# Formal Reasoning using Distributed Assertions

Farah Al Wardani[0000−0003−1520−7090],
Kaustuv Chaudhuri[0000−0003−2938−547X], and Dale Miller[0000−0003−0274−4954]

Inria Saclay & LIX, Institut Polytechnique Paris

**Abstract.** When a formal proof is checked by a proof system, we can say that the kernel *asserts* that the formula is a theorem in a particular logic. We describe an implemented framework in which such an assertion can be made global so that any other proof assistant willing to trust the generator of the assertion can use that assertion without rechecking any associated formal proof. In this framework, we propose moving beyond autarkic proof checkers—i.e., self-sufficient systems that trust proofs only when checked by their kernel—to an explicitly non-autarkic setting. In particular, this framework must explicitly track which proof tools and their operators are trusted when asserting that a formula is a theorem. We illustrate how our framework can be integrated into existing proof systems by making minor changes to the input and output subsystems of the prover. Our framework achieves a high level of distributivity using off-the-shelf technologies: IPFS, IPLD, and public key cryptography.

## 1 Introduction

Nearly every formal reasoning system today uses an *autarkic trust model*:[1] the system only trusts its own kernel. If a formal result is to be transferred from one autarkic system to another, either (a) formal proof certificates have to be *translated* from the source system to the target system, or (b) the source system must be re-implemented as a *certified* procedure in the target system [5,23]. Both transferal alternatives would be challenging for just two systems, even for a pair such as Coq and Matita that have nearly identical foundations. The challenge is considerably worse for a large number of systems, with varying degrees of syntactic, semantic, and foundational compatibilities, that may nevertheless want to share formal results. The challenge is worse still because of its *incumbency bias*: any new autarkic system that wants to reuse an existing result established in a different system must first translate from or re-implement the other system. The kernel of the new system must therefore be general enough to enable the transferal and performant enough to transfer non-trivial results.

In this paper we advocate for a different trust model for formal reasoning systems that is intentionally non-autarkic: we wish to allow for formal objects of knowledge to be exchanged between systems *without* needing to exchange proofs. Our proposed model allows a collection of *agents* to communicate *assertions*,

[1] In [10], the adjective *autarkic* was applied to computational components of a proof checker but not to an entire proof checker.

which are formal facts that may (or may not) be constructed using specific *tools* such as proof checkers. Any user can *retrieve* and use any assertion they understand, and this external import will be explicitly marked as a *dependency* if they choose in turn to *publish* the assertions they build with such imports. We say that this trust model is *distributed* because it enjoys the following properties.

- *Decentralized*: a global notion of *truth* is not imposed on all users by the means of a privileged logic, system, or software. Users can combine components from different sources if they *believe* the combination to be meaningful.
- *Reliable*: assertions have an *irrepudiable* provenance, and these objects of knowledge are *immutably* and *eternally* available, even with intermittent infrastructure and nefarious users or tools.
- *Egalitarian*: the barrier to entry is low for both users and systems.

This trust model can also be seen as a generalization of the autarkic model, so existing work that has already been done with currently mainstream autarkic systems can be readily incorporated. In fact, our model makes explicit certain aspects of the autarkic trust model that are often unstated, such as the information about *which* tools were used by *whom* in order to make a formal assertion.

Concretely, we propose a framework for representing formal objects in the distributed trust model, which we call the *Distributed Assertion Management Framework* (DAMF). This framework provides a small number of JSON schemas for representing formulas, assertions, dependencies, translations, etc. without any up front commitment to a formal syntax or any particular semantics. These JSON objects are then added to a *global store* in terms of the *InterPlanetary File System* (IPFS) [11] 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 the DAMF objects transparently using a variety of technologies whose precise details are not relevant for this paper.

Because dependencies are explicitly tracked in DAMF assertions, it is possible for any user to perform a *trust analysis* of a given conclusion with respect to a collection of assertions. The most important kind of analysis is *lookup*, which 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. Such lookup queries can be further controlled and filtered using accept/deny lists of agents and tools. Lookup queries can also form the basis of further *investigations*; for example, if a formula is found to be a non-theorem, then we can lookup compositions of assertions in DAMF that yield that formula to find agents whose trust parameters may need to be modified.

This paper is accompanied by two concrete implementations that illustrate the 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 implements the core *lookup* queries that enable further trust analysis.

Second, we implement a version of the Abella interactive theorem prover [8] 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.

In the next section, we describe the abstract design of DAMF and its underlying logic of assertions, which forms the basis of the trust analysis 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 the DAMF

### 2.1 Non-autarky: What to trust?

To say that a proof system departs from the autarkic trust model is equivalent to saying that the proof system does not limit its trust to its own kernel. What else can such a system trust? One answer might be that it can trust *executions* of *external tools* such as special purpose provers, decision procedures, static analyzers, model checkers, SAT and SMT solvers, etc. Such tools may be only generally described, or they may be precisely identified by (a cryptographic hash of) their executable binaries, for example. The TLAPS [14] proof system and the Why3 [22] static checking tool are examples of systems that rely on external tools. Shankar [36] has argued that *little engines of proof* need to be aggregated to yield the next generations of proof systems. The *evidential tool bus* [17,35] was proposed as a means to integrate such specialized inference systems into a unified formal system.

One issue not addressed with this enlarged view of trust is that the external tools may not necessarily be easy to re-execute repeatedly. The HOL Light system [25], for example, re-checks its entire standard library every time it is started,

taking on the order of minutes. Relying on a result that can be established externally in HOL Light thus presents a pragmatic engineering challenge: when, and how often, to run the external tool? Indeed, there is a more natural question to ask about trusted external tools: if they are successfully run once to establish a particular result, why is there a need to run them ever again?

Now, the external tool itself cannot reliably certify—at least, in a distributed and decentralized manner—that it has successfully performed a given execution. After all, this is the reason why trustworthy tools are built around independently verifiable proof evidence. Thus, it seems unavoidable that if we want to avoid repeated re-execution, our notion of trust must apply to the *agents* that ran the tools rather than the tools themselves. An agent is conceptually an entity that can execute a collection of tools in order to establish a formal result and cryptographically sign that result (together with any metadata necessary about the tools and processes involved). An agent can be identified using a public/private key-pair. An individual user may control many such key-pairs; we call these the *agent profiles* of the user. Agents are the sole carriers of *trust* in DAMF: an agent that signs an incorrect assertion will risk being evicted from allow lists. A malicious agent who tries to subvert the proper execution of a tool—see, for example, the discussion in [4] concerning attacks on Coq's `.vo` object files—will similarly find that publishing such manipulated results will only serve to damage their reputations.

### 2.2 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; this design choice is discussed further in Section 4.1. To determine that the input is well-formed, the source proof system may need further information about the symbols—predicates, function symbols, types, notations, etc.—that are used in the formula. This 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 $(\Sigma, F)_L$ where $\Sigma$ denotes a context, $F$ denotes a formula, and $L$ denotes a language, all of which may conceptually be thought of as strings. 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 $\Sigma$ and $F$; DAMF compares languages just by their identifiers. We will use the schematic variable $N$ to range over such formula objects. Note that formula objects are considered to be *closed*, i.e., every symbol that is used in the formula is either introduced by the context or is a standard symbol of the language.

### 2.3 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. We will use the schematic variable $S$ to range over sequents, and $\Gamma$ to range over ordered lists of formula objects. In a sequent $\Gamma \vdash N$, we say that $N$ is the *conclusion* and $\Gamma$ 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 above 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 `lem` $\vdash$ `ex_coq` (where we elide the context and the language).

Let the schematic variable $K$ 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 example of an *assertion*. In a proof system based on the non-autarkic trust model of DAMF, 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.

The *says* predicate is implemented in DAMF using public-key cryptography. In Section 3.1, the pairing of a sequent and a *mode*, called *production*, is what is actually signed by an agent. This allows for explicitly stating in which *mode* the assertion is intended, as a *conjecture* for example. The *mode* could also be a specific *tool*, which allows tracking what *tool* was used in a *production*, which is important to mitigate the damage for an agent's reputation in a scenario where the agent unknowingly uses a problematic tool.

### 2.4 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 generally 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 sequent 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

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

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

### 2.5 The logic of assertions, queries

The logic of assertions can be defined fully using Horn clauses, so its notion of provability can be seen as both classical and intuitionistic. Indeed, there is only one rule of inference that implements a cut-like rule for sequents, Compose.

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

Note that while signed assertions involving *says* persisted in the DAMF global store, it is not a one-to-one match. The conclusion of the Compose rule may, in particular, not be a sequent that has been explicitly signed by the agent $K$ even if

[2] This kind of relational representation of functional computation is described in [15].

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:

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

Such rules are neither syntactically well-formed nor desirable for our purposes. We use here a very weak access control logic. For a survey of such logics, see [1].

A database of assertions can be queried by posing an endsequent of the form $K$ *says* $(\vdash N)$ and building a derivation using instances of the COMPOSE rule whose leaves are signed assertions. Note that if COMPOSE is the only rule of inference, all the agents in the derivation will be identical; this is simply the autarkic trust model. To incorporate non-autarky, we will need to extend the inference system with the additional rule,

$$\frac{K_1 \text{ says } S}{K_2 \text{ says } S} \text{ TRUST } [K_1 \mapsto K_2].$$

This rule is applicable only when $K_1$ is in the user-specified *allow list* for the trust parameters of $K_2$ (i.e., $K_1$ *speaks for* $K_2$).

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:

$$\frac{K \text{ says } (\Gamma \vdash N_{A\to B}) \qquad K \text{ says } (\Gamma \vdash N_A)}{K \text{ says } (\Gamma \vdash N_B)} \text{ MP}.$$

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 *adapter 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 adapter agent $K_{\text{MP}}$ whose sole function is to generate assertions such as $K_{\text{MP}}$ *says* $(N_{A\to B}, N_A \vdash N_B)$ that correspond to applications of the MP rule. Of course, $K_{\text{MP}}$ will need to be in the *allow list* for any agent wanting to use this adapter.

## 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 Dispatch tool that we describe in the next subsection is then only one participant of many possible users of DAMF. 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. IPFS and IPLD provide us with the technical mechanisms needed to create and explore content-addressed linked data in a distributed network.

The structures we may want to specify in DAMF are built by composing several elements; for instance, an *assertion* contains a *sequent* structure, which in turn uses *formulas* that themselves contain *contexts*. In DAMF, we make the design choice to treat all such structures as *first class* objects stored in IPFS, and use the linked data representation of IPLD to refer to the constituent parts by means of their content ids (`cid`s). Because the `cid`s are cryptographically strong hashes of the objects, comparing two objects structurally is reduced to comparing their `cid`s as strings. Different assertions, created by distinct agents, can nevertheless refer to the same formula object in DAMF by simply reusing the `cid`; IPFS transparently handles the same object being added to the network at different nodes. Indeed, this kind of sharing is necessary for assertions to be composable using the COMPOSE rule described in Section 2.5, which has two premises that both refer to a common formula object. Giving formulas standalone `cid`s also makes the task of finding assertions that refer to the same formula more straightforward.

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: `"context"`, `"formula"`, etc. The remainder of these structures is as follows (Appendix A contains the full definitions):

- *Context*: contains a `language` field, which is an IPLD link to a *language* object described later, 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. Note that it is possible for the context and formula objects to have different languages, but this is highly unlikely to arise in practice. The contexts are made independent simply to allow for the possibility for multiple formula objects to share the same context, so a consumer of many formulas with the same context can potentially factor out the context processing.
- *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 sequent objects with a `mode` field denoting a *mode of production* of a sequent, described in more detail below.
- *Assertion*: a field `claim` mapped to an IPLD link to a production for example, 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.

A *production* is a sequent together with a `mode`, and it is productions and not sequents that are formally signed to make assertions. This `mode` field can be `null`, which is equivalent to the agent asserting the sequent directly with no further justification. The `mode` can be `"axiom"`, which signals that the sequent is not expected to be matched with a proof, i.e., that it is allowable as a leaf premise of a derivation in the assertion logic outlined in Section 2.5. The `mode` can also be `"conjecture"`, which signals that the agent does (did) not have a proof of the sequent but a proof can possibly be found. Finally, the `mode` can be an IPLD link to a *tool* object, which is an object in IPFS that contains the description of a tool; for instance, it can contain a link to its source code, a hash of its executable binary, etc. Like with languages, we compare tools by comparing the `cid`s of these tool objects. Other values of `mode` may be added in the future.

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 links to the core object and to any additional metadata. Generally speaking, annotated objects will not be referred to directly by other core objects; the sole exception is an assertion object that can refer to an annotated production object where the annotation carries, at minimum, at least one external name for the assertion. When the assertion is imported as an external dependency, it is this annotation that is used internally by the proof system to name the dependency. Of course, if a prover is capable of using a `cid` as the name of a lemma, then such annotations are unnecessary.

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

There are three processes in DAMF: *production*, *consumption*, and *trust analysis*. In a *production* process, DAMF objects are constructed, published, and stored based on local information. Each such object is publishable on its own. For example, one can decide to publish a production object with a `"conjecture"` mode, and other agents can try to prove and publish assertion objects linking to productions of the exact same sequent of the initial production with potentially different modes. In a *consumption* process, the language tag of the relevant formula object is first examined to see if the object can potentially be consumed (parsed). Finally, *trust analysis* starts from a formula `cid` (the desired goal) and a database of assertion `cid`s and searches for derivations in the assertion logic (Section 2.5) that have the formula as the endsequent. This process can then further analyze the agents (and modes) of all the assertions that constitute the assertion logic derivation.

Individual producers and consumers, such as theorem provers, can choose to implement all three 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 describe specified global types that allows for inlined objects in addition to IPLD linked objects. After syntactically validating this input, this command publishes each DAMF object individually and replaces internal links by IPLD links. Dispatch can also optionally interact with a specific storage service in order to make that object widely discoverable in the IPFS network. Finally, root `cid` of the object is returned. As an example, consider the following input format for an *assertion* object, where the formulas and contexts are linked by names such as `plus_comm`, etc.

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

This example is based on an output from our modified Abella prover (described in more detail in appendix B). A prover using the Dispatch tool only needs to be able to produce a JSON file with this structure.

In this input to `dispatch publish`, the suffix `damf:` is used as a Dispatch-specific way to indicate a DAMF object that is already present in the global store. For example, the first value of `"dependencies"` (line 12) refers to a *formula*

object `cid`. When locally created *contexts* or *formulas* are to be published, the local names used in this input serve as local references to the corresponding objects, which will be published and IPLD-linked by Dispatch. Similarly, local names can be used for `"language"`, and `"mode"` as references to existing *language* and *tool* `cid`s, which exist in Dispatch's configuration and are initialized with the `dispatch create-language` or `dispatch create-tool` subcommands. Likewise, the value of `"agent"` refers to an *agent profile*, which in Dispatch is a cryptographic key-pair, created separately using the `dispatch create-agent` subcommand. Input formats corresponding to other global types can be constructed similarly and are described further at the site mentioned in the introduction.[3]

The `dispatch get` subcommand takes a `cid` as an argument, fetches the IPLD `dag` (the full JSON object) referenced by it from the global store, 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 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 `cid`s (i.e., *global names*) in the output.

The `dispatch lookup` command performs a key step in trust analysis. Given a formula `cid` and a collection of assertion `cid`s, the output of this subcommand is a list of potential sets of (agent, mode/tool) pairs and axioms that would yield the indicated formula in an endsequent of the assertion logic derivation. In our current implementation, Dispatch exhaustively generates all possible ways of constructing the target formula. We are planning to change this aspect of the tool to allow for a more interactive and incremental exploration of such dependencies.

### 3.3 Edge systems example: Abella

We have implemented a DAMF-aware branch of Abella [8] 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 $\nabla$-quantifier [28] and a treatment of equality based on equivariant higher-order unification [24]). While the current implementation of Abella has succeeded with those meta-theoretic tasks [20,38], 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

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

proof systems do not deal well, in part, because of the need to encode and manipulate bindings [7,21]. 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. Of all these assertions, Abella is able to directly use only the assertions which have annotated productions, wherein the annotation is taken as the *local name* of the external dependency to be used in Abella's proofs; furthermore, Abella only uses those conclusions that are in Abella's own language.

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 $\lambda$-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 $\lambda$Prolog.

## 4 Discussion: Design choices and alternatives

### 4.1 Object representation and alignment

DAMF formula objects are built using strings, which is reasonable as a minimal requirement. DAMF does not specify how these strings are to be interpreted. In particular, it is compatible with DAMF for the strings to be serializations of abstract syntax trees, or for the representation to use more universal notational schemes such as those provided by OMDoc [26] or MMT [34]. The meaning of a formula object is determined by the users of the object and by community standards about what a *language* represents. Nevertheless, it is important to note that even incidental changes to the strings, such as $\alpha$-varying the bound variables, will lead to a different `cid` being assigned to the formula object, and

hence to the sequents and assertions. Participants in DAMF are incentivized to perform any syntactic standardization of their objects before adding them to the global store.

As stated in Section 2.2, a formula object is of the form $(\Sigma, F)_L$ and interpreted as closed, i.e., there are no externally visible symbols. In order to use the formula content $F$ of such an object, a proof checker will need to map the internal symbols introduced in the context $\Sigma$ to its own ambient context. DAMF does not prescribe any particular method in which this kind of mapping is to be done. If as a practical matter the proof checker is not able to align $N = (\Sigma, F)_L$ with its ambient context, but would be able to align a variant $N' = (\Sigma', F')_{L'}$, then, from the perspective of DAMF, the sequent $N \vdash N'$ can be asserted and appealed to to obtain $N'$. The use of this assertion $N \vdash N'$ will then be revealed by trust analysis; other participants can choose whether or not to accept it.

### 4.2 Logical consistency of heterogeneous combinations

DAMF participants are allowed—even incentivized—to combine assertions from multiple sources as long as they are willing to trust the agents making the assertions. This kind of unconstrained combination can at first glance appear to be risky. For example, suppose one attempts to import assertions from incompatible logics, say an assertion in classical logic during the proof of an intuitionistic theorem. Without exceptional care, the resulting theorem 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. 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 used in any arbitrary setting. 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* $(M \vdash N)$ published by $K$ is vacuous and useless (though not strictly "wrong").

### 4.3 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* [9], 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 systems, a great deal of effort has gone into the formalization, manipulation, and transformation of formal proof objects; see, for example, MMT [33], Logipedia [18], and foundational proof certificates [16]. 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.

### 4.4 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 [31], $\lambda$-tree syntax (a.k.a. *higher-order abstract syntax*) [2,21], generic judgments [28], and nominal abstraction [24]. 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 [29], 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. 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, which, in turn, makes errors in the kernel more likely to occur. With DAMF, once a trustworthy but slow kernel—e.g., a certified implementation of a kernel [37]—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.

### 4.5 Other use cases

While it is common to view tools that perform pure computations (such as functional program execution or proof search a la $\lambda$Prolog) 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 [30]. 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* [12,13] 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, we differ from the semantic web by focusing on objects rather than documents, and by using richer notions of logic and compositional reasoning.

Dedukti [6] is a dependently typed $\lambda$-calculus augmented with rewriting. One of the ways in which Dedukti can be used is as a translator: formal proofs from a source system can be transformed to Dedukti proofs and then transformed back into formal proofs in a different system. In particular, the Logipedia documentation mentions that "[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, ..." [27]. Hence, Dedukti may be a way to build trustworthy adapters for DAMF. Note that Dedukti and DAMF have somewhat orthogonal goals; in particular DAMF enables the ability to integrate provers in a distributed setting without needing to first take on the arduous and resource intensive task of producing and transforming proofs. A new prover can be incorporated into DAMF quickly by providing a few adapters that can translate the theorems it establishes into theorems in other provers.

## 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 work is incorporating trust information more uniformly as part of the external import process, as opposed to leaving trust analysis as a post hoc process as in its current implementation in Dispatch.

## 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. 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
7. 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
8. 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
9. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Transactions A of the Royal Society **363**(1835), 2351–2375 (Oct 2005)
10. 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
11. Benet, J.: IPFS-content addressed, versioned, P2P file system (2014). https://doi.org/10.48550/arxiv.1407.3561
12. Berners-Lee, T.: Semantic Web road map. Tech. rep., W3C Design Issues (1998), http://www.w3.org/DesignIssues/Semantic.html
13. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American Magazine (May 2001)
14. Chaudhuri, K., Cousineau, D., Doligez, D., Kuppe, M., Lamport, L., Libal, T., Merz, S., Vanzetto, H.: TLAPS: the TLA$^+$ proof system (2013), available from http://tla.msr-inria.inria.fr/tlaps/
15. 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

16. 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
17. 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
18. Dowek, G., Thiré, F.: Logipedia: a multi-system encyclopedia of formal proofs. http://www.lsv.fr/~dowek/Publi/logipedia.pdf (2019)
19. 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
20. 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
21. 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
22. 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)
23. 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
24. 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
25. Harrison, J.: The HOL Light tutorial (2017), https://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf
26. Kohlhase, M.: OMDoc–An Open Markup Format for Mathematical Documents [version 1.2]: Foreword by Alan Bundy, LNAI, vol. 4180. Springer (2006)
27. Logipedia in a nutshell. http://logipedia.inria.fr/about/about.php (2022)
28. 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
29. Momigliano, A., Pientka, B., Thibodeau, D.: A case-study in programming coinductive proofs: Howe's method. Submitted (2017)
30. 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
31. Pitts, A.M.: Nominal logic, A first order theory of names and binding. Information and Computation **186**(2), 165–193 (2003)

32. 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/
33. Rabe, F.: The future of logic: Foundation-independence. Logica Universalis **10**(1), 1–20 (2016)
34. Rabe, F.: The MMT Language and System. https://uniformal.github.io/ (2022)
35. 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
36. Shankar, N.: Little engines of proof. In: Eriksson, L.H., Lindsay, P.A. (eds.) FME 2002: Formal Methods - Getting IT Right, Proceedings of the International Symposium of Formal Methods Europe (Copenhagen, Denmark). LNCS, vol. 2391, pp. 1–20. Springer (Jul 2002). https://doi.org/10.1007/3-540-45614-7_1
37. 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
38. Tiu, A.: On the role of names in reasoning about $\lambda$-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)

# 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 $n$th 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.

## 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 $\lambda$-equivalence as its equational theory of $\lambda$-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 $n$th 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 [32] or ELPI [19]. 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": {
```

```
13      "language": "damf:bafyreice...",
14      "content": "fib (s (s (s (s (s z))))) ...",
15      "context": ["fib"] } },
16   "contexts": {
17     "fib": {
18       "language": "damf:bafyreice...",
19       "content": [
20          – 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.

```
1  Require Import Arith Lia.
2
3  Fixpoint fib (n : nat) :=
4    match n with
5    | 0 => 0
6    | S j =>
7        match j with
8        | 0 => 1
9        | S k => fib j + fib k
10       end
11   end.
12
13 Ltac liarw F :=
14   let h := fresh "H" in
15   assert (h : F) by (simpl ; lia) ; rewrite h in * ; clear h.
16
17 Theorem fib_square_lemma : forall n, 2 * n + 27 <= fib (n + 12).
18 induction n.
19 - simpl ; lia.
20 - liarw (n + 12 = S (n + 11)).
21   liarw (S n + 12 = S (S (n + 11))).
22   assert (Hl : 2 <= fib (n + 11)).
23     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.

Theorem fib_square_above : forall n, 13 <= n -> n ^ 2 < fib n.
intros n Hle ; pose (k := n - 13) ; liarw (n = k + 13) ; clear Hle.
induction k.
- simpl ; lia.
- liarw (k + 13 = S (k + 12)).
  liarw (S k + 13 = S (S (k + 12))).
  assert (fib (S (S (k + 12))) = fib (S (k + 12)) + fib (k + 12))
    by auto.
  liarw (S (S (k + 12)) ^ 2 = S (k + 12) ^ 2 + 2 * k + 27).
  specialize (fib_square_lemma k).
  lia.
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:

```
{ "format": "assertion",
  "agent": "exampleAgent",
  "claim": {
    "format": "annotated-production",
    "annotation": {"name": "fib_square_above"},
    "production": {
      "mode": "damf:bafyreium...",
      "sequent": {
        "conclusion": "fib_square_above",
        "dependencies": [] } } },
  "formulas": {
    "fib_square_above": {
      "language": "damf:bafyreikf...",
      "content": "forall n, 13 <= n -> n ^ 2 < fib n",
      "context": ["fib_square_above!ctx"] } },
  "contexts": {
    "fib_square_above!ctx": {
      "language": "damf:bafyreikf...",
      "content": [
        "Require Import Arith.",
        "Fixpoint fib (n : nat) := match ··· end."
      ] } } }
```

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  ->
```

```
  (fib x x2 <-> x = z \/ x = s z \/ x = s^12 z).
intros Hnat Hsquare. split.

%% ->
intro Hfib.
Hcs: assert x = z \/ x = s z \/ ··· \/ x = s^12 z \/ leq (s^13 z) x.
  case Hnat. search.
  case Hnat. search.
  ··· /* repeat 12 times in total */
  search. % leq (s^13 z) (s^13 x).
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.

(Note that the above example abuses notation slightly by writing long terms such as the unary representation of the number 13 using an abbreviated form `s^13 z` – this is not valid Abella syntax.)

### B.6 DAMF assertions, trust analysis

A list of the generated assertions from the above combined Abella, λProlog, and Coq developments can be found at `https://distributed-assertions.github.io`. Also at that site is an example of trust analysis performed using the Dispatch tool as described in Section 3.2.