Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar

Jesse Alama[1], Lionel Mamane[2], and Josef Urban[3]

[1] New University of Lisbon
[2] L-7243 Bereldangem, Luxembourg
[3] Radboud University Nijmegen

Abstract. Two methods for extracting detailed formal dependencies from the Coq and Mizar system are presented and compared. The methods are used for dependency extraction from two large mathematical repositories: the Coq Repository at Nijmegen and the Mizar Mathematical Library. Several applications of the detailed dependency analysis are described and proposed. Motivated by the different applications, we discuss the various kinds of dependencies that we are interested in, and the suitability of various dependency extraction methods.

1 Introduction

This paper presents two methods for extracting detailed formal dependencies from two state-of-the-art interactive theorem provers (ITPs) for mathematics: the Coq system and the Mizar system. Our motivation for dependency extraction is application-driven. We are interested in using detailed dependencies for fast refactoring of large mathematical libraries and wikis, for AI methods in automated reasoning that learn from previous proofs, for improved interactive editing of formal mathematics, and for foundational research over formal mathematical libraries.

These applications require different notions of *formal dependency*. We discuss these different requirements, and as a result provide implementations that in several important aspects significantly differ from previous methods. For Mizar, the developed method captures practically all dependencies needed for successful re-verification of a particular formal text (i.e., also notational dependencies, automations used, etc.), and the method attempts hard to determine the minimal set of such dependencies. For Coq, the method goes farther towards re-verification of formal texts than previous methods [5,14,4] that relied solely on the final proof terms. For example, we can already track Coq dependencies that appear during the tactic interpretation, but that do not end up being used in the final proof term.

The paper is organized as follows. Section 2 briefly discusses the notion of formal dependency. Section 3 describes the implementation of dependency extraction in the Coq system, and Section 4 describes the implementation in the Mizar system. Section 5 compares the two implemented approaches to dependency computation. Section 6 describes several experiments and measurements conducted using our implementations on the CoRN and MML libraries, including training of AI/ATP proof assistance systems on the data, and estimating the speed-up for collaborative large-library developments. Section 8 concludes.

2 Dependencies: What Depends on What?

Generally, we say that a definition, or a theorem, T *depends* on some definition, lemma or other theorem T', (or equivalently, that T' is a *dependency* of T) if T "needs" T' to exist or hold. The main way such a "need" arises is that the well-formedness, justification, or provability of T does not hold in the absence of T'. We consider formal mathematics done in a concrete proof assistant so we consider mathematical and logical constructs not only as abstract entities depending on each other, but also as concrete objects (e.g., texts, syntax trees, etc.) in the proof assistants. For our applications, there are different notions of "dependency" we are interested in:

- Purely semantic/logical view. One might claim, for example, that the lambda term (or proof object in the underlying formal framework) contains all sufficient dependencies for a particular theorem, regardless of any notational conventions, library mechanisms, etc.
- Purely pragmatic view. Such dependencies are met if the particular item still compiles in a particular high-level proof assistant framework, regardless of possibly changed underlying semantics. This view takes into account the proof assistant as the major dependency, with their sophisticated mechanisms like auto hint databases, notations, type automations, definitions expansions, proof search depth, parser settings, hidden arguments, etc.

Formal dependencies can also be implicit and explicit. In the simple world of first-order automated theorem proving, proofs and their dependencies are generally quite detailed and explicit about (essentially) all logical steps, even very small ones (such as the steps taken in a resolution proof). But in ITPs, which are generally oriented toward human mathematicians, one of the goals is to allow the users to express themselves with minimal logical verbosity and ITPs come with a number of implicit mechanisms. Examples are type mechanisms (e.g., type-class automations of various flavors in Coq [15] and Isabelle [9], Prolog-like types in Mizar [18,16]), hint mechanisms (in Coq and Isabelle), etc. If we are interested in giving a complete answer to the question of what a formalized proof depends upon, we must expose such implicit facts and inferences.

Formal dependencies reported by ITPs are typically *sufficient*. Depending on the extraction mechanism, redundant dependencies can be reported. Bottom-up procedures like congruence-closure and type closure in Mizar (and e.g., type-class mechanisms in other ITPs) are examples of mechanisms when the ITP uses available knowledge exhaustively, often drawing in many *unnecessary* dependencies from the context. For applications, it is obviously better if such unnecessary dependencies can be removed .

3 Dependency extraction in Coq

Recall that Coq is based on the Curry-Howard isomorphism, meaning that:

1. A statement (formula) is encoded as a type.
2. There is, at the "bare" logical level, no essential difference between a definition and a theorem: they are both the binding (in the environment) of a name to a type (type of the definition, statement of the theorem) and a term (body of the definition, proof of the theorem).
3. Similarly, there is no essential difference between an axiom and a parameter: they are both the binding (in the environment) of a name to a type (statement of the axiom, type of the parameter, e.g. "natural number").
4. There is, as far as Coq is concerned, no difference between the notions of theorem, lemma, corollary, . . .

Thus, in this section, and in other sections when talking of Coq, we do not always repeat "axiom or parameter", nor repeat "definition or theorem or lemma or corollary or . . . ". We will use "axiom" for "axiom or parameter" and "theorem" or "definition" for "definition or theorem or lemma or corollary or . . . ". Similarly for "proof" and "definition body".

There are essentially three groups of Coq commands that need to be treated by the dependency tracking:[4]

1. Commands that register a new logical construct (definition or axiom), either
   - From scratch. That is, commands that take as arguments a name and a type and/or a body, and that add the definition binding this name to this type and/or body. The canonical examples are

     ```
     Definition Name : type := body
     ```

     and

     ```
     Axiom Name : type
     ```

     The type can also be given implicitly as the inferred type of the body, as in

     ```
     Definition Name := body
     ```

   - Saving the current (completely proven) theorem in the environment. These are the "end of proof" commands, such as `Qed`, `Save`, `Defined`.
2. Commands that make progress in the current proof, which is necessarily made in several steps:
   (a) Opening a new theorem, as in

     ```
     Theorem Name : type
     ```

     or

     ```
     Definition Name : type
     ```

---

[4] As far as logical constructs are concerned.

   (b) An arbitrary strictly positive amount of proof steps.
   (c) Saving that theorem in the environment.

   These commands update (by adding exactly *one* node) the internal `Coq` structure called "proof tree".
3. Commands that open a new theorem, that will be proven in multiple steps.

The dependency tracking is implemented as suitable hooks in the `Coq` functions that the three kinds of commands eventually call. When a new construct is registered in the environment, the dependency tracking walks over the type and body (if present) of the new construct and collects all constructs that are referenced. When a proof tree is updated, the dependency tracking examines the top node of the new proof tree (note that this is always the only change with regards to the previous proof tree). The commands that update the proof tree (that is, make a step in the current proof) are called `tactics`. `Coq`'s tactic interpretation goes through three main phases:

1. parsing;
2. Ltac[5] expansion;
3. evaluation.

The tactic structure after each of these phases is stored in the proof tree. This allows to collect all construct references mentioned at any of these tree levels. For example, if tactic `Foo T` is defined as

```
try apply BolzanoWeierstrass;
solve [ T | auto ]
```

and the user invokes the tactic as `Foo FeitThompson`, then the first level will contain (in parsed form) `Foo FeitThompson`, the second level will contain (in parsed form)

```
try apply BolzanoWeierstrass;
solve [ FeitThompson | auto ].}
```

and the third level can contain any of:

- `refine (BolzanoWeierstrass ...)`,
- `refine (FeitThompson ...)`,
- something else, if the proof was found by `auto`.

The third level typically contains only a few of the basic atomic fundamental rules (tactics) applications, such as `refine`, `intro`, `rename` or `convert`, and combinations thereof.

---

[5] Ltac is the `Coq`'s tactical language, used to combine tactics and add new user-defined tactics.

3.1 Dependency availability, format, and protocol

`Coq` supports several interaction protocols: the `coqtop`, `emacs` and `coq-interface` protocols. Dependency tracking is available in the program implementing the `coq-interface` protocol which is designed for machine interaction. The dependency information is printed in a special message for each *potentially progress-making command* that can give rise to a dependency. A *potentially progress-making command* is one whose purpose is to change `Coq`'s state. For example, the command `Print Foo`, which displays the previously loaded mathematical construct `Foo`, is not a potentially progress-making command[6]. Any tactic invocation is a potentially progress-making command. For example, the tactic `auto` silently succeeds (without any effect) if it does not completely solve the goal it is assigned to solve. In that case, although that particular invocation did not make any actual progress in the proof, `auto` is still considered a potentially progress-making command, and the dependency tracking outputs the message `‘‘dependencies: (empty list)’’`. Other kinds of progress-making commands include, for example notation declarations and morphisms registrations. Some commands, although they change `Coq`'s state, might not give rise to a dependency. For example, the `Set Firstorder Depth` command, taking only an integer argument, changes the maximum depth at which the `firstorder` tactic will search for a proof. For such a command, no dependency message is output.

One command may give rise to several dependency messages, when they change `Coq`'s state in several different ways. For example, the `intuition` tactic[7] can, mainly for efficiency reasons, construct an ad hoc lemma, register it into the global environment and then use that lemma to prove the goal it has been assigned to solve, instead of introducing the ad hoc lemma as a local hypothesis through a cut. This is mainly an optimization: The ad hoc lemma is defined as “opaque”, meaning that the typechecking (proofchecking) algorithm is not allowed to unfold the body (proof) of the lemma when the lemma is invoked, and thus won't spend any time doing so. By contrast, a local hypothesis is always “transparent”, and the typechecking algorithm is allowed to unfold its body. For the purpose of dependency tracking this means that `intuition` makes *two* conceptually different steps:

1. register a new global lemma, under a fresh name;
2. solve the current subgoal in the proof currently in progress.

---

[6] Thus, although this commands obviously needs item `Foo` to be defined to succeed, the dependency tracking does not output that information. That is not a problem in practice because such commands are usually issued by a user interface to treat an interactive user request (for example “show me item `Foo`”), but are not saved into the script that is saved on disk. Even if they were saved into the script, adding or removing them to (from, respectively) the script does not change the semantics of the script.

[7] The intuition tactic is a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhof, extended to hand over subgoals which it cannot solve to another tactic.

Each of these steps gives rise to different dependencies. For example, if the current proof is `BolzanoWeierstrass`, then the new global lemma gives rise to dependencies of the form

> "`BolzanoWeierstrass_subproofN` depends on ..."

where the `_subproofN` suffix is Coq's way of generating a fresh name. Closing of the subgoal by use of `BolzanoWeierstrass_subproofN` then gives rise to the dependency

> "`BolzanoWeierstrass` depends on `BolzanoWeierstrass_subproofN`"

3.2 Coverage and limitations

The Coq dependency tracking is already quite extensive, and sufficient for the whole Nijmegen CoRN corpus. Some restrictions remain in parts of the Coq internals that the second author does not yet fully understand.[8] Our interests (and experiments) include not only purely mathematical dependencies that can be found in the proof terms (for previous work see also [14,4]), but also fast recompilation modes for easy authoring of formal mathematics in large libraries and formal wikis. The Coq dependency tracking code currently finds all logically relevant dependencies from the proof terms, even those that arise from automation tactics. It does not handle yet the non-logical dependencies. Examples include notation declarations, morphism and equivalence relation declarations,[9] `auto` hint database registrations,[10] but also tactic interpretation. At this stage, we don't handle most of these, but as already explained, the internal structure of Coq lends itself well to collecting dependencies that appear at the various levels of tactic interpretation. This means that we can already handle the (*non-semantic*) dependencies on logical constructs that appear during the tactic interpretation, but that do not end up being used in the final proof term.

Some of the non-logical dependencies are a more difficult issue. For example, several dependencies related to tactic parametrization (`auto` hint databases, `firstorder` proof depth search) need specific knowledge of how the tactic is influenced by parameters, or information available only to the internals of the tactic. The best approach to handle such dependencies seems to be to change (at the OCaml source level in Coq) the type of a tactic, so that the tactic itself is responsible for providing such dependencies. This will however have to be validated in practice, provided that we manage to persuade the greater Coq community about the importance and practical usefulness of complete dependency tracking for formal mathematics and for research based on it.

---

[8] Such as when and how dynamics are used in tactic expressions or a complete overview of all datatype tactics take as arguments.

[9] So that the tactics for equality can handle one's user-defined equality.

[10] `auto` not only needs that the necessary lemmas be available in the environment, but it also needs to be specifically instructed to try to use them, through a mechanism where the lemmas are registered in a "hint database". Each invocation of `auto` can specify which hint databases to use.

Coq also presents an interesting corner case as far as opacity of dependencies is concerned. On the one hand, Coq has an explicit management of opacity of items; an item originally declared as opaque can only be used generically with regards to its type; no information arising from its body can be used, the only information available to other items is the type. Lemmas and theorems are usually declared opaque[11], and definitions usually declared transparent, but this is not forced by the system. In some cases, applications of lemmas need to be transparent. Coq provides an easy way to decide whether a dependency is opaque or transparent: dependencies on opaque objects can only be opaque, and dependencies on transparent objects are to be considered transparent.

Note that the predicative calculus of inductive constructions (pCIC) uses a universe level structure, where the universes have to be ordered in a well-founded way at all times. However, the ordering constraints between the universes are hidden from the user, and are absent from the types (statements) the user writes. Changing the proof of a theorem T can potentially have an influence on the universe constraints of the theorem. Thus, changing the body of an opaque item T' appearing in the proof of T can change the universe constraints attached to it, potentially in a way that is incompatible with the way it was previously used in the body of T. Detecting whether the universe constraints have changed or not is not completely straightforward, and needs specific knowledge of the pCIC. But unless one does so, for complete certainty of correctness of the library as a whole, one has to consider *all* dependencies as transparent. Note that in practice universe constraint incompatibilities are quite rare. A large library may thus optimize its rechecking after a small change, and not immediately follow opaque reverse dependencies. Instead, fully correct universe constraint checking could be done in a postponed way, for example by rechecking the whole library from scratch once per week or per month.

4 Dependency extraction in Mizar

Dependency computation in Mizar differs from the implementation provided for Coq, being in some sense much simpler, but at the same time also more robust with respect to the potential future changes of the Mizar codebase. For comparison of the techniques, see Section 5. For a more detailed discussion of Mizar, see [12] or [8].

In Mizar, every article A has its own environment $\mathcal{E}_A$ specifying the context (theorems, definitions, notations, etc.) that is used to verify the article. $\mathcal{E}_A$, is usually a rather conservative overestimate of the items that the article actually needs. For example, even if an article A needs only one definition (or theorem, or notation, or scheme, or. . .) from article B, all the definitions (theorems, notations, schemes, . . .) from B will be present in $\mathcal{E}_A$. The *dependencies for an article* A are computed as the smallest environment $\mathcal{E}'_A$ under which A is still Mizar-verifiable (and has the same semantics as A did under $\mathcal{E}_A$). To get dependencies

---

[11] thereby following the mathematical principle of `proof irrelevance`.

of a particular Mizar item I (theorem, definition, etc.,), we first create a *microarticle* containing essentially just the item I, and compute the dependencies of this microarticle.

More precisely, computing fine-grained dependencies in Mizar takes three steps:

Normalization Rewrite every article of the Mizar Mathematical Library so that:

- Each definition block defines exactly one concept.
  Definition blocks that contain multiple definitions or notations can lead to false positive dependencies. For example, if two functions g and g are defined in a single definition block, and a theorem ϕ uses f but not g, then we want to be able to say that ϕ depends on f but is independent of g. Without splitting definition blocks, we have the specious dependency of ϕ upon g.
- All toplevel logical linking is replaced by explicit reference: constructions such as

  ```
  ϕ; then ψ;
  ```

  whereby the statement ψ is justified by the statement ϕ, are replaced by

  ```
  Label1: ϕ;
  Label2: ψ by Label1;
  ```

  where `Label1` and `Label2` are new labels. By doing this transformation, we ensure that the only way that a statement is justified by another is through explicit reference.
- Segments of reserved variables all have length exactly 1. For example, constructions such as

  ```
  reserve A for set,
          B for non empty set,
          f for Function of A, B,
          M for Cardinal;
  ```

  which is a single reservation statement that assigns types to four variables (`A`, `B`, `f`, and `M`) is replaced by four reservation statements, each of which assigns a type to a single variable:

  ```
  reserve A for set;
  reserve B for non empty set;
  reserve f for Function of A, B;
  reserve M for Cardinal;
  ```

  When reserved variables are normalized in this way, one can eliminate some false positive dependencies. A theorem in which, say, the variable `f` occurs freely but which has nothing to do with cardinal numbers has the type `Function of A,B` in the presence of both the first and the second sequences of reserved variables. If the first reservation statement is deleted, the theorem becomes ill-formed because `f` no longer has a type. But the reservation statement itself directly requires that the type `Cardinal` of cardinal numbers is available, and thus indirectly requires a part of the development of cardinal numbers. If the theorem has nothing to do with cardinal numbers, this dependency is clearly specious. By rewriting reserved variables in the second way, though, one sees that one can safely delete the fourth reservation statement, thereby eliminating this false dependency.

These rewritings do not affect the semantics of the Mizar article.

Decomposition For every normalized article A in the Mizar Mathematical Library, extract the sequence $\langle I_1^A, I_2^A, \ldots, I_n^A \rangle$ of its toplevel items, each of which written to a "microarticle" A_k that contains only I_k^A and whose environment is that of A and contains each A_j $(j < k)$.

Minimization For every article A of the Mizar Mathematical Library and every microarticle A_n of A, do a brute-force minimization of smallest environment $\mathcal{E}_{A_n}$ such that A_n is Mizar-verifiable.

The brute-force minimization works as follows. Given a microarticle A, we successively trim the environment for all the Mizar environment item kinds.[12] Each item kind is associated with a sequence s of imported items $\langle a_1, \ldots, a_n \rangle$ and the task is to find a minimal sublist s' of s such that A is Mizar-verifiable.[13]We apply a simple binary search algorithm to s to compute the minimal sublist s'. Applying this approach for all Mizar item kinds, for all microarticles A_k, for all articles A of the Mizar Mathematical Library is a rather expensive computation (for some Mizar articles, this process can take several hours). It is much slower than the method used for Coq described in Section 3. However the result is truly minimized, which is important for many applications of dependencies. Additionally, we have already developed some heuristics that help to find s', and these already do perform tolerably fast.

5 Comparison of the Methods

Some observations comparing the Coq and Mizar dependency computation can be drawn generally, without comparing the actual data as done in the following sections. Dependencies in the case of CoRN are generated by hooking into the actual code and are thus quite exactly mirroring the work of the proof assistant.

In the case of Mizar, dependencies are approximated from above. The dependency graph in this case starts with an over-approximation of what is known to be sufficient for an item to be Mizar-verifiable and then successively refines this over-approximation toward a minimal set of sufficient conditions. A significant difference is that the dependencies in Coq are not minimized: the dependency tracking there tells us exactly the dependencies that were used by Coq (in the particular context) when a certain command is run. Thus, if for example the context is rich, and redundant dependencies are used by some exhaustive strategies, we will not detect their redundancy. On the other hand, in Mizar we do not

---

[12] Namely, theorems, schemes, top-level lemmas, definitional theorems, definientia, patterns, registrations, and constructors. See [8] for a discussion of these item kinds.

[13] There is always one minimal sublist, since we assume that A is Mizar-verifiable to begin with.

rely on the proof assistant reporting how it exactly works, and instead try to exhaustively minimize the set of dependencies, until an error occurs. This process is more computationally intensive, however, it guarantees minimality (relative to the proof assistant's power) which is interesting for many of the applications mentioned below.

Another difference is in the coverage of non-logical constructs. Practically every resource necessary for a verification of a Mizar article is an explicit part of the article's environment. Thus, it is easy to minimize not just the strictly logical dependencies, but also the non-logical ones, like the sets of symbols and notations needed for a particular item, or particular automations like definitional expansions. For LCF-based proof assistants, this typically implies further work on the dependency tracking.

6 Evaluation, Experiments, and Applications

6.1 Dependency extraction for CoRN and MML

We use the dependency extraction methods described in 3 and 4 to obtain fine dependency data for the CoRN library and an initial 100 article fragment of the MML. As described above, for CoRN, we use the dependency exporter implemented directly using the Coq code base. The export is thus approximately as fast as the Coq processing of CoRN itself, taking about 40 minutes on contemporary hardware. The product are for each CoRN file a corresponding file with dependencies, which have altogether about 65 MB. This information is then post-processed by standard UNIX and other tools into the dependency graph discussed below.

For Mizar and MML we use the brute-force dependency extraction approach discussed above. This takes significantly longer than Mizar processing alone, also because of the number of preprocessing and normalization steps that need to be done when splitting articles into micro-articles. For our data this now takes about one day for the initial 100 article fragment of the MML, the main share of this time being spent on minimizing the large numbers of items used implicitly by Mizar. Note that in this implementation we are initially more interested in achieving completeness and minimality rather than efficiency, and a number of available optimizations can reduce this time significantly[14]. The data obtained are again post-processed by standard UNIX tools into the dependency graphs.

In order to compare the benefits of having fine dependencies, we also compute for each library the *full file-based dependency* graph for all items. These graphs emulate the current dumb file-based treatment of dependencies in these libraries: each time an item is changed in some file, all items in the depending files have to be re-verified. The two kinds of graphs for both libraries are then compared in Table 1.

---

[14] For example a very simple recent optimization done for theorems, definitions, and schemes, has reduced the processing time in half.

The graphs confirm our initial intuition that having the fine dependencies will significantly speed up partial recompilation of the large libraries, which is especially interesting in the CoRN and MML formal wikis that we develop.[15] For example, the average number of items that need to be recompiled when a random item is changed has dropped about seven times for CoRN, and about five times for Mizar. The medians for these numbers are even more interesting, increasing to fifteen for Mizar. The difference between MML and CoRN is also quite interesting, but it is hard to draw any conclusions. The corpora differ in their content and use different styles and techniques.

	CoRN/item	CoRN/file	MML-100/item	MML-100/file
Items	9 462	9 462	9 553	9 553
Deps	175 407	2 214 396	704 513	21 082 287
TDeps	3 614 445	24 385 358	7 258 546	34 974 804
P(%)	8	54.5	15.9	76.7
ARL	382	2 577.2	759.8	3 661.1
MRL	12.5	1 183	155.5	2 377.5

Deps Number of dependency edges
TDeps Number of transitive dependency edges
P Probability that given two randomly chosen items, one depends (directly or indirectly) on the other, or vice versa.
ARL Average number of items recompiled if one item is changed.
MRL Median number of items recompiled if one item is changed.

Table 1. Statistics of the item-based and file-based dependencies for CoRN and MML

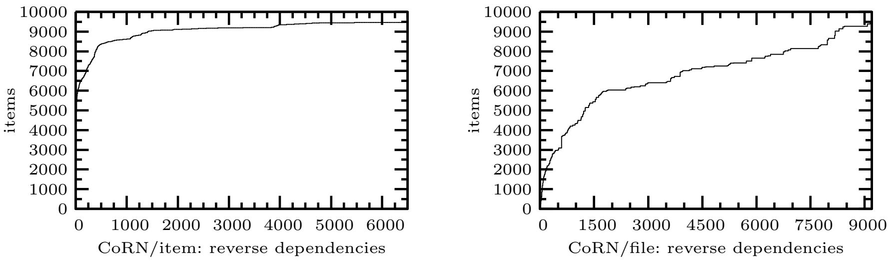

Fig. 1. Cumulative transitive reverse dependencies for CoRN: file-based vs. item-based

Another interesting new statistics given in Table 6.1 is the information about the number and structure of *explicit* and *implicit* dependencies that we have done for Mizar. Explicit dependencies are anything that is already mentioned in the original text. Implicit dependencies are everything else, for example dependencies on type mechanisms. Note that the ratio of implicit dependencies is very significant, which suggests that handling them precisely can be quite necessary for the learning and ATP experiments conducted in the next section.

---

[15] http://mws.cs.ru.nl/mwiki/, http://mws.cs.ru.nl/cwiki/

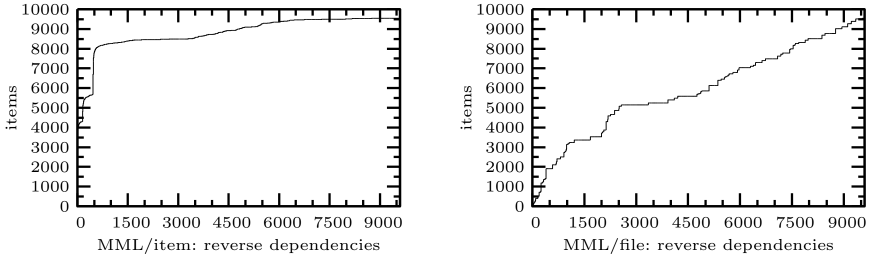

Fig. 2. Cumulative transitive reverse dependencies for MML: file-based vs. item-based

	theorem	top-level lemma	definition	scheme	registration
from	550134	44120	44216	7053	58622
to	314487	2384	263486	6510	108449

Table 2. Statistics of Mizar direct dependencies from and to different items

6.2 Dependency analysis for AI-based proof assistance

The knowledge of how a large number of theorems are proved is used by mathematicians to direct their new proof attempts and theory developments. In the same way, the precise formal proof knowledge that we now have can be used for directing formal automated theorem proving (ATP) systems and meta-systems over the large mathematical libraries. In [3] we provide an initial evaluation of the usefulness of our MML dependency data for machine learning of such proof guidance of first-order ATPs.

These experiments are conducted on a set of 2078 problems extracted from the Mizar library and translated to first-order ATP format. We emulate the growth of the Mizar library (limited to the 2078 problems), by considering all previous theorems and definitions when a new conjecture is attempted (i.e., when a new theorem is formulated by an author, requiring a proof). The ATP problems thus become very large, containing thousands of the previously proved formulas as available axioms, which obviously makes automated theorem proving quite difficult, see e.g. [17] and [13] for details. We run the state-of-the-art Vampire-SInE [10] ATP system on these large problems, and solve 567 of them (with a 10-second timelimit). Then, instead of attacking such large problems directly, we learn proof relevance from all previous fine-grained proof dependencies, using machine learning with a naive Bayes classifier.This technique works surprisingly well: in comparison with running Vampire-SInE directly on the large problems, the problems pruned by such trained machine learner can be proved by Vampire in 717 cases, i.e., the efficiency of the automated theorem proving is raised by about 30% when we apply the knowledge about previous proof dependencies, which is a very significant advance in the world of automated theorem proving, where the search complexity is typically superexponential.

In [2] we further leverage this automated reasoning technique by scaling the dependency analysis to the whole MML, and attempting a fully automated proof for every MML theorem. This yields the so-far largest number of fully automated proofs over the whole MML, allowing us (using the precise formal dependencies of the ATP and MML proofs) to attempt an initial comparison of human and automated proofs in general mathematics.

6.3 Interactive editing with fine-grained dependencies

A particular practical use of fine dependencies (initially motivating the work done on Coq dependencies in 3) is for advanced interactive editing. tmEgg [11] is a TeXmacs-based user interface to Coq.[16] Its main purpose is to integrate formal mathematics done in Coq in a more general document (such as course notes or journal article) without forcing the document to follow the structure of the formal mathematics contained therein.

For example, it does not require that the order in which the mathematical constructs appear in the document be the same as the order in which they are presented to Coq. As one would expect, the latter must respect the constraints inherent to the incremental construction of the formal mathematics, such as a lemma being proven before it is used in the proof of a theorem or a definition being made before the defined construct is used.

However, the presentation the author would like to put in the document may not strictly respect these constraints. For example, clarity of exposition may benefit from first presenting the proof of the main theorem, making it clear how each lemma being used is useful, and then only go through all lemmas. Or a didactic presentation of a subject may first want to go through some examples before presenting the full definitions for the concepts being manipulated.

tmEgg thus allows the mathematical constructs to be in any order in the document, and uses the dependency information to dynamically — and lazily — load any construct necessary to perform the requested action. For example, if the requested action is "check the proof of this theorem", it will automatically load all definitions and lemmas used by the statement or proof of the theorem.

An interactive editor presents slightly different requirements than the batch recompilation scenario of a mathematical library described in 6.1. One such difference is that an interactive editor needs the dependency information, as part of the interactive session, for partial in-progress proofs. Indeed, if any in-progress proof depends on an item T, and the user wishes to change or unload (remove from the environment) T, then the part of the in-progress proof that depends on T has to be undone, even if the dependency is opaque.

---

[16] The dependency tracking for Coq was actually started by the second author as part of the development of tmEgg. This facility has been already integrated in the official release of Coq. Since then this facility was extended to be able to treat the whole of the CoRN library. These changes are not yet included in the official release of Coq.

7 Related Work

Related work exists in the first-order ATP field, where a number of systems can today output the axioms needed for a particular proof. Purely semantic (proof object) dependencies have been extracted several times for several ITPs, for example by Bertot and the Helm project for Coq [5,14,4], and Obua and McLaughlin for HOL Light and Isabelle. The focus of the latter two dependency extractions is on cross-verification, and are based on quite low-level (proof object) mechanisms. A higher-level[17] semantic dependency exporter for HOL Light was recently implemented by Adams [1] for his work on HOL Light re-verification in HOL Zero. This could be usable as a basis for extending our applications to the core HOL Light library and the related large Flyspeck library. The Coq/CoRN approach quite likely easily scales to other large Coq libraries, like for example the one developed in the Math Components project [7]. Our focus in this work is wider than the semantic-only efforts: We attempt to get the full information about all implicit mechanisms (including syntactic mechanisms), and we are interested in using the information for smart re-compilation, which requires to track much more than just the purely semantic or low-level information.

8 Conclusion and Future Work

In this paper we have tried to show the importance and attractiveness of formal dependencies. We have implemented and used two very different techniques to elicit fine-grained proof dependencies for two very different proof assistants and two very different large formal mathematical libraries. This provides enough confidence that our approaches will scale to other important libraries and assistants, and our techniques and the derived benefits will be usable in other contexts.

Mathematics is being increasingly encoded in a computer-understandable (formal) and in-principle-verifiable way. The results are increasingly large interdependent computer-understandable libraries of mathematical knowledge. (Collaborative) development and refactoring of such large libraries requires advanced computer support, providing fast computation and analysis of dependencies, and fast re-verification methods based on the dependency information. As such automated assistance tools reach greater and greater reasoning power, the cost/benefit ratio of doing formal mathematics decreases.

Given our previous work on several parts of this program, providing exact dependency analysis and linking it to the other important tools seems to be a straightforward choice. Even though the links to proof automation, fast large-scale refactoring, and proof analysis, are very fresh, it is our hope that the significant performance boosts already sufficiently demonstrate the importance of good formal dependency analysis for formal mathematics, and for future mathematics in general.

---

[17] By *higher-level* we mean tracking *higher-level* constructs, like use of theorems and tactics, not just tracking of the low-level primitive steps done in the proof-assistant's kernel.

References

1. Adams, M.: Introducing HOL Zero. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) Mathematical Software—ICMS 2010, LNCS, vol. 6327, pp. 142–143. Springer (2010), http://dx.doi.org/10.1007/978-3-642-15582-6_25
2. Alama, J., Kuehlwein, D., Urban, J.: Automated and human proofs in general mathematics: An initial comparison. In: LPAR. Lecture Notes in Computer Science, Springer (2012), accepted
3. Alama, J., Kühlwein, D., Tsivtsivadze, E., Urban, J., Heskes, T.: Premise selection for mathematics by corpus analysis and kernel methods. CoRR abs/1108.3446 (2011), http://arxiv.org/abs/1108.3446
4. 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)
5. Bertot, Y., Pons, O., Pottier, L.: Dependency graphs for interactive theorem provers. Tech. rep., INRIA (2000), report RR-4052
6. Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.): Mathematical Software - ICMS 2010. Proceedings, LNCS, vol. 6327. Springer (2010)
7. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Tobias Nipkow, Christian Urban (eds.) TPHOLs. LNCS, vol. 5674. Springer (2009), http://hal.inria.fr/inria-00368403/en/
8. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning 3(2), 153–245 (2010), http://jfr.cib.unibo.it/article/view/1980/1356
9. Haftmann, F., Wenzel, M.: Constructive type classes in isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES. Lecture Notes in Computer Science, vol. 4502, pp. 160–174. Springer (2006)
10. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: CADE 11 (2011), to appear
11. Mamane, L., Geuvers, H.: A document-oriented Coq plugin for TEXMACS. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM 2007 - Work in Progress. RISC Report, vol. 07-06, pp. 47–60. University of Linz, Austria (2007), technical report no. 07-06 in Series
12. Matuszewski, R., Rudnicki, P.: Mizar: the first 30 years. Mechanized Mathematics and Its Applications 4, 3–24 (2005)
13. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
14. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: UITP 1998. Eindhoven University of Technology (1998)
15. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. CoRR abs/1102.1323 (2011)
16. Urban, J.: MoMM—fast interreduction and retrieval in large libraries of formalized mathematics. International Journal on Artificial Intelligence Tools 15(1), 109–130 (2006)
17. Urban, J., Hoder, K., Voronkov, A.: Evaluation of automated theorem proving on the Mizar Mathematical Library. In: Fukuda et al. [6], pp. 155–166
18. Wiedijk, F.: Mizar's soft type system. In: Schneider, K., Brandt, J. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 4732, pp. 383–399. Springer (2007)