The Coq Library as a Theory Graph

Dennis Müller, Florian Rabe, and Claudio Sacerdoti Coen[3]

[1] University Erlangen-Nuremberg
[2] LRI Paris
[3] University of Bologna

Abstract. Representing proof assistant libraries in a way that allows further processing in other systems is becoming increasingly important. It is a critical missing link for integrating proof assistants both with each other or with peripheral tools such as IDEs or proof checkers. Such representations cannot be generated from library source files because they lack semantic enrichment (inferred types, etc.) and only the original proof assistant is able to process them. But even when using the proof assistant's internal data structures, the complexities of logic, implementation, and library still make this very difficult.
We describe one such representation, namely for the library of Coq, using OMDoc theory graphs as the target format. Coq is arguably the most formidable of all proof assistant libraries to tackle, and our work makes a significant step forward.
On the theoretical side, our main contribution is a translation of the Coq module system into theory graphs. This greatly reduces the complexity of the library as the more arcane module system features are eliminated while preserving most of the structure. On the practical side, our main contribution is an implementation of this translation. It takes the entire Coq library, which is split over hundreds of decentralized repositories, and produces easily-reusable OMDoc files as output.

1 Introduction and Related Work

Motivation A critical bottleneck in interactive theorem proving is data sharing, both between proof assistants and between proof assistants and related tools. The general situation is in starch contrast to the global push for FAIR data practices [oFD18], i.e., findable, accessible, interoperable, and reusable sharing of data. Currently, for example, any reuse of a library must go through the respective proof assistant. Thus, any novel idea is typically limited to the implementation framework and data flows provided by the proof assistant; and out-of-the-box experiments that by-pass the proof assistant are expensive, often prohibitively so. This limitation is particularly relevant as proof assistants are becoming more mature and many challenges are shifting from prover design to library management tasks like refactoring, reuse, search, and user interfaces.

For multiple reasons, Coq is the most formidable target for library sharing among all proof assistants: Firstly, the logic of Coq is arguably the most complex among all major proof assistants. This applies not only to the core logic, but also to the processing pipeline from user-visible source to kernel representation and the library-building features of module system, sections, etc. Secondly, the code base of the Coq system has grown to a point where it is very hard for outsiders to navigate it. Thirdly, Coq has been so successful that its library is now so vast that it is non-trivial to even understand its extent — it is now split over several hundred repositories with non-uniform build processes.

Contribution Our overall goal is making the Coq library easier to access, search, interoperate with, and reuse. Even though we make a substantial first step, comprehensive progress on this front will take years. Concretely, in this paper, we translate the high-level structure of the Coq library that is visible to the Coq kernel (including modules and sections but not records or type classes) into the MMT language [RK13] for theory graphs. Theory graphs are an attractive target format because they allow preserving most of the library structure while being significantly simpler. Besides being designed to maintain theory graphs, MMT also provides a flexible logical framework that allows us to define the logical syntax of Coq. Thus, our MMT theories include all information in the Coq kernel including universes, inductive types, proof terms, and termination of recursive functions.

We translate all 49 Coq packages that are distributed via `opam` (a package manager originally designed for ocaml software) and that compile with the latest version of Coq (8.9.0). These comprise more than 383,500 logical declarations, making ours one of the largest proof assistant library translations ever and the largest for Coq.

Related Work Multiple ad hoc exports between different proof assistant libraries have been achieved. The general design of instrumenting the proof assistant kernel to export the library as a trace of checking it was first applied in [OS06]. This has proved to be the only feasible design, and all major later exports including ours employed variants of it. For example, Coq was the target format in [KW10].

Exports specifically into MMT were achieved for Mizar in [IKRU13], HOL Light in [KR14], PVS in [KMOR17], and very recently for Isabelle in not-yet published work. This overall line of research was presented in [KR16].

Similarly, to the MMT research, proof assistant libraries have been exported into Dedukti [BCH12]. Coqine is the tool used for translating Coq to Dedukti, and while MMT exports focus on preserving the original structure, Coqine focuses on reverifying proofs. Unlike our logic definition, Coqine includes a formalization of the typing rules in order to type check the export. In order to make this feasible, this translation eliminates several features of the Coq logic so that the typing rules become simpler. Our export, on the contrary, makes the dual trade-off, covering the entire logic at the expense of representing the typing rules. Concretely, the original version [BB12] covered most of the small standard library, using simplifications like collapsing the universe hierarchy. A later reimplementation [Ass15] used a more faithful representation of the logic. But it still omitted several language features such as modules, functors and universe polymorphism and therefore could not translate a significant part of the library.

[CK18] develops a translation of Coq into first-order logic in order to apply automated provers as hammers. Like Coqine, it only covers a subset of the language. It can in principle be used to translate the entire library, but that would have very limited use: Indeed, due to the nature of this application, it is actually *beneficial* to ignore all modular structure and even to not care about soundness, for example to collapse all universes.

Overview The structure of our paper follows the three major parts of building a library: the core logical language, the language for library building, and the library itself. Sect. 3 describes the representation of the Coq logics (As we will see, Coq technically provides a few slightly different logics.) in MMT. This results in a few manually written MMT theories. Sect. 4 specifies how the library language features of Coq can be mapped to MMT theories and morphisms. And Sect. 5 describes the implementation that translates the Coq library into MMT. We recap the relevant preliminaries about Coq and MMT in Sect. 2, and we discuss limitations and future work in Sect. 6.

Acknowledgments The authors were supported by DFG grant RA-18723-1 OAF and EU grant Horizon 2020 ERI 676541 OpenDreamKit. Tom Wiesing helped with scripting for automatically creating the necessary `mathhub.info` repositories and pushing to them.

2 Preliminaries

2.1 Coq

We give only an extremely dense explanation of the Coq language and refer to Appendix A and [Coq15] for details. We use a **grammar** for the *abstract* syntax seen by the Coq kernel (Fig. 1) because that is the one that our translation works with. Even though we do not have space to describe all aspects of the translation in detail, we give an almost entire grammar here in order to document most of the language features we translate. A slightly more comprehensive grammar is presented in the companion paper [Sac19]: the omitted features do not pose additional problems to the translation to MMT and are omitted to simplify the presentation.

The Coq library is organized hierarchically into (from high to low) packages, nested directories, files, nested modules, and nested sections, where "nested" means that multiple levels of the same kind may be nested. Modules and sections are optional, i.e., logical declarations can occur in files, modules, or sections. When forming base logic expressions E, universes U, module expressions M, and module type expressions T, declarations can be referred to by **qualified identifiers** e, u, m, resp. t formed from

1. The root identifier that is defined by the Coq package. 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.

$$
\begin{array}{lll}
\texttt{decl} & ::= & \text{— \textit{base logic declarations}} \\
 & & \underline{e}@\{y^*\} : E[\,:= E] \\
 & | & \textbf{Universe}\ \underline{u} \\
 & | & \textbf{Constraint}\ u(< \mid \leq \mid =)u \\
 & | & (\textbf{Inductive} \mid \textbf{CoInductive})\ (\underline{e}@\{y^*\} : E := (\underline{e}@\{y^*\} : E)^*)^* \\
 & & \text{— \textit{section declarations and variables in sections}} \\
 & | & \textbf{Section}\ \underline{s}\ := \texttt{decl}^* \\
 & | & \textbf{Variable}\ x : E \\
 & | & \textbf{Polymorphic Universe}\ y \\
 & | & \textbf{Polymorphic Constraint}\ (u \mid y)(< \mid \leq \mid =)(u \mid y) \\
 & & \text{— \textit{module (type) declarations}} \\
 & | & \textbf{Module Type}\ \underline{m}\ (\underline{m} : T)^* <: T^*\ := (T \mid \texttt{decl}^*) \\
 & | & \textbf{Module}\qquad \underline{t}\ (\underline{m} : T)^*\ [\,: T]\ <: T^*\ [\,:= (M \mid \texttt{decl}^*)] \\
E & ::= & \text{— \textit{base logic expressions}} \\
 & & e@\{U^*\} \mid x \mid \texttt{Prop} \mid \texttt{Set} \mid \texttt{Type}@\{U\} \mid \mathit{\Pi} x : E.E \mid \lambda x : E.E \mid E\ E \\
 & | & \texttt{Match}\, e\, E\, E\, E^* \mid (\texttt{Fix} \mid \texttt{CoFix})\, \mathbb{N}\, (\underline{e} : E := E)^* \mid \texttt{let}\ x : E := E\ \texttt{in}\ E \\
 & | & E.\mathbb{N} \mid (E : E) \\
U & ::= & \text{— \textit{universes}} \\
 & & u \mid y \mid \max U\, U \mid \texttt{succ}\, U \\
T & ::= & \text{— \textit{module type expressions}} \\
 & & [!]\ t\ m^* \mid T\ \texttt{with}\ e' := E \mid T\ \texttt{with}\ m' := M \\
M & ::= & \text{— \textit{module expressions}} \\
 & & [!]\ m\ m^* \\
x, y & ::= & \text{variables for term, universe respectively} \\
e, u, m, t, s & ::= & \text{qualified identifiers of expressions, universes, modules, module types, sections} \\
e', m' & ::= & \text{relative qualified identifiers of expressions, modules} \\
\underline{e}, \underline{u}, \underline{m}, \underline{t}, \underline{s} & ::= & \text{fresh (unqualified) identifiers}
\end{array}
$$

Fig. 1. Coq Kernel Grammar

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

Expressions are the usual λ-calculus expressions with dependent products $\mathit{\Pi} x : \mathit{term}.\mathit{term}$ (used to type λ-abstractions), let binder, `let...in`, 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 hierachy** (a directed acyclic graph) with consistency constraints of the form $U(< \mid \leq)U'$.

Module types and modules 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 specify what declarations are public in a module. 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 $\underline{m}$ that is declared in T and does so with a module type that conforms to the module type (= the set of public declarations) of $\underline{m}$ 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). 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. A module may also be abstract, i.e., have neither body nor definiens. The $<:$ and $:$ operators may be used to attach conformation conditions to a module (type), and we will explain their semantics in Sect. 4.2.

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. Conformation induces a notion of subtyping between module and functor types. Coq treats functor types contravariantly and allows for higher-order functors. However, from our experiments, it seems that this feature is never used in any of the libraries we exported from Coq.

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 may be used to subdivide files and module (type)s. 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 normal declarations are $\lambda\Pi$-abstracted over all **Variable**/**Polymorphic** declarations.

2.2 MMT

MMT aims at being a universal representation language for formal systems. Its syntax was designed carefully to combine simplicity and expressivity. Fig. 2 gives the fragment needed for Coq, and we refer to [RK13,Rab17] for details.

$$
\begin{array}{lll}
\texttt{decl} & ::= & \textbf{Theory}\ l =^{[E]} \texttt{decl}^* \\
 & \mid & \textbf{Morph}\ l : E \to E =^{[E]} \texttt{decl}^* \\
 & \mid & \textbf{include}\ E \\
 & \mid & l[:E][=E] \\
 & \mid & \textbf{Rule}\ \text{Scala object} \\
E & ::= & g \mid g?l \mid x \mid g?l((x[:E][=E])^*, E^*) \\
g & ::= & URI?l \mid g/l \\
l & ::= & \text{local identifiers}
\end{array}
$$

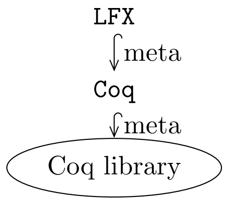
Fig. 2. Mmt Grammar

An expression **Theory** $l =^{[E]} B$ introduces a named **theory** l with body B and (optional) meta-theory E. In the simplest case, nested theories create a tree of declarations, whose leafs are **constant** declarations $l : E_1 = E_2$ introducing a local identifier l with optional type E_1 and definiens E_2.

Named theories have global identifiers g of the form $g = NS?l_1/\ldots/l_n$ where NS is the namespace URI assigned by the containing source file, and the l_i are the local identifier of the nested theories (i.e. l_n is the local identifier of the theory itself, l_{n-1} is the local identifier of the containing theory etc.).

Every constant has a unique URI of the form $g?l$ where g is the global identifier of the containing theory and l is the local identifier of the constant. In an expression E, every theory or constant is always referenced via its global identifier.

Every theory g induces a set of **expressions** E formed from identifiers, bound variables x, and composed expressions $g?l(C, E_1, \ldots, E_n)$, where C is a (possibly empty) list of variables $x[(:E')][(=E'')]$ considered bound in the subexpressions E_i. The semantics of expressions is signaled by the chosen constructor $g?l$, which is usually a constant declared in the meta-theory.



Meta-theories yield the language-independence of Mmt: The meta-theory L of a theory t defines the language in which t is written. For example, the diagram on the right indicates the form of the theory graph we built in this paper: all theories in the Coq library will be translated to Mmt theories with meta-theory `Coq`, which in turn has meta-theory `LFX`. `LFX` is an extension of the logical framework LF [HHP93] that is strong enough to define the Coq logic in a theory `Coq`. The semantics of `LFX` itself is obtained by declaring **rules**: these are Scala objects that are injected dynamically into the Mmt kernel. We refer to [Rab18] for the general mechanism and the definition of LF. Most importantly, `LFX` declares

1. 5 constants for forming the composed expressions `type`, $A \to B$, $\Pi x : A.B$, $\lambda x : A.t$ and function applications $f(a_1, \ldots, a_n)$;
2. syntax rules that render the composed expression `LFX?lambda`$(x : A, t)$ as $\lambda x : A.t$; and
3. about 10 typing rules for the LF type system.

The MMT theory graphs arise from adding **morphisms** $m : s \to t =^M B$. Here m is a morphism that maps the meta-theory of s to t, and the body B must contain a defined constant $g?c = E$ for every s-constant $g?c$ and some t-expression E. Then the homomorphic extension of m maps any s-expression to a t-expression in a way that, critically, preserves all judgments, e.g., if $\vdash_s e : E'$, then $\vdash_t m(e) : m(E')$. Such morphisms have theorem-flavor and are used to represent language translations and refinement or interpretation theorems. Alternatively, include declarations **include** s in a theory t are used to create morphisms $s \to t$ that hold by definition: their semantics is that all s-constants are also visible to t, which implies that the identity map of constants is a theory morphism. Include morphisms are used to represent inheritance and extension relations between theories and are depicted in diagrams as $s \hookrightarrow t$. Expressions over a theory t can use all constants declared in t, in the meta-theory of t, a parent theory of t, or in theories included into t.

3 Defining the Coq Base Logics in a Logical Framework

We define an MMT theory `Coq` that defines the base logic of Coq. This theory will occur as the meta-theory of all MMT theories generated from files in the Coq library (except when flags are used, see below). The theory `Coq` is available at `https://gl.mathhub.info/Coq/foundation/blob/master/source/Coq.mmt`. We briefly describe it in the sequel and refer to Appendix B for details.

Expressions Due to lack of space, we only present the encoding of the PTS fragment of Coq, i.e. we omit `let...in`, projections, inductive types, pattern matching and (co)fixpoints. The encoding is quite straightforward.

Theory $\mathtt{Coq} =^{\mathtt{LFX}}$
$\mathtt{univ} : \mathtt{type} \quad \mathtt{max} : \mathtt{univ} \to \mathtt{univ} \to \mathtt{univ} \quad \mathtt{succ} : \mathtt{univ} \to \mathtt{univ}$
$\mathtt{expr} : \mathtt{type} \quad \mathtt{Prop} : \mathtt{expr} \quad \mathtt{Set} : \mathtt{expr} \quad \mathtt{Type} : \mathtt{univ} \to \mathtt{expr}$
$\Pi : \mathtt{expr} \to (\mathtt{expr} \to \mathtt{expr}) \to \mathtt{expr} \quad \lambda : \mathtt{expr} \to (\mathtt{expr} \to \mathtt{expr}) \to \mathtt{expr}$
$\mathtt{app} : \mathtt{expr} \to \mathtt{expr} \to \mathtt{expr}$
$\mathtt{hastype} : \mathtt{expr} \to \mathtt{expr} \to \mathtt{type}$
$\mathtt{exprOfType} : \mathtt{expr} \to \mathtt{type} = \lambda e : \mathtt{expr}.\{x : \mathtt{expr} | \mathtt{hastype}\, x\, e\}$

Our representation of the syntax is a Curry-style encoding, in which all expressions have the same LF type and the binary typing judgment between expressions is formalized by separate judgment `hastype`. We do not give any typing rules here, but they could now be added in a straightforward way (except of course that Coq's typing rules are very complex and doing so is correspondingly time consuming). There are alternative Church-style encodings, where a Coq expression of type E is represented as an LF-expression of type $\mathtt{exprOfType}\, E$ for an operator $\mathtt{exprOfType} : \mathtt{expr} \to \mathtt{type}$. These would be preferable because they allow declaring Coq identifiers as, e.g., $\mathtt{zero} : \mathtt{exprOfType}\, \mathtt{Nat}$ instead of erasing their type by using a declaration $\mathtt{zero} : \mathtt{expr}$. This is also why they are used in Coqine [BB12,Ass15] to formalize the calculus of constructions in Dedukti. However, Church encodings introduce so much representational overhead that they would make the translation of the entire Coq library infeasible. To gain the best of both worlds, we use predicate subtyping to define the $\texttt{exprOfType}\,E$ as the subtype of `expr` containing those x for which the judgment $\texttt{hastype}\,x\,E$ holds.

With these declarations in place, we can for example translate the definition of a universe-polymorphic identity function

$$\texttt{id@}\{y\} : \mathit{\Pi} A : \texttt{Type@}\{y\}.A \to A \;:=\; \lambda A : \texttt{Type@}\{y\}.\lambda x : A.x$$

of Coq to the following Mmt definition over the theory `Coq`

$$\begin{aligned}\texttt{id} : \mathit{\Pi} y : \texttt{univ}.\ \texttt{exprOfType}\left(\mathit{\Pi}\,(\texttt{Type}\,y)\,(\lambda A.A \to A)\right)\\ = \lambda y.\ \lambda\,(\texttt{Type}\,y)\,(\lambda A.\lambda\,A\,(\lambda x.x))\end{aligned}$$

This captures all relevant information of the Coq definition with minimal representational overhead. Note how Coq's $\mathit{\Pi}$ and λ-binding are represented using LF higher-order abstract syntax, whereas universe polymorphism is represented directly using LF's binders.

Logic Variants Maybe surprisingly, Coq does not actually use a single logic: it offers flags that allow choosing variants of the type theory. Two flags are of particular importance as they are required by some of the libraries:

– `-impredicative-set` changes the typing rule of the dependent function space $\mathit{\Pi}$ so that the type of functions that takes in input an inhabitant of a large universe and return a set is still a (small) set instead of being a larger type; the flag is inconsistent when assumed together with any axiom of choice and classical logic
– `-type-in-type` squashes all universe except `Prop` and `Set` into the single universe `Type`. The resulting inconsistency `Type : Type` is acceptable and useful in some applications, e.g., those that focus on computation rather than deduction and need the possibility to write non terminating functions.

All variants can be formalized similarly using slightly different typing rules. As we omit the typing rules anyway, we simply create theories `Coq`, `ImpredicativeCoq`, and `InconsistentCoq`, all of which include `CoqSyntax` and then contain placeholder comments for the typing rules. When extracting the library from Coq, we record the flags used to compile each Coq file. Depending on that information, we choose one of the above three theories as the meta-theory of the Mmt-theory that is the translation of that file.

4 Representing the Coq Structuring Language in MMT

4.1 Overview

Fig. 3 gives an overview of our translation. Above the file level, our translation preserves the structure of Coq exactly: every directory or file in a Coq package is

Coq	MMT
package	namespace
directory	namespace
file	file that declares a theory
module type	theory
module	theory
visibility of a module m to p	inclusion morphism $m \hookrightarrow p$
module typing $M : T$	morphism $T \to M$
module conformation $M <: T$	morphism $T \to M$
module type conformation $T <: T'$	morphism $T' \to T$
section	theory
variable in a section	constant
any base logic declaration	constant

Fig. 3. Overview of the Translation

translated to a corresponding directory resp. file in an MMT archive. Therefore, all Coq directories result in MMT namespace URIs.

A **qualified identifier** consisting of root r, directories $d_1, \ldots, d_r$, file name f with extension `v`, modules (types) $m_1, \ldots, m_s$, and name n is translated to the MMT URI `coq`$:/r/d_1/\ldots/d_r?f/m_1/\ldots/m_s?n$. Note that the MMT URI makes clear, which parts of the qualified identifier are directory, file, or module (type) names without having to dereference any part of the URI.

The only subtlety here is that we translate every Coq source file to a theory. Effectively, we treat every Coq file f.`v` in directory D like the body of a module of name f; and we translate it to an OMDoc file f.`omdoc` containing exactly one MMT theory with URI $D?f$.

If we translated files to namespaces instead of theories, the above MMT URI would be `coq` $: /r/d_1/\ldots/d_r/f?/m_1/\ldots/m_s?n$. We would have preferred this, but it is not possible: In Coq, base logic declarations may occur directly in files whereas MMT constants may only occur inside theories. Thus, we have to wrap every Coq source file into an MMT theory. This is inconsequential except that we have to add corresponding include declarations in MMT: for every file f' that is referenced in a file f, the resulting MMT theory must include the MMT theory of f'. Fortunately, this information is anyway stored by Coq so that this is no problem.

In the sequel, we write $\bar{i}$ for the MMT translation of the Coq item i except that, if i is a Coq identifier, we write i in MMT as well if no confusion is possible. We omit the translation of base logic expressions and refer to Appendix C for the translation of sections.

4.2 Modules and Module Types

We translate all files and module (type)s to MMT theories. Thus, the parent p of every module (type), which is either a file or a module (type), is always translated to an Mmt theory; and every module (type) with parent p is translated to an Mmt theory nested into $\overline{p}$. Overall, Coq's tree of nested module (type) and constant declarations is translated to an isomorphic tree of nested Mmt theories and constants, augmented with the theory morphisms induced by module type conformation (explained below).

We first consider the non-functor case and generalize to functors in Sect. 4.3. An algebraic module (type) is translated by first computing its explicit representation as an interactive one, according to the meta-theory of Coq. In this way we only have to consider the interactive case and we lift from the user the burden of understanding the intricacies of algebraic module (type) resolution (e.g. the complex semantics given by "!" annotations, or the issue of generativity for functors application).

Module and Module Types as Theories So let us consider an interactive module type **Module Type** $t <: T_1 \dots T_n \ := \ B$. We translate it to an Mmt theory $\overline{t}$ whose body arises by declaration-wise translation of the declarations in B. However, we have to treat universes specially because Coq maintains them globally: all universes and constraint declared in B are not part of $\overline{t}$ and instead treated as if they had been declared at the beginning of the containing source file. We discuss the treatment of $<:$ attributions below.

A module is translated in exactly the same way as a module type. The semantic difference between modules and module types is that a module m, once closed, exports all declarations in its body to its parent p. We capture this difference in Mmt exactly by additionally generating an include declaration **include** $\overline{m}$ after the theory $\overline{m}$, which makes $\overline{m}$ available to $\overline{p}$.

It may be tempting to alternatively translate module types t to theories $\overline{t}$ and a module $m : t$ to a theory morphism $\overline{m} : \overline{t} \to \overline{p}$. This would elegantly capture how every module is an implementation of the module type by providing definitions for the abstract declarations in t. But that is not possible because Coq allows abstract fields even in modules, and such modules would not induce Mmt theory morphisms. A maybe surprisingly example is the following, which is well-typed in Coq:

$$\begin{array}{l} \textbf{Module Type}\ s \ := \ e : \texttt{False} \\ \textbf{Module}\ m : s \ := \ e' : \texttt{False},\ e : \texttt{False} := e' \\ x : \texttt{False} := m.e \end{array}$$

Here the abstract declaration of e' in the module m is allowed even though it is used to implement the interface s of m.

Conformation as a Theory Morphism Now we translate the attributions $<: T_i$ on a module (type) and the attributions $: T$ on a module. Our translation does not distinguish modules and module types, and if multiple attributions $<: T_i$ are present, they are translated individually. Thus, we only need to consider the cases $m <: T$ and $m : T$. In both cases, our translation consists of a morphism $\overline{m^*}$ from $\overline{T}$ to $\overline{m}$ that witnesses that m conforms to T.

Inspecting the grammar, we see that T has normal form $(t\, m_1\, \ldots\, m_r)\, \texttt{with}\, k_1 := K_1\, \ldots\, \texttt{with}\, k_s := K_s$, where $k := K$ unifies the cases of constant and module instantiations. As we will see below, if t is a module type functor (i.e., if $r! = 0$), its module parameters $x_1 : T_1, \ldots, x_r : T_r$ are translated as if t were not functorial and the $x_i : T_i$ were abstract modules in the body of t. Accordingly, we treat T in the same way as $t\, \texttt{with}\, x_1 := m_1\, \ldots\, \texttt{with}\, x_r := m_r\, \texttt{with}\, k_1 := K_1\, \ldots\, \texttt{with}\, k_s := K_s$, and therefore we can restrict attention to the case $r = 0$.

We know that t is translated to a theory $\overline{t}$, and if T is well-typed, recursively translating the K_i already yields a partial theory morphism φ from t to $\overline{p}$. Because $\overline{m}$ is a theory nested into $\overline{p}$, φ is also a morphism into $\overline{m}$. It remains to extend φ with assignments for the remaining declarations of $\overline{t}$. Now we observe that if m conforms to T in Coq, we obtain a well-typed MMT theory morphism $\overline{m^*}$ by extending φ with assignments $\overline{t}?k := \overline{m}?k$ for every such name k. (The converse is also true if we add typing rules to `Coq` that adequately capture the typing relation of base logic expressions.)

$$\begin{tikzcd} \overline{t} \arrow[r, "\overline{m^*}"] & \overline{m} \arrow[r, "r"] & \overline{m.\texttt{impl}} \end{tikzcd}$$

For <: attributions, this is all we have to do. But an attribution $m : T$ is stronger than an attribution $m <: T$. It additionally restricts the interface of m to what is declared in T. Therefore, we have to do a little bit more in the case $m : T$ as shown on the right:

1. We rename the theory $\overline{m}$ to $\overline{m.\texttt{impl}}$.
2. We create a second theory $\overline{m}$ that is a copy of $\overline{t}$ where all qualified names use m in place of t.
3. We create a morphism $r : \overline{m} \to \overline{m.\texttt{impl}}$ that maps every name of $\overline{m}$ to itself.
4. We create the renaming morphism $\overline{m^*}$ with codomain `m` in the same way as for the case $m <: T$. The morphism just performs the renaming since $\overline{m}$ and $\overline{t}$ only differs on names.

Module Type s := f : `Nat`
Module Type t :=
  Module $m : s$:= f : `Nat` := 0
Module $n : t$:=
  Module m := f : `Nat` := 1

The : attributions of Coq are peculiar because $\overline{m.\texttt{impl}}$ can never be referenced again — the morphism r can be seen as a dead end of the theory graph. In fact, trying to understand this part of the translation made us realize the following curiosity about the Coq module system. Consider the well-typed example on the right, where we use indentation for scoping. The attribution $m : s$ in the module type t hides the definition of the field f in the module m. Because that definition is never considered again, the module n can supply a different definition for f later on.

Indeed, the Coq kernel imperatively throws away $\overline{m.\texttt{impl}}$ after checking it. When f declares a logical axiom instead of a type like **Nat**, the behaviour is somewhat more intuitive: if we only care that a definition (i.e., proof) exists, it is fine to give two different ones in different places. But this treatment is markedly different from analogous features of other languages: In object-oriented programming, n would not be allowed to redefine m because the definition of f is still inherited even if remains inaccessible. Similarly, in theory graphs with hiding [MAH06,CHK+12], the model n of t would be required to implement e in a way that is consistent with the hidden definition in t.

4.3 Functors

Declaring Functors In many ways the parameters of an interactive module (type) can be treated in the same way as the declarations in its body B. Indeed, the declaration **Module** (**Type**) $t(x_1 : T_1, \ldots, m_r : T_r) := B$ is well-typed iff **Module** (**Type**) $t :=$ **Module** $x_1 : T_1, \ldots,$ **Module** $m_r : T_r, B$ is.

This motivates what we call the *covariant* translation of functors, which we employ: parameters of interactive modules or module types are translated as if they occurred as abstract module declarations at the beginning of the body. Thus, the two variants of t above are translated to the exact same Mmt theory. The resulting diagram is shown on the right. Note that the theories $\overline{t/x_i}$ are nested into $\overline{t}$ and additionally included into $\overline{t}$. We also add metadata to the declarations of the $\overline{t/x_i}$ to record the fact that they used to be functor parameters. Algebraic module (type) functors and <: and : attributions are handled in the same way as in Sect. 4.2.

$$\begin{tikzcd}
\overline{T_1} \arrow[d, "\overline{t/x_1^*}"'] & \cdots & \overline{T_r} \arrow[d, "\overline{t/x_n^*}"] \\
\overline{t/x_1} \arrow[dr, hook] & & \overline{t/x_r} \arrow[dl, hook'] \\
& \overline{t} &
\end{tikzcd}$$

Technically, only a *contra*variant translation that translates functors to theory-valued functions would capture the semantics of functors adequately. For example, the covariant translation of **Module** $m(x_1 : T_1, \ldots, m_r : T_r) := B$ results in the same diagram as above with an additional inclusion morphism from $\overline{m}$ into the parent p of m. Thus, the theories $\overline{t/x_i}$ become falsely included into $\overline{p}$. More formally:

1. the covariant translation preserves well-typedness only if the library does not rely on Coq's contravariant rule for functor type subtyping, which is the case for all the libraries exported so far.
2. the covariant translation does not reflect well-typedness.

However, considering that the Coq library is already well-typed and that the covariant translation is so much simpler, that is sufficient for many practical applications.

Applying Functors Coq functor application may be partial and curried. Thus, it is sufficient to restrict attention to $r = 1$. So consider a module type declaration **Module Type** $t(x : T) := B$ and a module $m : T$. We have to define the translation of the module type $t(m)$, whose semantics is determined in Coq by substituting m for x in B. We want the translation to be compositional, i.e., defined in terms of the theory $\overline{t}$ arising from the translation of t and the morphism $\overline{m}^* : \overline{t} \to \overline{m}$ arising from the translation of

$$\begin{tikzcd}
\overline{T} \arrow[r, "\overline{t/x^*}"] \arrow[d, "\overline{m^*}"] & \overline{t/x} \arrow[r, hook] & \overline{t} \arrow[dd] \\
\overline{m} \arrow[d, hook] & & \\
\overline{p} \arrow[rr, hook] & & \overline{t(m)}
\end{tikzcd}$$

m, as in the diagram on the right. As defined above, the functor t is translated to a theory $\bar{t}$ with a nested theory $\overline{t/x}$ that conforms to $\overline{T}$ as well as an include of $\overline{t/x}$. Let p be the Coq file or module (type) in which $t(m)$ is well-typed; thus, $\overline{p}$ is a theory that includes $\overline{m}$.

This situation is well-known in MMT theory graphs: to translate $t(m)$, we have to create a new theory nested into $\overline{p}$ such that the diagram is a pushout. The canonical choice of a pushout [Rab17,CMR16] amounts to copying over all declarations in $\bar{t}$ except for replacing all occurrences of x by the homomorphic translation along $\overline{m^*}$. This yields the same theory as translating the flattening of $t(m)$.

5 Translating the Coq Library

Our translation is implemented in two steps. Firstly, Coq is instrumented via kernel hooks to export the internal kernel data structures into Coq-near gzipped XML files. This part of the translation is described in detail in [Sac19]. Secondly, we read these XML files into MMT and translate them to MMT data structure, which we then write out to disk as OMDoc files. (Actually, we use xz-compressed OMDoc files because the uncompressed files would be too large.)

This separation into a Coq-export and an MMT-import may seem inefficient. But this design has proved very successful in the past [IKRU13,KR14,KMOR17]. Moreover, it allows separating the formidable practical task of exporting anything from the theoretical task of specifying the translation.

Notably, the whole export of the 49 opam packages for Coq libraries that currently compile with Coq 8.9.0 (recently released) comprises about 1.3 million XML files totaling 224.7 GB (interestingly, merely counting the number and sizes of XML files takes around 15 minutes). More packages will be translated in the future as soon as they are ported from previous Coq versions. Translating to MMT only the Coq standard library takes about 22 hours on a standard laptop, converting 15.4GB of (uncompressed) XML into 28.9MB of (compressed) OMDoc. This reduction is not only due to a high compressibility of the OMDoc, but also reflects the fact that every declaration in Coq corresponds to multiple XML files with partially redundant information.

6 Conclusion

Evaluation Our translation covers entirely the syntax of the Coq language and it preserves typing and soundness, with the exception of higher order functors, functor declarations and contravariant functor subtyping. The latter three features do not seem to occur in the 49 libraries that have an opam package which is up-to-date with the last Coq relase. As more libraries become available, we will have to verify that our covariant functor translation is still adequate.

Moreover, we are confident that, if and when future work yields a complete formalization of the Coq typing rules in an LF-like logical framework, our translation will be in a format suitable for rechecking the entire library — with the obvious caveat that such a rechecking would face even more serious scalability issues than we had to overcome so far.

An obvious way to verify that the exported information is sound and complete for type-checking would be to implement an importer for Coq itself or, in alternative, for an independent verifier for the logic of Coq, like the one implemented in Dedukti or the one developed in the HELM/MoWGLI projects [APS+03] and later incorporated into Matita 0.5.x series [ARCT11]. In both cases one would need to develop a translation from MMT theories to the modular constructs of the language, which requires more research. For example, no translation of MMT theories and theory morphisms into modules, module types and functors is currently known.

We would also like to stress that independent verification is not the aim of our effort: the main point of exporting the library of Coq to MMT is to allow independent services over them, like queries, discovery of alignments with libraries of other tools or training machine learning advisers that can drive hammers. Most of these services can be implemented even if the typing information is incomplete or even unsound (e.g. if all unvierses are squashed to a single universe, making the logic inconsistent).

Limitations and Future Work Our translation starts with the Coq kernel data structures and is thus inherently limited to the structure seen by the kernel. Therefore, record types and type classes are presented just as inductive types, that is the way they are elaborated before passing them to the kernel. This is unfortunate as recent Coq developments, most importantly the Mathematical Components project [GGMR09], have made heavy use of records to represent theory graph–like structuring and an unelaborated representation would be more informative to the user and to reasoning tools.

In fact, even sections are not visible to the kernel, and we were able to include them because we were able to reconstruct the section structure during our translation. We expect that similar efforts may allow for including record types and canonical structures in the theory graph in the future and we plan to start working on that next.

Many libraries avoid module and functors and achieve modularity using other more recent features of Coq that are invisible at the kernel level, like type classes. Moreover type classes, canonical structures, coercions, etc. are necessary information to extend a library because they explain how the various mathematical notions are meant to be used. While the already cited services that we plan to provide do not depend on them, importing the library in another system to build on top of it surely does. Therefore a future challenge will be to find system independent generalizations and representations of such constructs, which will be necessary to incorporate them into a logic and system independent tool like MMT.

Our formal representation of Coq declarations includes the types of all constants and variables, but we use a single type in the logical framework for all Coq expressions. As we explain in Sect. 3, we consider a typed representation of expressions infeasible at this point. Our representation does not include the typing rules for the expression, but this is not due to a principal limitation: it is possible to add these rules to let Mmt type-check the library. But formalizing the rules of the Coq type system is in itself a major challenge, and representing the details of, e.g., Coq's treatment of pattern matching or sort polymorphism may even require innovations in logical framework design.

References

APS$^+$03. Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical Knowledge Management in HELM. *Ann. Math. Artif. Intell.*, 38(1-3):27–46, 2003.

ARCT11. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In *Automated Deduction - CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011.

Ass15. A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École Polytechnique, 2015.

BB12. M. Boespflug and G. Burel. CoqInE: Translating the Calculus of Inductive Constructions into the lambda Pi-calculus Modulo. In D. Pichardie and T. Weber, editors, *Proof Exchange for Theorem Proving*, 2012.

BCH12. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

CHK$^+$12. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. A Proof Theoretic Interpretation of Model Theoretic Hiding. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 118–138. Springer, 2012.

CK18. L. Czajka and C. Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018.

CMR16. M. Codescu, T. Mossakowski, and F. Rabe. Selecting Colimits for Parameterisation and Networks of Specifications. In M. Roggenbach and P. James, editors, *Workshop on Algebraic Development Techniques*, 2016.

Coq15. Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

GGMR09. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

IKRU13. M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

KMOR17. M. Kohlhase, D. Müller, S. Owre, and F. Rabe. Making PVS Accessible to Generic Services by Interpretation in a Universal Format. In M. Ayala-Rincon and C. Munoz, editors, *Interactive Theorem Proving*, pages 319–335. Springer, 2017.

KR14. C. Kaliszyk and F. Rabe. Towards Knowledge Management for HOL Light. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 357–372. Springer, 2014.

KR16. M. Kohlhase and F. Rabe. QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge. *Journal of Formalized Reasoning*, 9(1):201–234, 2016.

KW10. C. Keller and B. Werner. Importing HOL Light into Coq. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, pages 307–322. Springer, 2010.

MAH06. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program*, 67(1–2):114–145, 2006.

oFD18. European Commission Expert Group on FAIR Data. Turning fair into reality, 2018. URL: `https://doi.org/10.2777/1524`.

OS06. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In N. Shankar and U. Furbach, editors, *Automated Reasoning*, volume 4130. Springer, 2006.

Rab15. F. Rabe. Lax Theory Morphisms. *ACM Transactions on Computational Logic*, 17(1), 2015.

Rab17. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

Rab18. F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 19(4):1–43, 2018.

RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

Sac19. Claudio Sacerdoti Coen. A plugin to export Coq libraries to XML. In *12th International Conference on Intelligent Computer Mathematics (CICM 19)*, Lecture Notes in Artificial Intelligence, 2019.

A Details on the Coq Logic

Base Logic Typical **base logic declarations** are of the form $\underline{e} : E = E'$ introducing a typed, defined constant. If the type is omitted in concrete syntax, it is inferred, i.e., it is always present in abstract syntax. If the definiens is omitted, we call $\underline{e}$ *abstract*; this is needed to declare axioms in files or inside modules (that cannot be instantiated) or fields in module types (that are subject to instantiation). Type and definiens are *base logic expressions*, which are formed from universes U, qualified references e to such identifiers, the usual constructors of dependently typed λ-calculus with local definitions (**let** ... **in**) and a few more constructs that deal with primitive (co)inductive definitions, pattern matching, and total (co)recursive functions.

Coq does not use a fixed, totally ordered **universe hierarchy** and instead maintains a dynamically growing partial order of universes with strict ($U < U'$) and lax ($U \leq U'$) constraints between them. Checking the consistency of the whole set of constraints amounts to verify ayclicity of a certain graph of universes.

Universes are either declared explicitly or are generated automatically: every occurrence of `Type` in the input syntax is elaborated as `Type@`$\{u\}$ for a fresh universe constant u. Universes are formed from qualified universe constants u and universe variables y using the maximum and successor functions. `Type@`$\{u\}$ together with `Prop` and `Set` make up the **sorts** of Coq.

It is far beyond the scope of this paper to describe the details of Coq's constructors that deal with **inductive types**. Moreover, their typing rules have been subject to many extensions and refinements in the last few years (e.g. recent additions have been non uniform parameters, sort polymorphism, universe polymorphism). We only mention that the grammar presents the syntax visible in the kernel and supported by our translation. For example, a pattern-match takes as arguments the id of an inductive type, the term to match, a term that is used to compute (according to complex rules) the type expected from each case and for the whole match, and for each case the term that buids the result.

Modules and Module Types Module types and modules roughly correspond to ML signatures and structures or to Java abstract and concrete classes. Their syntax is almost identical, but they have different semantic roles. Intuitively, only declarations in **modules** exist in the sense that they can be used and executed, e.g., declaring an inconsistent axiom in a module type is inconsequential.

Module types serve to either permanently hide definitions, declarations or definiens from a modules, or to assert their existence in a module declaration. More precisely, the semantics of every module (type) is a sequence of declarations, and 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,
- if any such name has a definiens in T, it has the same definiens (up to computation) in M,
- M declares every module name that is declared in T and does so with a module type that conforms to the module type in T,

– M declares every module type name that is declared in T and does so with a definition that conforms to the one in T.

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

Maybe surprisingly, even though universes and constraints can be declared in module (type) bodies, they are not part of that module (type): a single universe hierarchy is maintained globally. Declaring a universe or constraint in a module (type) is semantically equivalent to declaring it just before the module (type), except that the qualified identifiers (that carry no semantics but their identity) are different. Consequently, universes are ignored by conforms-relation.

Modules M and module types T are introduced by a number of different declarations, which can be condensed into the four productions given in our grammar. Both modules and module types may be defined in two different ways: as an abbreviation for a module (type) expression (the definiens), or as a new module (type) given by a list of declarations (the body). These two cases are called *algebraic* resp. *interactive* module (type) declarations in Coq. For modules (but not module types), a third option exists: a module may be abstract, i.e., have neither body nor definiens.

Every module (type) expression can be elaborated into a list of declarations, and this is always done by the Coq kernel. We call that that *flattening* of the module (type) expression. Thus, algebraic module (type) declarations can be seen as abbreviations of interactive ones.

Every module (type) declaration may use the <: operator to give a list of module types that it conforms to. In that case, Coq checks that conformation condition after checking the module (type) declaration. Instead of <: attributions, a module declaration $\underline{m}$ may use the : operator to attribute a module type. (Our grammar allows both <: and : for simplicity.) A type attribution $: T$ is stronger than a conformation attribution $<: T$: it additionally restricts the interface of $\underline{m}$ to be exactly that of T (up to qualified identifiers) — any additional names or definiens in the body/definiens of $\underline{m}$ can never be referenced (and are in fact thrown away by Coq after checking them).

Functors Both modules and module types can be functors, i.e., they take a list of module bindings $\underline{m} : T$, which may be used in the definiens/body. To form module (type) expressions, a module (type) functor can be applied to a list of appropriately module-typed modules, the semantics of which is given by β-reduction in the usual way, followed by replacements of some occurrences of constants with their definiens according to additional "!" flags and complex rules. Partial applications to fewer arguments than bindings is possible.

Some subtleties are notable about functors:

– A module (type) functors may only take module bindings, i.e., no base logic bindings $x : E$.
– A module (type) functor can be applied only to module *identifiers* and not to module *expressions*. Thus, functor application cannot be nested, which is awkward from a concrete syntax perspective but significantly simplifies the implementation (only names can be substituted for names).

- When applying a functor, β-reduction is usually avoided: instead of substitution every occurrence of a bound module name $\underline{m} : T$ with a module n, one can add a module definition $\underline{m} : T := n$. This is semantically equivalent but may be more practical or efficient. The "!" qualifier is used when avoiding β-reduction does not yield the intended result.
- We will discuss higher-order functors in Sect. 4.3.

The grammar provides one more production for module type expressions only: the `with` operator injects a definiens for a previously undefined base logic name or module name in the flattening of a module type.

Higher-Order Functors A curiosity of Coq is that the abstraction $(x : S)$ in a module type functor **Module Type** $t(x : S) := T$ can be understood in two very different ways.

- *λ-abstraction:* $t := \lambda x : S.T$ The module type t is parameterized over x, i.e. t is a function that returns a module type. The parameterized module type t can be used, for example, to type explicitly defined functors in the following way: **Module** $m(x : S) : (t\ x) := M$. In particular, for each module n, $m(n) : t(n)$. However, it cannot be used to type a functor declaration.
- *Π-abstraction:* $t := \Pi x : S.T$ Here t is the type of functors of domain S and codomain T. It is useful to type functor declarations and arguments of higher order functors in the following ways: **Module** $m : t :=$ of **Module** $m(n : t) : \ldots := \ldots$.

The Π-reading leads to a much more expressive language than the λ-reading causing deep difficulties such as a lack of conservativity, the breakdown of our covariant translation, and the problem of applicative vs. generative functors.

Surprisingly again, Coq supports *both* uses at once without distinguishing them syntactically, i.e., t can be used both as a module type–valued function and as a functor type. In other words, while the basic logic of Coq is a Π-typed λ-calculus, the module language implements a λ-typed λ-calculus, like the ones used at the beginning of interactive theorem proving (e.g. in the Automath project) and then abandoned for lack of good meta-theoretical properties. Even more surprisingly, while functor types are heavily used in the entire Coq library, they seem in practice to never be used to type functor declarations. Therefore once again our covariant translation that forces the λ-reading is sufficient for now.

Sections Coq files and modules can be divided into nested sections (but not vice versa: sections may not contain module (type) declarations). A section is similar to a module functor except that the arguments of the functor are (i) base logic or universe/constraint declarations instead of module declarations, and (ii) interspersed throughout the section instead of listed at the beginning of the module.

Concretely, inside a section additional declarations are allowed, namely the base logic declarations with the **Variable** qualifier for constants or the **Polymorphic** qualifier for universe declarations and constraints. (Arguably, the latter is a misnomer, and all three should use the qualifier **Variable**.) Inside the section, these behave like their base logic counterparts. Outside the section, they disappear: all other declarations of the section are $\lambda\Pi$-abstracted over all **Variable** declarations that they depend on and over all **Polymorphic** declarations of the section. (Constraints are not abstracted explicitly but are treated as proof obligations that are attached to constants and inductive type declarations, to be triggered when the constant/type is applied to concrete universes.)

Qualified Identifiers Declarations of constants $\underline{e}$, universes $\underline{u}$, modules $\underline{m}$, or module types $\underline{t}$ can occur inside source files, module types, modules, and sections. When forming base logic expressions E, universe U, module expressions M, and module type expressions T, these declarations can be referred to using qualified identifiers e, u, m, resp. t. The qualified identifier is the list containing the following:

1. The root identifier that is defined by the Coq package. 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 sections are ignored when forming qualified identifiers: all declarations in a section introduce names in the containing module (type) or file.

B Details on the Encoding of the Coq Logic

For an object logic with universes like Coq, there are generally three ways to encode object logic expressions t of type A of universe U:

- weak: as terms $\overline{t} : \texttt{expr}$ for some $\texttt{expr} : \texttt{type}$ and with $\Pi : \texttt{expr} \to (\texttt{expr} \to \texttt{expr}) \to \texttt{expr}$
- semi-strong: as terms $\overline{t} : \texttt{expr}\,\overline{U}$ for some $\texttt{univ} : \texttt{type}$ and $\texttt{expr} : U \to \texttt{type}$, and with $\Pi : \Pi u, v.(\texttt{expr}\,u \to \texttt{expr}\,v) \to \texttt{expr}\;\max\,u\,v$
- strong: as terms $\overline{t} : \texttt{expr}\,\overline{U}\,\overline{A}$ for some $\texttt{univ} : \texttt{type}$, $\texttt{tp} : \texttt{univ} \to \texttt{type}$, and $\texttt{expr} : \Pi u.\texttt{tp}\,u$ and with

$$\Pi : \Pi u, v.\Pi A : \texttt{tp}\,u.(\texttt{expr}\,u\,A \to \texttt{tp}\,v) \to \texttt{tp}\,(\max\,u\,v)$$

The stronger encodings have the advantage that $\overline{t}$ always carries its type and/or universe so that some aspects of the type systems are already enforced by the representation of the syntax. In particular, the strong encoding allows only well-typed terms so that the entire type system rules are already captured in the syntactic rules. The weaker encodings require the formalization of additional typing judgments and rules.

Stronger encoding can be very elegant, but if the type systems becomes more complex (as is certainly the case for Coq), they can become awkward or infeasible. Firstly, if the universe hierarchy is cumulative (as for Coq), terms that carry their universe must be explicitly cast whenever they are used at higher universes.

These casts must be inferred and inserted when translating from Coq to LF. The casts then interfere with equality checking and therefore additional commutation rules to push the casts around have to be inserted, further complicating the encoding (the so-called coherence hell problem). Secondly, the representation of the syntax takes additional arguments. For example, with the strong encoding, the encoding of function application takes 4 additional arguments (2 universes and 2 types), which have to be inferred. This has little relevance for human-written content as the additional arguments are implicit and can be reconstructed by the framework. But it has huge implications for exporting large libraries: inferring and storing the additional arguments would cause an infeasible (= non polynomial) increase in size and time, unless sharing is employed. Sharing, however, is very complex in the case of languages with binders and hard to capture with linear syntax.

Therefore, we (have to) use the weak encoding. This also has the advantage that we can omit the formalization of Coq's typing rules for inductive types, the module system and totality checking, which are very long to implement and error prone.

Coqine [BB12,Ass15] uses a formalization of the calculus of constructions in Dedukti. The latest version[4] contains multiple variants of the strong encoding. Contrary to Coq's partial order, the encodings use a fixed, total order of universes.

C Details on the Translation of Sections

The section mechanism of Coq pre-dates the module system, but was not abandoned when modules were introduced. Like functors, it allows to describe a theory parameterized over a bunch of assumptions that can later be instantiated. Differently from functors, the assumptions do not need to be collected in a module to perform the instantiation (which is a benefit), but on the other hand the instantiation must be performed for every occurrence instead of once.

In practice, in most cases users still seem to prefer sections over modules. Moreover, functors are only implemented by Coq (and the languages of the ML families), whereas other provers, like Isabelle and PVS, implement constructs that behave like sections (e.g. the locales of Isabelle).

In Coq sections are a feature of the input language, but they are elaborated away and thus not visible in the kernel language: all definitions in a section occur at the kernel level outside the section and abstracted over all the variable declarations, polymorphic[5] universe declaration and and polymorphic constraints declared in the section. However, it seems important to recover them during the translation to avoid presenting to the user an input that is significantly different from what he wrote.

[4] See `https://github.com/Deducteam/CoqInE`.

[5] Polymorphic is really a misnomer for "variable universes" (and constraints over them). Moreover, for no real reason, Coq abstracts over all polymorphic unverses and constraints, even if unused, but only over the variables that are used.

We were able to recover the sections: for each section s we generate an MMT theory $\overline{s}$. **Variable**, **Polymorphic Universe** and **Polymorphic Constraint** declarations are translated as usual and put inside s with additional metadata to mark their "abstracted" status. More metadata are attached to the constants that were declared inside the section to identify the abstractions that were over section variables/universes in the input language.

One might think that we can additionally generate a morphism $\overline{s} \to \overline{p}$ that witnesses how the section relates to $\overline{p}$. This is not true — the relation is non-compositional and a formal statement would require the much more complex *lax* theory morphisms of [Rab15].

However, it is worth noting that in our implementation, we were able to capture the section semantics by using a variant of the MMT feature we developed in [KMOR17] to represent PVS's includes of uninstantiated parametric theories. A section is represented as a nested theory, that generates constants (where the section variables are abstracted away) in the parent theory. Notably, this implementation is independent of the Coq meta theory and can immediately be used in other settings as well.