A Coq plugin to export its libraries to XML

Claudio Sacerdoti Coen <claudio.sacerdoticoen@unibo.it>

Department of Computer Science and Engineering - University of Bologna

Abstract. We introduce a plugin for the interactive prover Coq to export its libraries to a machine readable XML format. The information exported is the one checked by Coq's kernel after the input is elaborated, augmented with additional data coming from the elaboration itself.
The plugin has been applied to the 49 Coq libraries that have an opam package and that currently compile with the latest version of Coq (8.9.0), generating a large dataset of 1,235,934 compressed XML files organized in 18,780 directories that require 17GB on disk.

1 Introduction

In recent years there is a renewed interest in exporting libraries of interactive provers (ITP) in machine-readable formats that can be checked, mined and elaborated independently of the ITP. Examples of possible applications are: automatic alignment of concepts coming from different provers or from different libraries [MGK+17]; machine learning of premise-selection [ISA+16] or automatic methods to advance in proofs [AGS+04]; implementation of hammers to blast away proof obligations [CK18]; system independent search & retrieval [KS06]; independent verification.

Coq [Tea19] is one of the major ITPs in use and its libraries are huge. The first XML export from Coq was presented at MKM 2003 by the author [Sac03] and the exported library was heavily used in the European project MoWGLI (Math on the Web: Get it by Logic and Interfaces) [AW02]. Since then both the logic and the implementation of Coq were changed, in particular adding a complex system of modules and functors to structure libraries. Therefore the exportation code ceased to work a few years after. More recently several plugins have been written independently to export libraries from Coq. None of the plugins attempt to provide a complete exportation. For example, the ones implemented for the Logipedia project [DT] and to implement a Coq Hammer [CK18] both ignore modules and functors.

In this paper we introduce a new Coq plugin that resumes most of the functionalities of the 2003 plugin, but extends them to cover all recent features of Coq. The output of the plugin is already used as input to other projects. In particular, in [MRS] the author and collaborators translate the generated XML files to the MMT language [RK13] preserving typeability and in [CKM+] they automatically extract from the XML files a large set of RDF triples that can be used to perform system agnostic searches in the library.

2 Introduction to Coq and the XML exportation plugin

Following a rather standard architecture for an interactive theorem prover, user provided files are written in a source (also called external) language and are then elaborated by Coq to an internal language that is fed to the kernel. The kernel is only responsible for verifying that the input is correct according to the rules of the logic/type-system. The elaborator (also called refiner [ASTZ06]) is instead a much more complex piece of software that, among other things, is responsible for: inferring omitted types; resolving mixfix mathematical notation and typical ambiguity; patching the user input introducing subterms to promote values from a type to another; automatically search for missing proof terms necessary to interpret proof statements or single proof steps; elaborating high-level declarations in terms of a simpler language (e.g. implement canonical structures using records that are in turn implemented using inductive types).

Writing independent tools able to understand the external language of an interactive theorem prover is an impossible challenge: not only the external language is subject to continuous changes, but to understand it one would need to reimplement all the algorithms of the elaborator, that are continuously improved, typically undocumented and based on a huge number of ever evolving heuristics. Therefore tools that address interoperability usually tackle the internal language understood by the kernel. This language is usually small, simple and well-documented (it is the syntax of the type theory or logic the system is based on) in order to reduce the trusted code base to a minimum. This is the so called De Brujin criterium. Moreover, since the logic evolves slowly, the code of the kernel is usually quite stable, making it easier to write exporters.

Unfortunately, the type system Coq is based on is all but small and simple, and more features are constantly added. The kernel of Coq is indeed about 13,000 lines of OCaml code and the one of Matita [ARST11], that implements a subset and a further simplification of the type theory of Coq, has more than 4,000 lines of OCaml code. For this reason in Section 2.2 we will present the grammar of the language without attempting to explain all the various constructors in detail.

Elaboration is a non invertible transformation: from the kernel language it is from hard to impossible to reconstruct the user input. In particular, all extra-logical information, i.e. information not necessary to implement type-checking, is not passed to the kernel. This complicates a lot the writing of an exporter, in particular for Coq. Indeed, in order for the exported library to be user-understandable or for tools that learn from the library to be able to apply knowledge to new theories, some extra-logical information is to be retained during the exportation. I list here a few, significant examples:

1. when the user introduces a definition, it flags it with an extra-logical flavor like theorem, lemma, corollary, fact, definition, let, etc.; similar synonyms can be used for axioms.
2. some definitions are automatically generated by the system, e.g. when some tactics are applied or an inductive type is defined; those are indistinguishable from the user provided ones in the kernels

3. the user can explicitly name universes and add constraints over them; the system automatically generates hundreds of additional universes, automatically generating meaningless names, and the kernel cannot distinguish between the universes that the user named because they were meaningful to him, and the remaining ones
4. some definitions are flagged by the user to be used automatically in certain situations, e.g. as rewriting rules, as coercions to promote values to different types, as canonical structure instances to be used during proof search for an instance. If this information is forgot, it becomes practically impossible to reuse the library in a different system
5. sections are a mechanism, similar to Isabelle's locales [KWP99], that allow to declare a logical context of declarations and assumptions, give definitions in this context and later automatically abstract all definitions over the variables of the context that were actually used. The kernel only receives the result of this abstraction, while a user looking at the library only sees the definition before the abstraction.

2.1 The XML exportation plugin.

To address all the examples at the end of the previous section and a few more, we designed our plugin as follows: we have forked the latest official release of Coq (8.9.0) in the Git repository `https://github.com/sacerdot/coq`. Ideally, the fork should just add to the source code some hooks to the kernel and to the elaborator, so that the XML plugin could export all the required information. In practice, the fork does more and therefore it will take more time to negotiate with the Coq team how to incorporate the changes upstream.

First of all, to deal with the case of sections, when the occurrence of a constant is found the plugin needs to know if the constant was declared inside a section or not. The reason is that, to preserve sections in the output, the output of the plugin produces terms in a variation of the calculus where abstractions over variables in a section are explicitly instantiated by name (see [Sac03] or [Sac04] for details). To recover this information, Coq must remember it in the compiled files. The code to perform this recording is still present in the Coq code from the 2003 version of the plugin, but it does not work correctly any more. Therefore we have to patch it.

Second, the plugin must be automatically activated when Coq starts, in order to export all libraries, comprising the basic ones. The mechanism Coq uses to load and activate a plugin is to insert a special command in a script. However, that's too late: when the command is encountered, some information is already lost. Therefore the fork automatically registers and activate the plugin.

Third, there are some "bugs" in the way Coq represents some information in the kernel. In practice, some information is recorded in a wrong way w.r.t. how it is recorded elsewhere, but the rest of the code is able to cope with it. Since two wrongs don't make a right, the data remains bugged and exporting it correctly would complicate the plugin code a lot. We prefer therefore to fix the representation in the code of Coq.

The need to patch some functions in the code of Coq partially explains why We did not try to use SerAPI [GA16] or a modification of it to implement XML serialization outside the system. The main reason, though, is that we take care of exporting also information that is not recorded in the kernel because it can be easily computed inside Coq, but not outside it without reimplementing all of the kernel. For example, for every sub-term of a proof we export its type if it is proving a formula, i.e. when the sub-term is a proof step according to Curry-Howard (see [Sac01,Sac03] for details). This information has many interesting uses: first of all it allows to implement tools to explain the proof term in pseudo-natural language [Sac10]; secondly, by identifying the statement of all proof steps in every proof, it creates a formidable dataset for machine learning tools that try to automatically prove theorems.

Manual plugin activation The plugin has only been tested on Linux. In order to activate the plugin, it is sufficient to set the `$COQ_XML_LIBRARY_ROOT` variable to point to a directory that will contain the extracted files. After that one can just run the Coq compiler `coqc` in the usual way on an input file to extract its content. For example, the command `coqc -R mathcomp.field . galois.v` runs the compiler on the file `galois.v` after declaring that all files rooted in the current directory `.` will be compiled in the logical prefix `mathcomp.field`. Thus, the content of the file `galois.v` will be exported on disk in the directory `$COQ_XML_LIBRARY_ROOT/mathcomp/field/galois/`. See Section 3 for details on the representation on disk of the extracted information.

Even if manual plugin activation is easy, it does not address the question of how to automate extraction of XML data from existing libraries. In order to solve the issue, we set up extraction scripts that exploit the opam system.

Opam packages There is a recent effort by the Coq team to push developers of libraries to release *opam* packages for them. Opam is a package manager for OCaml libraries that can be (ab)used to automatically download, compile and install Coq libraries as well, keeping track of versioned dependencies and automatically installing them on demand. Moreover, to release an opam package some Dublin-core like metadata like author and synopsis must be provided. Other interesting mandatory metadata are license and version. Finally, opam packages specify the exact Coq version they depend on, granting that compilation will succeed. At the time of writing of this paper, there are only 49 opam packages that compile with the recently released version 8.9.0 of Coq. More are under porting and will be available soon. However, there are also hundreds of additional libraries, many no longer maintained, that have no corresponding opam package.

With significant help by Tom Wiesing (FAU Erlangen-Nürnberg), we set up bash and python scripts to automate the XML exportation from opam packages. In particular, for every opam package the scripts create a Git repository hosted at the following address: `https://gl.mathhub.info/Coqxml`. Currently, each repository contains a copy of the `$COQ_XML_LIBRARY_ROOT` the package extracted to, plus Comma Separated Values (CSV) and Resource Description Framework

(RDF) files automatically generated from the XML files (see [CKM+]). Our idea is that further elaboration of the XML exportation by third parties could also in the future be committed to the repository. For example, we are planning to populate the repository also with the translation of the XML files to MMT (see [MRS]).

In order to have one's library included, it is sufficient to release an opam package for it following the instructions on the `https://coq.inria.fr/opam/www/` repository.

2.2 The internal language of Coq

The Calculus of (Co)Inductive Constructions, the type theory Coq is based on, is very complex and Coq extended it multiple times, e.g. with a module system inspired by the OCaml language. We briefly sketch here our enrichment of the internal language recognized by Coq 8.9.0, i.e. an abstract and simplified description of the language understood by the kernel of Coq enriched with additional information coming from the external syntax and preserved by our plugin by means of hooks and additional tables.

The grammar of the enriched internal language is in Table 1. Some additional constraints are not captured by the grammar. In detail, modules and module types cannot be nested inside sections and each extra-logical flavor is allowed to occur only in a subset of the positions where a flavor is expected.

The starting symbol of the grammar is *file*, that represents the content of a `.v` file. We represent a `.v` file as a set of `type-theory-flags` followed by a list of declarations. The flags are not actually written in the file by the user, but passed on the command line to `coqc`. They can be used to change the rules of the type theory to make the `Set` sort impredicative, to collapse the universe hierarchy (and obtain an inconsistent system) or to disable termination checking for functions (and also obtain an inconsistent system).

Due to space constraints we give only an extremely dense explanation of the language, focusing on the features relevant to the exportation.

Comments, flavors and requires Comments and flavors are extra-logical data preserved from the user input. The plugin records as a flavor the exact keyword that was used to introduce a definition or declaration. Some flavors only carry the intent of the definition (e.g. to be a theorem or a lemma). Some others record information about the use of the definition during elaboration (e.g. as a coercion, type class instance, canonical structure).

The only comments preserved by the plugin are special comments in the source file that are meant to be elaborated by the `coqdoc` tool. These are the comments that the user expect to see in the automatically generated HTML description of a library. Therefore we assume those to be important.

An extra-logical **Requires** statement is used in a source file to ask Coq to elaborate another source file. Require statements force a direct acyclic graph (DAG) structure on Coq libraries. We preserve the statement because this structure is meaningful to the user and likely to be exploited also by automated tools.

Table 1. The enriched internal language

file	::=	`type-theory-flags` $(\texttt{decl})^*$
`decl`	::=	*extra logical commands*
		`"a comment: some text + markup here"`
	|	**Requires** *file*
	|	*base logic declarations*
		$\mathcal{F}\ \underline{e}@\{y^*\} : E[:= E]$
	|	**Universe** $\underline{u}$
	|	**Constraint** $u(< \mid \leq \mid =)u$
	|	$\mathcal{F}$ (**Inductive** | **CoInductive**) $(\underline{e}@\{(y)^*\} : E := (\underline{e}@\{(y)^*\} : E)^*)^+$
		section declarations and variables in sections
	|	**Section** $\underline{s}$:= `decl`*
	|	$\mathcal{F}$ **Variable** $x : E$
	|	**Polymorphic Universe** y
	|	**Polymorphic Constraint** $(u \mid y)(< \mid \leq \mid =)(u \mid y)$
		module (type) declarations
	|	**Module Type** $\underline{m}\ (\underline{m} : T)^*\ <: T^*\ := (T \mid \texttt{decl}^*)$
	|	**Module** $\underline{t}\ (\underline{m} : T)^*\ [: T]\ <: T^*\ [:= (M \mid \texttt{decl}^*)]$
		base logic expressions E and universes U
E	::=	$e@\{(U)^*\} \mid x \mid$ `Prop` $\mid$ `Set` $\mid$ `Type`$@\{U\} \mid \Pi x : E.E \mid \lambda x : E.E \mid E\ E$
	|	`Match` $e\ E\ E\ E^* \mid$ (`Fix` $\mid$ `CoFix`) $\mathbb{N}\ (\underline{e} : E := E)^* \mid$ `let` $x : E := E$ `in` E
	|	$E.\mathbb{N} \mid (E : E)$
U	::=	$u \mid y \mid \max U\ U \mid$ `succ` U
		module type expressions T and module expressions M
T	::=	[!] $t\ m^* \mid T$ `with` $e' := E \mid T$ `with` $m' := M$
M	::=	[!] $m\ m^*$
e, u, m, t	::=	qualified identifiers of expressions, universes, modules, module types
e', m'	::=	relative qualified identifiers of expressions, modules
$\underline{e}, \underline{u}, \underline{m}, \underline{t}$	::=	fresh (unqualified) identifiers
$\mathbb{N}$	::=	a natural number
		extra-logical flavours
$\mathcal{F}$	::=	**Lemma** | **Theorem** | **Corollary** | **Declaration** | **Definition** | **Axiom**
	|	**Conjecture** | **Let** | **Example** | **Coercion** | **SubClass** | **Remark**
	|	**CanonicalStructure** | **Fixpoint** | **CoFixpoint** | **Projection**
	|	**IdentityCoercion** | **Scheme** | **Instance** | **Method** | **Fact** | **Property**
	|	**Proposition** | **Record** | **Inductive** | **Coinductive** | **Assumption**
	|	**Hypothesis** | **Conjecture** | **LocalDefinition** | **LocalLet** | **LocalFact**

Qualified identifiers The library is organized hierarchically. When forming base logic expressions E, module expressions M, and module type expressions T, declarations can be referred to by qualified identifiers e, m, resp. t formed from

1. The root path (sequence of identifiers) that is defined by the Coq package. This is the first argument of the `-R` option described above. Typically, but not necessarily, every package introduces a single unique root identifier.
2. One identifier each for every directory or file that leads from the package root to the respective Coq source file.
3. One identifier each (possibly none) for every nested module (type) inside that source file that contains the declarations.
4. The unqualified name $\underline{e}$, $\underline{u}$, $\underline{m}$, resp. $\underline{t}$.

Note that section identifiers do not contribute to qualified names: the declarations inside a section are indistinguishable from the ones outside the section. The only exception to the rule is for section variables: their qualified expressions follow the same rules above, but after the identifier for the innermost module we have the list of identifiers of the sections that surround the section variable.

Universes are also assigned by Coq qualified identifiers u, but, since universes as considered to be globally declared anywhere, the module identifiers do not contribute: the identifier is obtained putting an unique progressive number that carries no logical content after the file name.

Relative qualified names are always relative to a module type, i.e. they are missing the root identifiers and the directory identifiers parts.

Definitions, declarations, terms and universes Besides the usual λ-calculus with `let...in`, the expressions include dependent products $\Pi x : term.term$ (used to type λ-abstractions), sorts `Prop`, `Set`, `Type`@{U} (used to type types), casts $(E : E)$, (co)inductive types with primitive pattern-matching, (co)fixpoints definitions (i.e. terminating recursive functions) and record projections ($E.\mathbb{N}$). Notably, Coq maintains a partially ordered **universe hierarchy** (a direct acyclic graph) with consistent constraints of the form $U(< \mid \leq)U'$. Coq implements universe polymorphism, i.e. definitions and declarations of constants, inductive types and inductive constructors are abstracted over a list $(y)^*$ of universe variables that can occur in universe expression. Occurrences of constants are always fully applied to actual universe expressions: $e@\{(U)^*\}$. Finally, for (co)inductive types Coq implements a notion of subtyping: when two occurrences of an (co)inductive type are compared for subtyping, the formal universe parameters in corresponding position are compared according to the variance annotation associated to each actual universe parameter. The variance information is not explicitly shown in the grammar, but we will explicitly export it.

Module and module types Module types and module are the main mechanism for grouping base logic declarations. Public identifiers introduced in **modules** are available outside the module via qualified identifiers, whereas **module types** only serve to describe what declarations are public in modules. We say that a module M *conforms* to the module type T if

- M declares every constant name that is declared in T and does so with the same type,
- M declares every module name that is declared in T and does so with a module type that conforms to the type (= the set of public declarations) of the module in T,
- if any such name has a definiens in T, it has the same definiens in M.

Conformation of a module *type* to a module type is defined accordingly.

Both modules and module types may be defined in two different ways: *algebraically* as an abbreviation for a module (type) expression (the definiens), or *interactively* as a new module (type) given by a list of declarations (the body). A module may also be abstract, i.e., have neither body nor definiens. Every module (type) expression can be elaborated into a list of declarations so that algebraic module (type) declarations can be seen as abbreviations of interactive ones. The $<:$ and $:$ operators may be used respectively to attach conformation conditions to a module or to restrict the interface of a module.

Module (type)s can be abstracted over a list of module bindings $\underline{m} : T$, which may be used in the definiens/body. When the list is not empty the module (type) is called a *functor (type)*. A functor must be typed with a functor type that has the same list of module bindings.

Module (type) expressions can be obtained by functor application, whose semantics is defined by β-reduction in the usual way, unless "!" annotations are used. According to complex rules that we will ignore in the rest of the paper for lack of space, the "!" annotations performs β-reduction and then triggers the replacement of constants defined in the actual functor argument with their definiens. Finally, the `with` operator adds a definition to an abstract field in a module type.

Sections Coq files and module (type)s can be divided into nested **sections**. These are similar to module functors except that they abstract over base logic declarations, which are interspersed in the body and marked by the **Variable** and **Polymorphic** keywords. The section itself has no semantics: outside the section, all constant and inductive type declarations are $\lambda\Pi$-abstracted over all **Variable** that occur in the declaration and over all **Polymorphic** declarations.

3 The exported data

In this section we describe how the enriched internal language of Coq is mapped to disk.

URIs We turn the qualified names of Coq to Uniform Resource Identifiers (URI) according to the following schema:

- A qualified module, module type or section identifier like $mathcomp.field.galois.SplittingFieldTheory$ is turned into the URI `cic:/mathcomp.field/galois/SplittingFieldTheory` simply prepending `cic:/` and turning all dots into slashes

– A qualified variable/constant/mutual/(co)inductive/block identifier *path.id* is mapped to `uripath/id.ext` where `uri` is the URI computed for *path* and `ext` belongs to the set {`var`,`con`,`ind`} according to what it refers to
– A qualified universe identifier is used as an URI without modifications. The reason is that all universes are considered to be global by Coq and there is no XML file on disk that declares them anyway because no information is attached to the universe. In other words, a qualified universe identifier is just an unique identifier with no attached meaning.

Namespaces We assume the following invocation of `coqc`:

```
coqc -R logical_path physical_path path_to_file/filename.v
```

used to compile a `filename.v` file whose path `path_to_file` is a suffix of the physical path `physical_path`. Then the logical path of the file is obtained replacing `physical_path` with `logical_path` in `path_to_file`, mapping all slashes to dots and removing the `.v` extension.

For example, `coqc -R mathcomp . field/galois.v` and `coqc -R mathcomp.field field field/galois.v` both determine the logical path `mathcomp.field.galois`.

The logical path is mapped to disk by creating a nested directory for each identifier but the filename. In the example above, we create `mathcomp/field` and enter the directory before exporting the file.

We call a directory obtained in this way a *namespace*. Namespaces can be identified on disk because they lack both a corresponding `.role` file (see module (type)s and sections below) and a corresponding `.theory` file (see files below).

Files Each file `foo.v` is mapped to disk to two different structures: a directory `foo` that contains the logical entries of the file (excluding the type theory flags) and a `foo.theory.xml` file that contains the type theory flags and all the extra-logical information (comments, flavors, requires).

The `foo.theory.xml` file is an XHTML file obtained running `coqdoc` on the concatenation of the comments in the enriched internal language expression that the file is mapped to. Moreover, XML elements belonging to the "`ht:`" XML namespace are added to the XHTML file as follows:

– one `ht:TYPETHEORY` element whose attributes encode the type theory flags
– one `ht:REQUIRE` element for each **Requires** in the file, with an attribute that holds the URI of the included file
– one `ht:DEFINITION`, `ht:AXIOM`, `ht:THEOREM` or `ht:VARIABLE` for every declaration or definition of a constant, variable or mutual inductive types block that where explicitly provided by the user. The ones generated automatically are not recorded in the `.theory.xml` file. We use `ht:AXIOM` for declared constants that do not have a definiens and `ht:THEOREM` for defined constants whose type is a proposition, i.e. it is typed by `Prop`. An attribute of the element specifies the URI of the declaration or definition while a second attribute gives the exact flavor.

– one `ht:UNIVERSE` for each (polymorphic) universe declaration that was explicitly provided by the user. The ones generated automatically are not recorded in the `.theory.xml` file. An attribute of the element specifies the URI of the declaration; a second attribute records the user provided universe name, that cannot be inferred from the URI; a third attribute whether the universe declaration is polymorphic or not.
– one `ht:CONSTRAINT` for each (polymorphic) universe constraint declaration that was explicitly provided by the user. The ones generated automatically are not recorded in the `.theory.xml` file. Three attributes specify the two sides of the constraint and the constraint relation.
– one `ht:SECTION` for each section. An attribute specify the section identifier. The context of the element is obtained recursively processing the content of the section.
– one `ht:MODULE` for each module declaration or module (type) definition. An attribute allow to distinguish between an interactive module, an algebraic module, an interactive module type, and an algebraic module type. Another attribute is the module (type) identifier. Finally the context of the element is obtained recursively processing the content of the section.

The next paragraphs describe the logical information that goes into the `foo` directory. With the exception of the type theory flags that are stored in the `foo.theory.xml` file only, all the logical information is inside the directory and a tool interested only into that (e.g. an independent verifier) could ignore the `foo.theory.xml` file completely.

On the other hand a tool that wants to stay close to the original input by the user will only consider the file in the `foo` directory that are referenced (by means of URIs) in the `foo.theory.xml`. In this way definitions automatically generated by Coq are ignored.

Modules and module types We recall that modules and module types are declared or defined according to the following grammar:

$$\begin{array}{ll} \textbf{Module Type} & \underline{m}\ (\underline{m} : T)^*\ <: T^*\ :=\ (T \mid \texttt{decl}^*) \\ \textbf{Module} & \underline{t}\ (\underline{m} : T)^*\ [\,: T\,]\ <: T^*\ [\,:= (M \mid \texttt{decl}^*)\,] \end{array}$$

The exporter maps both to disc in a very similar way. Therefore it is useful for the sake of the discussion to unify the two entries using the generalized grammar

$$\textbf{Module}\ [\textbf{Type}]\ \underline{mt}\ (\underline{m} : T)^*[\,: T\,] <: T^*\ [\,:= (MT \mid \texttt{decl}^*)\,]$$

where $\underline{mt}$ can be either $\underline{m}$ or $\underline{t}$ and MT can be either M or T.

On disk we create:

1. a directory $\underline{mt}$ that contains all the logical declarations exported by $\underline{mt}$, i.e. the interface (or actual type) of $\underline{mt}$. Concretely, if the module type expression $[\,: T\,]$ is provided, the exportation of its flattening will be the content of the directory $\underline{mt}$. Otherwise, if $\texttt{decl}^*$ is provided, then it is the content of the

directory. Otherwise, if the module (type) expression MT is provided, the exportation of its flattening will be the content of the directory.
In practice, tools that work on the XML files do not need to know anything about module (type) expressions and can just work on lists of logical declarations. Moreover, looking in the $\underline{mt}$ directory only, they will directly see the interface of the module (type), without having to re-implement the case reasoning above.

2. a directory $\underline{mt}$`.impl` if the module type expression $[\,:T]$ and $[\,:=M \mid$ `decl`$^*]$ are both provided. The directory $\underline{mt}$`.impl` contains all the logical declarations generated by the module definiens: if the definiens is a module expression M, then we export its flattening; otherwise we directly export the list of declarations `decl`*.
   Tools can inspect this directory to know what is the witness for the module type interface $\underline{mt}$. We call it a witness and not an implementation because the module system of Coq does not allow to recover the definiens in any way. This is counter-intuitive w.r.t. programming languages of the ML family where the implementation is the one that is actually run after linking when the program is invoked. The only reminiscent of this in Coq happens when OCaml code is extracted from the Coq development: Coq modules are extracted to OCaml modules and the witness becomes accessible to computation.
3. a file $\underline{mt}$`.role`. The file is a textual file (not an XML file) that only contains one keyword to allow to distinguish the various uses of directories. In particular, for a module declaration the keyword is `DeclaredModule`. It is `Module` for a module definition or texttttModuleType for a module type definition.
4. every module declaration in $(\underline{m} : T)^*$ is exported as if it were declared via **Module** $\underline{m} : T$ inside $\underline{mt}$ and also inside $\underline{mt}$`.impl`, if the latter exists. The only difference that distinguish the parameter $\underline{m}$ from an actual module declaration is $\underline{mt}.\underline{m}$`.role` (and also $\underline{mt}$`.impl.`$\underline{m}$`.role` if it exists) that uses the keyword `Parameter` in place of `DeclaredModule`.
   This choice is perfectly coherent both with the naming scheme of Coq and with the semantics of functor type application: $\underline{t}\ \underline{m}'$ and $\underline{t}$ `with` $\underline{m} := \underline{m}'$ yields exactly the same result respectively when $\underline{t}$ is a functor type abstracted over the parameter $\underline{m} : T$ and when $\underline{t}$ is a module type that begins with a declaration of $\underline{m} : T$. Indeed most tools are likely to ignore the distinction, that becomes relevant only when parameterized module types are used to type functors. Search for "covariant translation" in [MRS] for a lengthy discussion of this issue.
5. if $\underline{mt}$ was obtained flattening a module expression or a module theory expression, then an XML file $\underline{mt}$`.expr.xml` is generated to describe the expression.
6. if $\underline{mt}$`.impl` was obtained flattening a module expression, then an XML file $\underline{mt}$`.impl.expr.xml` is generated to describe the expression.
7. if the list of module theory expressions $<: T^*$ is not empty, a file $\underline{mt}$`.sub.xml` is generated. It contains the list of description of module expressions.
   This is the only case where we do not export the flattened version of the module expression. Therefore it will be hard to write tools that exploit the $<: T^*$ construct. Note, however, that the only use of this in Coq is for users

to check conformity with a module type in order to later feed the module to a functor. In other words, it is only used to catch errors earlier. Adding the flattening to disk just require a few lines of code, but it generates many more files when the construct is used. Therefore we plan to export the flattening only if somebody will require that information in the future.

Tools that want to parse the latter three entries are required to have full understanding of the flattening process of Coq. For the time being we do not know of any potential application requiring such a knowledge. Moreover, the expressions in XML ignore the "!" flags to functor applications: inside the kernel of Coq the flag are represented using abstract types that do not allow to recover the user provided information. To print it out we would need another invasive patch to the kernel. We leave it as future work in case some tools that care about module expressions will be created.

Sections A section $\underline{s}$ is exported as a directory $\underline{s}$ plus a textual file $\underline{s}$`.role` containing the keyword `Section`. The sections and variables declared into the section $\underline{s}$ are recursively exported inside the directory $\underline{s}$. The constants and inductive types declared into $\underline{s}$, instead, are exported outside the sections surrounding them. The mapping between logical URIs and the physical path is therefore preserved, since sections names do not contribute in Coq to qualified identifiers (see Section 2.2). This choice is backward compatible with [Sac01].

Declarations and definitions We distinguish the case of variables, constants and blocks of mutual (co)inductive types.

- Case $\mathcal{F}\ \underline{e}@\{y^*\} : E[:= E']$: we create three or four files $\underline{e}$`.con.xml.gz`, $\underline{e}$`.con.body.xml.gz`, $\underline{e}$`.con.types.xml.gz`, $\underline{e}$`.con.constraints.xml` containing respectively: the name $\underline{e}$, type E, list of universe parameters $(y)^*$ and list of section variables that the constant depends on; the definiens E' of the constant, if given; the type of all subterms of E and E' that inhabits a proposition, together with pointers to the subterms in the *uid*`.con.xml.gz` and *uid*`.con.body.xml.gz` files; the list of all constraints between universes that need to hold for the constant to be well typed.
- Case $\mathcal{F}$ **Variable** $x : E$: the output is the same of the constant case, but `.con` is replaced by `.var` and the `.con.body.xml.gz` file is missing because a variable has no definiens
- Case $\mathcal{F}$ (**Inductive** | **CoInductive**) $(\underline{e}@\{(y)^*\} : E := (\underline{e}@\{(y)^*\} : E)^*)^+$. Let $\underline{e}1$ be the $\underline{e}$ of the first type defined in the block. We generate the $\underline{e}1$`.ind.xml.gz`, $\underline{e}0$`.ind.types.xml.gz` and $\underline{e}1$`.ind.constraints.xml` files. The former contains all the logical information in the declaration. The latter contains both the set of constraints between universes that need to hold for the constant to be well typed and the variance information for every universe parameter.

The output is backward compatible with [Sac01], up to the changes to the exportation of terms inside the files and with the exceptions of universe polymorphism

that was not there. In particular `.constraints.xml.gz` are new as well as the universe parameters $(y)^*$.

Terms The encoding of terms to XML is rather straightforward: all the data available is just turned into XML elements and attributes. Moreover, at the level of terms the differences with the 2003 plugin are minimal: primitive record projection have been added to Coq; universe polymorphism has been added so that occurrences of constants are now applied to universes; finally we now export a precise description of universes whereas before we were just mapping to `Type` all universes — universes were handled differently by Coq at the time. Therefore we have just applied minimal changes to the DTD we were using in 2003.

Derived information Once the XML files are exported, we use python and bash scripts to add to the repository additional files containing data that can be inferred processing the XML files, but that require time to generate. This data is currently considered experimental and stored either in Comma Separated Values (CSV) or Resource Description Framework (RDF) formats.

In particular a `graph.csv` file collects all logical dependencies between variable/constant/inductive types in the form of "URI1 occurs in URI2". Additional dependencies are generated lifting the previous ones to directories: DIR2 depends on DIR1 iff DIR1 contains an object URI1 that occur in URI2 that is contained in DIR2. These dependencies can be used to automatically generate graphical representations of dependency graphs for libraries at different level of granularities (files, modules, objects).

The RDF files store many more additional triples, comprising the ones required by the Whelp search engine [AGS+04] and the ones that belong to the Upper Library Ontology (ULO) [CKM+].

4 Statistics and final considerations

The new XML exportation plugin has been already applied to the 49 Coq libraries that have an opam package and that currently compile with the latest version of Coq (8.9.0), generating a large dataset of 1,235,934 compressed XML files organized in 18,780 directories that require 17GB on disk. The library, which is already one of the largest dataset for formal libraries, consists in 16,322 variable declarations, 159,617 constants (143,165 of them have a definiens) and 2,712 blocks of mutual (co)inductive type declarations. Everything is organized in 1,889 theory files, 3,931 sections, 2,368 module definitions, 52 module declarations and 387 module type definitions. Among the constants there are 47,978 user declared theorems and we compute the statement of their 7,426,140 atomic proof steps. More packages are likely to be exported in the next weeks since the Coq developers are currently working on porting more Coq libraries to 8.9.0.

We hope that a large number of researchers will consider the library a good dataset to experiment with Mathematical Knowledge Management in the large, and will be able to build interesting tools over it. Search engines are an easy

example, but we already have interesting proof of concepts since the early 2000s (see for example [AGS+04,GS03] that worked on the data exported by the old version of the plugin). We are more interested into more recent tools like hammers, automatic theory aligners and tools able to perform automatic refactoring, which are all promising to deliver to users tools that have an immediate value for the daily formalization activity.

Finally, we envision the generated Git packages as an open repository for data and we encourage all researchers that generate data derived from our XML files to contribute by committing their additional data to the repositories.

References

ABD03. Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors. *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*. Springer, 2003. doi:10.1007/3-540-36469-2.

AGS+04. Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2004. doi:10.1007/11617990_2.

ARST11. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *23rd International Conference on Automated Deduction, CADE-23, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011. doi:10.1007/978-3-642-22438-6_7.

ASTZ06. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2006. doi:10.1007/978-3-540-74464-1_2.

AW02. Andrea Asperti and Bernd Wegner. MoWGLI - An approach to machine-understandable representation of the mathematical information in digital documents. In *Electronic Information and Communication in Mathematics, ICM 2002 International Satellite Conference, Beijing, China, August 29-31, 2002, Revised Papers*, volume 2730 of *Lecture Notes in Computer Science*, pages 14–23. Springer, 2002. doi:10.1007/978-3-540-45155-6_2.

CK18. Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018. doi:10.1007/s10817-018-9458-4.

CKM+. Andrea Condoluci, Michael Kohlhase, Dennis Müller, Florian Rabe, Claudio Sacerdoti Coen, and Makarius Wenzel. Relational data across mathematical libraries. Submitted to CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-ulo.pdf.

DT. Gilles Dowek and François Thiré. Logipedia: a multi-system encyclopedia of formal proofs. URL: http://www.lsv.fr/~dowek/Publi/logipedia.pdf.

GA16. Emilio Jesús Gallego Arias. SerAPI: Machine-friendly, data-centric serialization for Coq. Working paper or preprint, October 2016. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408.

GS03. Ferruccio Guidi and Irene Schena. A query language for a metadata framework about mathematical resources. In Asperti et al. [ABD03], pages 105–118. doi:10.1007/3-540-36469-2_9.

ISA+16. Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. DeepMath - Deep sequence models for premise selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016. URL: http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.

KS06. Michael Kohlhase and Ioan Sucan. A search engine for mathematical formulae. In *Artificial Intelligence and Symbolic Computation, 8th International Conference, AISC 2006, Beijing, China, September 20-22, 2006, Proceedings*, volume 4120 of *Lecture Notes in Computer Science*, pages 241–253. Springer, 2006. doi:10.1007/11856290_21.

KWP99. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999. doi:10.1007/3-540-48256-3_11.

MGK+17. Dennis Müller, Thibault Gauthier, Cezary Kaliszyk, Michael Kohlhase, and Florian Rabe. Classification of alignments between concepts of formal mathematical systems. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2017. doi:10.1007/978-3-319-62075-6_7.

MRS. Dennis Müller, Florian Rabe, and Claudio Sacerdoti Coen. The Coq library as a theory graph. Submitted to CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-coq.pdf.

RK13. Florian Rabe and Michael Kohlhase. A scalable module system. *Inf. Comput.*, 230:1–54, 2013. doi:10.1016/j.ic.2013.06.001.

Sac01. Claudio Sacerdoti Coen. Project IST-2001-33562 MoWGLI. Technical Report D2.a Exportation Module, Information Society Technologies (IST) Programme, 2001. URL: http://mowgli.cs.unibo.it/misc/deliverables/transformation/D2a_exportation_module/report.pdf.

Sac03. Claudio Sacerdoti Coen. From proof-assistants to distributed libraries of mathematics: Tips and pitfalls. In Asperti et al. [ABD03], pages 30–44. doi:10.1007/3-540-36469-2_3.

Sac04. Claudio Sacerdoti Coen. *Mathematical knowledge management and interactive theorem proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5, 2004. URL: http://www.cs.unibo.it/~sacerdot/tesidott/thesis.ps.gz.

Sac10. Claudio Sacerdoti Coen. Declarative representation of proof terms. *J. Autom. Reasoning*, 44(1-2):25–52, 2010. doi:10.1007/s10817-009-9136-7.

Tea19. The Coq Development Team. The Coq Proof Assistant, version 8.9.0, January 2019. doi:10.5281/zenodo.2554024.