annotated-sequent",
  "sequent": sequentLink,
  "annotation": ipldLink }

type annotatedProduction =  {
  "format": "annotated-production",
  "production": productionLink,
  "annotation": ipldLink }

type collection = {
  "format": "collection",
  "name": string,
  "elements": damfLink[] }
```

A.2 Examples

The following are examples of published objects of *assertion*, *annotated-production*, *production*, *sequent*, *formula*, and *context* types respectively. The full objects can be viewed using the IPLD Explorer.

Assertion

```
// bafyreiek2t75whn7gi6ygrymegguescqi4iu...
{ "format": "assertion",
  "claim": {
    "/": "bafyreibvtxzqhvht5rfxpw3rkgx3xliotvjs..." },
  "agent": "-----BEGIN PUBLIC KEY-----\nMFIwEA...",
  "signature": "3040021e10db76a6606d7a813747849028c79e..." }
```

Annotated production

```
// bafyreibvtxzqhvht5rfxpw3rkgx3xliotvjs... [OR]
// bafyreiek2t75whn7gi6ygrymegguescqi4iu.../claim
{ "format": "annotated-production",
  "production": {
    "/": "bafyreihlji3p7py5sbfklmutnoclqjs4uql3..." },
  "annotation": {
    "/": "bafyreieonawrhw3czj27bcdxo6lpydazmu22..." } }
```

Production

```
// bafyreihlji3p7py5sbfklmutnoclqjs4uql3... [OR]
// bafyreibvtxzqhvht5rfxpw3rkgx3xliotvjs.../production
{ "format": "production",
  "mode": { "/": "bafyreihnx2hp5xhgozl5mcu3ixh3xlebsqhb..." },
  "sequent": { "/": "bafyreigw3o7kbd65zl43f6ksinyljdzbk3nj..." } }
```

Sequent

```
// bafyreigw3o7kbd65zl43f6ksinyljdzbk3nj... [OR]
// bafyreihlji3p7py5sbfklmutnoclqjs4uql3.../sequent
{ "format": "sequent",
  "conclusion": {
    "/": "bafyreieysj5wtzzi6jx64octfttrpaij5k7q..." },
  "dependencies": [
    { "/": "bafyreihw6ggod5k5nvrfs7a3prtvjoqd6t3u..." },
    { "/": "bafyreiawv2xo62nkwoto2cw77gwbcpef3ete..." } ] }
```

Formula

```
// bafyreieysj5wtzzi6jx64octfttrpaij5k7q... [OR]
// bafyreigw3o7kbd65zl43f6ksinyljdzbk3nj.../conclusion
{ "format": "formula",
  "language": { "/": "bafyreidytsnnzmr7mcmd4abvy4ufp7rwh..." },
  "content": {
    "/": "bafyreigytwz7qnrssba2vdqsle765kgyuvnp..." },
  "context": [
    { "/": "bafyreifitbc5ywrbxrvnykdohxxc5d6yvdsx..." } ] }
```

Context

```
// bafyreifitbc5ywrbxrvnykdohxxc5d6yvdsx... [OR]
// bafyreieysj5wtzzi6jx64octfttrpaij5k7q.../context/0
{ "format": "context",
  "content": { "/": "bafyreia4m2rkjud4jdregjtajo3v4yws5..."},
  "language": { "/": "bafyreidytsnnzmr7mcmd4abvy4ufp7rwh..." } }
```

B A Complete Example

This section presents a complete example of proving the following theorem in Abella using external lemmas from Coq and λProlog:

> 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 sizeable 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 solutions for $\texttt{fib}(n) = X$ for $n \in \{0, 1, \ldots, 12\}$. 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.

Further details on this example, including the various input files and the precise tools and commands to run to verify the development, can be found in the following location:

`https://distributed-assertions.github.io/example-walkthrough/`.

B.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 namespace 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 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. 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
; lt z (s X)
; lt (s X) (s Y) := lt X Y.

% plus X Y Z ≡ Z = X + Y
Define plus : nat -> nat -> nat -> prop by
; plus z X X
; plus (s X) Y (s Z) := plus X Y Z.

% times X Y Z ≡ Z = X × Y
Define times : nat -> nat -> nat -> prop by
; times z X z
; times (s X) Y Z :=
    exists U, times X Y U /\ plus U Y Z.
```

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.
```

B.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 [35] or ELPI [21]. 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.
end.
```

```
%% 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.
end.
```

With this definition, we can ask a λProlog engine to solve `fib` goals where the first argument is ground. For example:

```
[fib] ?- fib (s (s (s z))) X.

The answer substitution:
X = s (s z)

More solutions (y/n)? y

no (more) solutions
```

We can also, of course, check that a given ground predicate is indeed derivable.

```
[fib] ?- fib (s (s (s (s (s z))))) (s (s (s (s (s z))))).

yes
```

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:bafyreigk...",
      "sequent": {
        "conclusion": "fib5",
        "dependencies": [] } } },
  "formulas": {
    "fib5": {
      "language": "damf:bafyreice...",
      "content": "fib (s (s (s (s (s z))))) ...",
      "context": ["fib"] } },
  "contexts": {
    "fib": {
      "language": "damf:bafyreice...",
      "content": [
          – contents of fib.sig as a string –  ,
21            – contents of fib.mod as a string –
22        ] } } }
```

The "language" values in lines 13 and 18 are understood to be canonical references to a DAMF object referencing the language of λProlog. Similarly, the "mode" value in line 7 is a canonical reference to a DAMF object describing the Teyjus implementation. The "agent" value in line 2 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.

B.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 full development in Coq v. 8.16.1 is shown below.

```
Require Import Arith Lia.

Fixpoint fib (n : nat) :=
  match n with
  | 0 => 0
  | S j =>
      match j with
      | 0 => 1
      | S k => fib j + fib k
      end
  end.

Ltac liarw F :=
  let h := fresh "H" in
  assert (h : F) by (simpl ; lia) ; rewrite h in * ; clear h.

Theorem fib_square_lemma : forall n, 2 * n + 27 <= fib (n + 12).
induction n.
- simpl ; lia.
- liarw (n + 12 = S (n + 11)).
  liarw (S n + 12 = S (S (n + 11))).
  assert (H1 : 2 <= fib (n + 11)).
    clear IHn ; induction n ; [simpl ; lia | ].
    liarw (n + 11 = S (n + 10)).
    liarw (S n + 11 = S (S (n + 10))).
    assert (H : fib (S (S (n + 10))) = fib (S (n + 10)) + fib (n + 10))
      by auto.
    lia.
  assert (fib (S (S (n + 11))) = fib (S (n + 11)) + fib (n + 11))
    by auto.
  lia.
Qed.
```

```
33
34 Theorem fib_square_above : forall n, 13 <= n -> n ^ 2 < fib n.
35 intros n Hle ; pose (k := n - 13) ; liarw (n = k + 13) ; clear Hle.
36 induction k.
37 - simpl ; lia.
38 - liarw (k + 13 = S (k + 12)).
39   liarw (S k + 13 = S (S (k + 12))).
40   assert (fib (S (S (k + 12))) = fib (S (k + 12)) + fib (k + 12))
41     by auto.
42   liarw (S (S (k + 12)) ^ 2 = S (k + 12) ^ 2 + 2 * k + 27).
43   specialize (fib_square_lemma k).
44   lia.
45 Qed.
```

Note the various appeals to the linear integer arithmetic solver `lia` in the proofs, either used directly or via a defined Ltac `liarw` defined in lines 13–15.

If Coq were to add publishing support for DAMF, the sequent generated for the `fib_square_lemma` theorem above may look something like this:

```
1  { "format": "assertion",
2    "agent": "exampleAgent",
3    "claim": {
4      "format": "annotated-production",
5      "annotation": {"name": "fib_square_above"},
6      "production": {
7        "mode": "damf:bafyreium...",
8        "sequent": {
9          "conclusion": "fib_square_above",
10         "dependencies": [] } } },
11   "formulas": {
12     "fib_square_above": {
13       "language": "damf:bafyreikf...",
14       "content": "forall n, 13 <= n -> n ^ 2 < fib n",
15       "context": ["fib_square_above!ctx"] } },
16   "contexts": {
17     "fib_square_above!ctx": {
18       "language": "damf:bafyreikf...",
19       "content": [
20         "Require Import Arith.",
21         "Fixpoint fib (n : nat) := match ··· end."
22       ] } } }
```

Here, the `"mode"` on line 7 represents a DAMF object that describes the Coq (v. 8.16.1) tool, while the `"language"` field in lines 13 and 18 describe the Coq language. Note that this is in the input format intended for `dispatch publish`.

B.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 B.2 is given the `cid bafyrei4j`.... Here is how we would import it in Abella:

```
37  %% FibExample.thm continuing...
38  Import "damf:bafyrei4j..." as
39  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",
2     "agent": "exampleAgent",
3     "claim": {
4        "format": "annotated-production",
5        "annotation": {"name": "fib5"},
6        "production": {
7          "mode": null,
8          "sequent": {
9            "conclusion": "fib5",
10           "dependencies": [
11             "damf:bafyrei4j.../claim/sequent/conclusion" ] } } },
12    "formulas": {
13      "fib5": {
14        "language": "damf:bafyreig8...",
15        "content": "fib (s (s (s (s (s z))))) ...",
16        "context": ["fib5!context"] } },
17    "contexts": {
18      "fib5!context": {
19        "language": "damf:bafyreig8...",
20        "content": [
21          "Kind nat type.", "Type z nat.", "Type s nat -> nat.",
22          "Define fib : nat -> nat -> prop by ...." ] } } }
```

Lines 14 and 19 above are references to a DAMF object describing the Abella language. In line 7, 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 11, 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 38 of `FibExample.thm`. As explained in Section 3.1, 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 39 of `FibExcample.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:

```
40  %% FibExample.thm continuing...
41  Import "damf:bafyreiyv..." as
42  Theorem fib_square_above : forall n, nat n -> le (s^13 z) n ->
43    forall u, times n n u ->
44      forall v, fib n v -> lt u v.
```

The `cid` `damf:bafyreiyv`... on line 41 is that of the assertion corresponding to the theorem `fib_square_above` in Coq. The imorted assertion is rewritten as shown in lines 42–44. As before with λProlog, the assertion corresponding to this assertion, with the `"mode"` of production set to `null`, can be easily generated and published by Abella.

B.5 Assembling the final theorem in Abella

Given these external lemmas from λProlog and Coq, the final desired theorem is fairly straightforward to assemble; the essential cases are shown below.

```
45  %% FibExample.thm continuing...
46
47  % Some more easy lemmas
48  Theorem fib_deterministic : forall x y z, fib x y -> fib x z -> y = z.
49  ... % proof elided
50  Theorem lt_irreflexive : forall x, nat x -> lt x x -> false.
51  ... % proof elided
52  Theorem times_result_nat : forall m n k, nat m -> nat n ->
53    times m n k -> nat k.
54  ... % proof elided
55
56  %%% main theorem
57
58  Theorem fib_squares : forall x x2, nat x -> times x x x2  ->
59    (fib x x2 <-> x = z \/ x = s z \/ x = s^12 z).
60  intros Hnat Hsquare. split.
61
62  %% ->
63  intro Hfib.
64  Hcs: assert x = z \/ x = s z \/ ··· \/ x = s^12 z \/ leq (s^13 z) x.
65    case Hnat. search.
66    case Hnat. search.
67    ··· /* repeat 12 times in total */
68    search. % leq (s^13 z) (s^13 x).
69  case Hcs. search. % case of x = z
```

```
case Hcs. search. % case of x = s z
case Hcs.
  % case of x = s (s z)
  Ha : apply fib2. % fib (s (s z)) (s z)
  case Hsquare. ··· % etc. to instantiate x2 to 4
  apply fib_deterministic to Ha Hfib. % contradiction: 1 ≠ 4
··· /* so on for 3, ..., 11 */
case Hcs. search. % case of x = s^12 z
% finally, case of x = s^13 z
H : apply fib_square_above to Hnat Hcs.
H : apply *H to Htimes Hfib. % lt x2 x2
Hnat' : apply times_result_nat to Hnat Hnat Htimes.
apply lt_irreflexive to Hnat' H. % obtains a contradiction

%% <-
intro Hcs.
case Hcs. search. % case of x = z
case Hcs. search. % case of x = (s z)
apply fib12. search. % case of x = s^12 z
```

The uses of the external lemmas are highlighted in red. There are some other minor lemmas (e.g., `times_result_is_nat`, `fib_deterministic`) which are easily proved within Abella.