INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME

Project IST-2001-33562 MoWGLI

Report n. D2.a
Exportation Module

Main Author:
C. Sacerdoti Coen

Project Acronym: MoWGLI
Project full title: Mathematics On the Web: Get it by Logic and Interfaces
Proposal/Contract no.: IST-2001-33562 MoWGLI

Contents

1 Overview

This document describes the first public version of the MoWGLI Exportation Module for the Coq system. The Exportation Module is a Coq standard contribution which adds new commands to export to XML theorems and definitions. The Makefile of Coq and the utility `coq_makefile` are also modified to easily allow to export in a batch process the whole standard library of Coq and all the future new theories. The XML output is valid with respect to tentative DTDs we provide; they are also included as appendix to this document.

The MoWGLI Exportation Module subsumes the functionalities of the previous HELM Exportation Module, now included in the official distribution of Coq, version 7.3. The Coq team decided to include it in the next releases of Coq.

The next section describes the module from the user point of view. Then we present the objectives and the main guidelines we followed. The third section is an exact description of the information exported. Finally, in section 5 we sketch the design of the Exportation Module.

2 MoWGLI Exportation Module Usage

The MoWGLI Exportation Module is a standard contribution of the Coq system. Thus the new commands it provides are already available to the Coq toplevel, without having to `Require` anything. They are

```
Show XML File "filename" Proof.
Print XML File "filename" qualified_name.
```

and their variants

```
Show XML Proof.
Print XML qualified_name.
```

`Show XML Proof` and `Print XML` can be used anywhere `Show Proof` and `Print` are allowed. In particular `Show XML Proof` is used to export to XML an incomplete proof and all its related information. `Print XML` is used to export any constant or inductive definition block.

Note: *exporting a theorem using* `Show XML Proof` *just before closing it using* `Qed` *or* `Save` *provides more information than exporting it using* `Print XML` *once it is closed. Thus, the first way is recommended.*

WARNING: `Print XML` *exports objects in their current, possibly partially discharged, form. In order for the exported XML library to be coherent,* `Print XML` *must be applied only to undischarged objects; in other words objects must be exported before closing the section they are defined in.*

If a filename is not provided, all the generated XML files are output one after another on the console. Otherwise the provided `filename` is used as a basename to generate (using different extensions) the names of all the files, which are all put in the same directory.

The above commands are useful to create XML files to inspect during the interactive development of a theory. To export whole theories, a different machinery is provided: the commands `coqc` and `coqtop` now have a new flag "`-xml`". When the flag is set, theorems and definitions are automatically exported to XML as soon as they are defined. The output goes to the console if the `COQ_XML_LIBRARY_ROOT` environment variable is unset. Otherwise a whole hierarchy of files and directories are written to the disk; the hierarchy is rooted in `COQ_XML_LIBRARY_ROOT`. For each exported object several files are generated, all in the same directory. The directory chosen reflects the qualified name of the object.

To automatically activate the "`-xml`" flag for the compilation of both the Coq sources and user-provided theories, the `COQ_XML` environment variable must be set to `-xml`. To achieve this effect, we modified both the Coq `Makefile` and the utility `coq_makefile`. Thus, before exporting an old theory to XML for the first time, `coq_makefile` must be run again.

WARNING: *The new Exportation Module is not backward compatible to the old* HELM *module. All the others previously available commands are removed. The exported information is different and all the DTDs have been changed.*

3 Objectives and Design Guidelines

The main aim of the Exportation Module is to provide Coq users a way to contribute their developments to the distributed library of documents that will be managed in MoWGLI. The information exported must be detailed enough to allow all the planned functionalities. In particular, the output must contain enough logic relevant information to allow independent proof-checking; it must contain enough additional logic redundant information to make it possible to develop stylesheets to map the information to a content encoding first and to a human readable presentation encoding later on; it must form a fine-grained hypertext of mathematical results, that can be assembled together to form new mathematical documents; it must record the commands issued to the system to achieve robustness in spite of changes in the underlying theory[1].

The first design challenge was to select exactly *what* information must be exported and *how it must be organized* both in the DTDs and on the file system. To make our decision, we have followed a set of guidelines that were maturated in our previous experience with the HELM exportation module for Coq. In that prototype, developed in the far 1999, we exported only a much smaller subset of the relevant information we are interested in now. Moreover, trying to develop prototypes of other tools on top of the exported information, we realized that we made several mistakes in the development of the DTD. This and some other exporting experiences[2] greatly helped us in identifying some guidelines. In the rest of the section we present those guidelines. Section 4 is devoted to a short description of the information actually exported.

[1]If some definition a theorem is relying on change, the user must be able to easily redo the theorem, reusing as much as he can of the original proof. For Coq, this amounts to change some of the given commands.

[2]The group of Bologna also wrote an exportation module prototype for the NuPRL library. The group of Saarbrucken exported the libraries of many theorem-provers and some proof-assistant to the OMDoc format.

Guidelines for the Development of Exportation Modules and Related DTDs

Catalogue the information according to its use.
When the information is required for more than one task, factorize.

Before writing the DTDs, catalogue all the information that is available inside the system and all the information you are interested in (which may or may not be already available). Since different tools may require different subsets of the whole knowledge and since we are likely to change the DTDs and the documents often, insulating the tools from changes in other parts of the library is almost mandatory. It is not unusual to find out later that the grain was not fine enough. Example: not every operation that is applied to a theorem requires both the statement and the proof. Finding out which lemma can be applied, for instance, just requires the statement or, even better, the list of its hypotheses and the conclusion.

Sometimes some data is required to perform more than one task. In that case it is better to factorize, even if the factorized information does not seem at once to have any special meaning.

It is often the case that the information required is logical redundant or, more generally, implicit in the system. A typical example is missing linking information, for example between the occurrence of a variable and its definition. Since extracting the implicit information inside the system is easy, but it may be very difficult, time consuming or hardly possible to do with an external tool, our second guideline is to

Make implicit information explicit.

Once the information classes have been identified, it is time to start developing a syntax (a DTD in the case of XML) for them. At this stage it is also important to understand what are the relations between the data belonging to the different classes. Those relations must be made explicit. Thus the third guideline is

Develop one DTD for each class we are interested in.
Make the links between different instances explicit in the DTD.
Locate the atomic components.

Our initial expectation was to be able to heavily link at a very fine grained level the information collected in the documents belonging to the different classes. For example, it is reasonable to link:

- every type of a sub-term of a proof (inner-type) to that sub-term;
- every node of the proof-tree (that corresponds to a user or system issued tactic) to the sub-term it generated.

Still it is often the case that there is no real correspondence between those notions inside the system. For example it may happen that a tactic generates a term that is not present in the final proof, because a subsequent tactic modified or discarded it. Another possibility is that the whole proof can be generated using dirty tactics that create detours or lot of redexes that are later simplified during a clean-up stage. We consider this behaviour of the system to be a debatable design decision, since it introduces a gap between the proof that the user wants to describe and the proof that is produced at the end. This may be especially annoying when the user is really interested in the generated proof-object and he is not given any effective tool to inspect it.

In these cases, instead of simply avoiding to provide that linking information, it is better to pinpoint to the developers of the system the problematic operations to be modified. In the worst case it may be necessary to create a branch in the system development to obtain a slightly modified tool that is used to produce the library that will be eventually exported.

The atomic components that must be located belong to the same information classes and represent the minimal referentiable units that have a precise meaning when considered as stand-alone documents[3]. Sometimes this notion is quite fuzzy. For example, what are the atomic components of a block of mutually defined functions? We may choose that the only atomic component is the whole block, and the functions are simply subparts that may be referenced starting from the whole block (using XPaths, in the case of XML). Or we may consider each function to be atomic and make it refers to the other mutual defined functions via links. Note that some operations (rendering, proof-checking) require the whole block, while others (extraction of metadata used to answer queries) operate on one function at a time. Note also that, usually, the whole block is a tiny object.

The need to clearly identify the atomic components is related to our next guideline:

> Do not mix information about several components or belonging to several classes: one file for each class and for each component.

Assembling together information from different classes is perhaps the worst pitfall, for several reasons:

- The set of operations we want to perform on the data is not a fixed one. For example, we can implement several indexing tools to be able to answer to different queries. Since every tool requires and produces new information that should be put into the library, it is fundamental to be able to define new file formats (DTDs) and change existent ones. If several information classes are assembled together, this means that we will have to modify all the already developed tools to handle (usually to ignore!) the new or modified syntax. Moreover we will have to regenerate huge amounts of files distributed over the network (we are supposed to work with distributed libraries of formal mathematical knowledge).

- The amount of information we want to handle is large. For example the information we export from the Coq library, once saved as compressed XML files, require more than 1Gb. It is not unusual to have single theorems that, exported in XML and compressed, require 24Kb. The biggest proof tree alone is an XML file that, uncompressed, requires about 1Gb. Thus parsing (or even lexically analyze) these files is an expensive operation. Of course we want to avoid as much as possible to waste time in parsing parts of a file we are not interested in.

- It is reasonable, at least for some applications, to use a data-base to hold the exported information. In that case the XML representation is used just as an exchange format and mixing several data in the same file is not a problem, since the information will be disgregated inside the data base. This approach, though, limits a priori the possibility of implementing other tools that, instead of connecting themselves to the database, will just work on the original files. Moreover, the database approach is surely heavier from

[3]Of course the documents may have links to other documents that must be considered when determining their meaning. What is important here is that if some kind of context is request, it must be explicitly stated by means of explicit links.

the point of view of Web-publishing, where the most successful model is that of Web pages publishing, which is simpler and less demanding.

So far we have exposed the guidelines we followed on how to decide which classes of information should be exported and how files should be organized. We will now go back to the problem of designing the DTD and focus on the main difficulties that must be faced.

The main aim is to assure that information exported is complete, general enough and unbiased, in the sense that it must not force some design choices that we would better avoid. The risk is to later find out to be unable to develop the expected tools.

> Describe, as much as possible, the theory and not the implementation.
> Do not be afraid of mapping concepts from the implementation back to the theory.
> Make the theory explicit.
> Introduce new theories to justify the implementation.

There may be a big gap between the theory a system is based on and the way it is implemented. Proof-assistants are implemented starting from a well-known theory (e.g. Martin-Löf Constructive Type Theory for NuPRL; the Calculus of Constructions for Coq). Later on, extensions to the theory are implemented and possibly proved to be correct. For example, both in the case of NuPRL and HOL, logic constructions that are derived in the original theory are made primitive in the implementation for efficiency reasons. In NuPRL this is done, for example, for the integer numbers and the usual arithmetic operations on them.

What usually happens is that after some years it is difficult to reconstruct a unified theory of all the extensions provided, while many other changes are completely undocumented. This happens not only in the kernel of the systems, which is responsible of checking that inference steps are well-applied, but also in external layers that rely on techniques (e.g. higher order unification) that are well-known in the literature and that requires extensions to the representation of the proofs (e.g. metavariables that are typed holes in a proof-term).

When exporting from Coq in our 1999 prototype, the HELM group decided to be quite close to the implementation, even if some parts of what it exported were partially not understood. Later on the HELM team developed a prototype proof-checker first and a proof-assistant later, and realized that it had no clear foundation for the theory it was implementing. Moreover, having represented not the terms of the theory but their encoding in the structures of Coq, the implementation of many operations were forcedly isomorphic to what was done in Coq, even if this prevents experimenting with other solutions. Sections 4.1 and 4.2 give two examples of situations where in MoWGLI we decided to export information according to a new formulation of the underlying theory.

Another severe problem that must be faced when exporting the information is that a proof-assistant may be implemented in an imperative way, where there is a strong notion of time and the objects of the library change from time to time. This behaviour seems utterly incompatible with the notion of mathematical library:

> A library, due to its nature, is a random access structure, both in space and time. Try to minimize the dependencies, give them the right direction, beware of "imperative" commands of the system that change already defined things. Make them immutable.

How can we face the situation in which we have to export some imperative information?

For example, Coq has many other examples of such kind of information:

- A list of theorems that are considered while automatically searching a proof. Theorems can be added and removed from this list at pleasure.
- Implicit variables. Even if the system automatically chooses the variables it thinks it may infer, the user can force at any time the set of implicit variables for an object. Subsequent commands will use the new set.
- Opacity. It can be changed from transparent to opaque, to simulate abstract data types.
- Parsing and pretty-printing rules.

The situation can be described in two ways. The first one is to add the imperative command that changes the system status to the library, adding links from the command to all the other objects that are affected. The second way is to say that every command or definition of the system has an implicit dependency on the state of Coq. This dependency can be effectively make explicit. While the first solutions is easier to implement, because that information is already available in the system, interpreting a command means in practice re-running all the commands in a clone of the Coq system to build the status requested. The second solution, instead, allows to avoid rebuilding the status. Remember that, in a distributed setting, collecting all the given commands and definitions just to update them because of imperative commands is already an unaffordable operation. Managing a distributed status may be even worse. The price to pay, of course, is that objects that change are replicated again and again for each status they may have. Our experience shows that either the requested status information is minimal or there is some problem in the formulation of the theory that should be better removed, as in the case of discharging.

Conclusions

To conclude, even if many of the above suggestions may appear obvious, the temptation of sticking to the internals of the system when exporting the information is great. The experiences with the previous HELM exportation module suggest that the design phase of the DTDs requires a lot of time and a lot of though; but the result is worth the time spent, since wrong decisions will greatly slow-down further developments.

In the next months we will validate our work trying to build tools that work on our exported information. The feedback we will get allows us to better understand the strength and weakness of the exported information. It is possible that we will need to redesign parts of the DTDs or to export other information. Moreover, the Coq system is ever-evolving. The Coq team is already changing again the base theory of the system, introducing, for example, a notion of modules. As a consequence, a constant work is required to keep the exportation module updated and to export the new information. Success is granted by the fact that the module is already developed in the INRIA CVS and maintained by the Coq team, which is a member of MoWGLI.

4 The Information Exported

We have identified the following classes of information in the library of the Coq system:

- Definitions, inductive definitions and proof-objects as sets of lambda-terms, according to the Curry-Howard isomorphism and the theory of CIC. Even for definitions, this is not what the user entered to the system, but the result of complex post-processing, typing and refining rules implemented in Coq. This is what is actually *proof-checked*, by type-checking the lambda-terms and testing the convertibility of their inferred type (in case of a proof, what the theorem really proves) with the expected type (the statement of the theorem). Theorems are further refined (see below).

- Statements of the theorems. They are useful not only for proof-checking. First of all the user can be interested just in the statement of a theorem, that can be parsed and rendered independently of its proof. Secondly they are the only information required to answer several kind of logic-dependent queries:

  1. which lemma can be applied in this situation?
  2. which theorem concludes an instance or a generalization of something?
  3. what can we conclude from a certain set of hypotheses?

  Note that the previous queries are essentially based on the notion of unification, which is an expensive operation when performed on very large sets of candidates (that must also be parsed and loaded into memory to apply unification).

  Statements of the theorems can be further divided into hypotheses and conclusion. This is not a trivial task, since this information is encoded in a lambda-term and because the constructors of the logical framework (Π-abstractions) are overloaded in Coq to mean either a dependent product, a non dependent product (function space), a logical implication or a universal quantification. Other logical frameworks as the one of NuPRL, instead, can have several primitive or derived constructors for introducing hypotheses. Thus, to implement the logic independent queries in a general way, the separation must be performed in advance (either statically or dynamically, when needed).

- Logic independent information extracted from a statement. For example, the single notion of occurrence of a definition in a statement is useful to answer interesting queries:

  – which theorem states some property of an operator?
  – which theorem *may* be applied in a certain situation? Note that if we want to prove some fact about the multiplication of two real numbers, we are interested only on those statements where that multiplication occurs and no other uninteresting definition (let's say the "`append`" function of two lists) occurs. So we can effectively use this information to quickly filter out those theorems whose application will never succeed. Note that this filtering operation is logic independent (then it can be provided once and for all for every system), it is easily implemented using a standard relational or XML-based data-base and can be extremely more efficient than trying one at a time the applications, which usually involve higher-order unification of two expressions.

- Proofs. Their size is usually orders of magnitude bigger than the size of their statements. They are never rendered alone, but are an interesting part of the library for data-mining (for example to recognize similar proofs or to understand the complexity of some proofs). Usual operations involving proofs alone are their improvement, where a group of inference steps, possibly automatically found by the system, may be replaced with a shorter proof, usually human provided.

- Logic independent information extracted from a proof. As for the case of statements, the most interesting notion is that of occurrence. Given the list of occurrences it is easy to answer the following queries:
  - which proofs depend, directly or indirectly, on a given lemma or axiom?
  - which axioms does a proof depend on, either directly or indirectly?
  - which part of the library will be affected if some definition or axiom is changed?
  - what should I learn to be able to understand the following theorem?
- System dependent information related to a proof or definition. For example, in Coq and other systems there exists a notion of implicit arguments, which are those arguments that can be automatically inferred by the system if they are not provided. For example, when writing $x = \pi$ it is clear that the monomorfic equality of type $\forall T.T \rightarrow T \rightarrow \mathbf{Prop}$ is applied to the set of real numbers. This information is not necessary to proof-check a document, but may be useful to render it: information that can be inferred by a system can often be inferred by the human being and is usually omitted in an informal presentation. Finally, note that implicit arguments are really system dependent, since a more powerful system may be able to infer more information and thus consider more arguments to be implicit.
- Redundant information related to a proof or definition. A typical example is the opacity of a constant. Opaque constants are abstract data types, whose exact definition can not be inspected. Proofs in proof-irrelevant systems are always opaque. In those systems that are not proof-irrelevant, such as Coq, all the constants may be considered transparent for the sake of proof-checking. This means that all constants may be expanded during proof-checking. Often, though, it is not necessary to expand every constant for proof-checking and knowing in advance which constants may not be expanded can make the system much more performant. This information is essentially logical redundant, since there is an easy algorithm to make it explicit: try type-checking without any expansion and, in case of failure, backtrack and expand. Of course the computational complexity of this algorithm is, in the worst case, exponential in the number of constants that occur in the theorem being typed.

  Another even more interesting example of redundant information is the types of the sub-expressions of one proof, that corresponds to the conclusions of the subproofs. This information is completely logical redundant (if type-inference is decidable), but it is essential to render the term in a pseudo-natural language.
- Metadata related to definitions and theorems. These are other logic independent information such as author name or the version of the system the proof was developed in that can be useful to implement other kind of queries. Metadata can range from simple to very complex ones, as those needed in educational systems. In Coq, though, they are not provided inside the system or they are just given as unstructured comments. So we are not able to export this information.

- A system dependent history of the operations that lead to the creation of a definition or theorem. In the case of Coq, we can export the proof-tree, which is the internal structured representation of the list of tactics (and their arguments) used to prove one theorem. This information may be useful both for rendering purposes and to replay the proof inside the system, in case we need to modify it.

- Comments. Comments provided by the users are an extremely valuable form of documentation. The problem is that they are discharged during the lexical analysis of the input to the system and so they are unavailable inside the system itself. Thus we can not export them right now.

- Parsing and pretty-printing rules. Coq performs pretty-printing to ASCII notation, while we are interested in more advanced visual rendering. Thus we do not have any use for this information, that will not be exported.

- Tactics and decision procedures. It would be extremely interesting to put the code of tactics and decision procedures into the library. In this way it would become possible to replay the construction of a proof independently from the system that generated it and from its version. Indeed a problem we face is that every time some detail of the implementation of Coq changes, the same script produces a new slightly different proof. This represents a problem from the point of view of proof-engineers, that must continuously update the proofs every time a definition changes.

  The problem is that, to export tactics to the library, we would need first to formalize a language (and the libraries) to implement them. This is definitely outside the scope of MoWGLI.

Before describing the files we actually export for each object of the Coq system, we now describe two problems we faced. The solution to both of them consisted in exporting logic information from Coq after reformulating it in a variant of the theory of Coq. All the residual information is basically exported as it is found in Coq internals.

4.1 Metavariables and Existential Variables

According to the Curry-Howard isomorphism, a correct proof can be seen as a well-typed lambda-term in some lambda-calculus. Thus an incomplete, partially correct proof must be a well-typed lambda-term with holes therein. To extend the notion of well-typing to terms with holes, an hole (called *metavariable* in the literature) can not simply be a missing term, but must be associated to a sequent. So a metavariable has a given type and a typed context.

The two main operations on metavariables are instantiation and restriction. A metavariable can only be instantiated with a term that is closed w.r.t. the metavariable context and that has the expected type in that context. Instantiating a metavariable is an heavy operation, since it requires a linear visit of the whole proof-term. The other operation, restriction, deletes some hypotheses from the metavariable context. It is required, for example, to perform unification: to unify two metavariables the first step is to restrict both of them so that their contexts become equal (possibly up to a decidable convertibility relation).

How can those operations be implemented efficiently? One possibility is to implement restriction using instantiation: every time a metavariable should be restricted, a new metavariable of the right shape is generated and used to instantiate the old one. Since restriction occurs quite often, this implies that an efficient way to perform instantiation must be designed. This is the approach of Coq: restriction is reduced to instantiation and instantiation is not performed explicitly, but delayed using a new environment that maps every instantiated metavariable to the term used to instantiate it.

A completely different approach implements restriction explicitly, making each hypothesis in the sequent optional, so that it is possible to remove one hypothesis keeping trace of the fact that it was removed[4]. Nothing is done to speed up instantiation, that becomes a seldom required operation.

The two possibilities requires different data-structures and representation of metavariables. Moreover, for technical reasons, in Coq there is a further distinction between full-fledged metavariables (called existential variables and use to represent holes in the type of other metavariables) and restricted metavariables that are used to represent just the open goals.

The old HELM Exportation Module did not change the Coq representation. The HELM team, when implementing its own proof-assistant, found out that that choice forced the treatment of metavariables of Coq. Moreover the distinction between the two kinds of metavariables did not allow any progress in the proof[5]. So the HELM developers defined a different internal representation of metavariables, following the second solution above, and they decided to propose their own new DTD.

Thus, which is the right format for describing metavariables in the MoWGLI library? The only reasonable choice is to stick to the theory, where hypotheses in a metavariable context are not optional and there is no environment to delay instantiation. It is a responsibility of the systems to map back and forth between this standard and well-understood format and their internal encoding.

4.2 Sections, Variables and Discharging

In the syntax of Coq it is possible to abstract a group of definitions and theorems with respect to some assumptions. Example (in Coq syntax):

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
 Definition H1 : Prop := A /\ B.
 Theorem T1 : H1 -> A.
  Proof.
  <some proof>
  Qed.
End S.
(* Here the type of H1 and T1 are different. See code fragment below *)
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True False (I,I)).
 Qed.
```

The previous fragment should be equivalent, from a logic point of view, to the following input:

[4]This information is required for managing explicit substitutions that are needed to allow reduction of terms with metavariables.

[5]This is not a problem in Coq since the proof-tree with holes is generated on-demand only for pretty-printing and exporting purposes. Proof-trees are instead used to describe an incomplete proof and the progress on it.

```
Definition H1 : Prop -> Prop -> Prop := [A:Prop ; B:Prop]A/\B.
Theorem T1 : (A:Prop ; B:Prop)(H1 A B) -> A.
 Proof.
 <some slightly different proof>
 Qed.
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True False (I,I)).
 Qed.
```

The operation that transforms the first code fragment in the second one is called *discharging*. Discharging is implemented in an external layer of the Coq system, in such a way that the kernel of the system is given the discharged term (and the theory of the kernel of Coq does not need to be modified).

Since, for rendering purposes, we are more interested in the first fragment, we exported the undischarged form of the theorems and definitions. The problem with that representation is that the theorem `T1` that is used in theorem `T2` is no more equal to the one defined above: its type is different! This implies two kind of problems in developing tools:

1. While rendering `T2`, we would like to make the occurrence of `T1` an hyperlink to its definition. What we get is misleading for the reader: the theorem `T1` is shown to have a type that is not the same of its occurrence in `T2`.

2. To proof-check `T2` we need the type of the discharged form of `T1`. So we are obliged to discharge `T1` and this leads to serious problems: either we save the discharged form, as Coq does, and this goes against our initial choice; or we discharge the theorem on-the-fly when needed, and, being this an expensive procedure, we have to implement complex caching machineries[6].

The solution we are adopting now is simply to redesign the theory, replacing the notion of discharging with that of explicit named substitution. So, while exporting, we completely change the definition of `T2` to the following one:

```
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1[A := True ; B := False] (I,I)).
 Qed.
```

In this way we are no more exporting the exact definition of `T2` inside Coq, but something more well-behaved for our purposes and that can be mapped back, if needed, to the Coq representation. Both of the previous problems are solved with this representation, since `T1` can be rendered as it is just adding the explicit substitution to the top and `T2` can be type-checked without discharging `T1`, by taking care of the explicit substitution in the typing rules of the system.

The Exported Files

So fare we implemented only the exportation of part of the previously identified information. We will export the missing data as soon as DTDs are proposed in other working packages. The files we export for each object class are now described.

[6]This is what HELM implemented, but the nature of the discharging operation interfered with the usual locality reference principle of caches. As a result we got very poor and unexpected performances.

Constants (i.e. definitions, theorems and axioms):

- A compressed XML file *constant_name*`.con.xml.gz` holding the type of the constant (which is the thesis if the constant is a theorem). Its `DOCTYPE` is `ConstantType`, defined in the CIC DTD (see A).
- If the constant is a definition or a finished theorem, we export another compressed XML file *constant_name*`.con.body.xml.gz` holding:
  1. the body of the constant (which is the proof if the constant is a theorem)
  2. the list of section variables the constant depends on (as the attribute `params`)
  3. an explicit reference (URI) to its type file (as the attribute `for`)

  Its `DOCTYPE` is `ConstantBody`, defined in the CIC DTD (see A).
- If the constant is an unfinished proof, we export another compressed XML file *constant_name*`.con.body.xml.gz` holding:
  1. the partial proof of the theorem, which contains metavariables
  2. a list of conjectures, which are essentially the sequents associated to metavariables. The conjecture context is not a named context (as in Coq internals); De Brujin indexes are used instead of named references.
  3. an explicit reference (URI) to its type file (as the attribute `of`)

  Its `DOCTYPE` is `CurrentProof`, defined in the CIC DTD (see A).
- A compressed XML file *constant_name*`.con.types.xml.gz` holding, for each subterm of sort `Prop` of the constant, the inferred type for that subterm and, in case it is different, its expected type. Inferred and expected types are computed using the algorithm proposed by Yann Coscoy in his PhD. thesis. Its `DOCTYPE` is `InnerTypes`, defined in the Inner-Types DTD (see B).
- If the constant is a theorem that was exported using the `Show XML Proof` command, we also export another compressed XML file *constant_name*`.con.proof_tree` holding the proof-tree that generated the proof. Its `DOCTYPE` is `ProofTree`, defined in the Proof-Trees DTD (see C).

Section Variables (i.e. instantiable hypothesis and lemmas):

- A compressed XML file *variable_name*`.var.xml.gz` holding both the type of the constant and its body (if present). Only variables without body may be later instantiated applying explicit named substitutions to the occurrences of constants that depends on them. The `DOCTYPE` of the file is `Variable`, defined in the CIC DTD (see A).
- A compressed XML file *variable_name*`.var.types.xml.gz` holding, for each subterm of sort `Prop` of the variable, the inferred type for that subterm and, in case it is different, its expected type. Inferred and expected types are computed using the algorithm proposed by Yann Coscoy in his PhD. thesis. Its `DOCTYPE` is `InnerTypes`, defined in the Inner-Types DTD (see B).

- If the variable is a theorem that was exported using the `Show XML Proof` command, we also export another compressed XML file *variable_name*`.var.proof_tree` holding the proof-tree that generated the proof. Its `DOCTYPE` is `ProofTree`, defined in the Proof-Trees DTD (see C).

Mutual Inductive Types Block:

- A compressed XML file *first_type_name*`.ind.xml.gz` holding the whole block of mutual inductive types declarations. The declarations of every constructor of each type are also included. The types occurs in the types of the constructors as free De Brujin indexes; the lower free index is the last type. This reflects Coq encoding. The `DOCTYPE` of the file is `InductiveDefinition`, defined in the CIC DTD (see A).
- A compressed XML file *first_type_name*`.ind.types.xml.gz` holding, for each subterm of sort `Prop`, the inferred type for that subterm and, in case it is different, its expected type. Inferred and expected types are computed using the algorithm proposed by Yann Coscoy in his PhD. thesis. Its `DOCTYPE` is `InnerTypes`, defined in the Inner-Types DTD (see B).

The CIC DTD defines roughly three classes of elements:

- Objects elements: there is one root element for each one of the `DOCTYPE`s described above. Every element has all the attributes already described; moreover it has an `id` attribute of type `ID`, which can be used to attach information to it[7].
- Term elements: we have roughly an element for each constructor of CIC, plus the `instantiate` parameter introduced for explicit named substitutions. The content and attributes of the elements should be clear to people acquainted to the theory of CIC.
- Syntactic sugar: we introduced lots of syntactic sugar elements to make the files clear to theory experts without having to look at the DTD. Some attributes have been moved from term elements to syntactic sugar elements when this improved readability.

The Inner-Types DTD is trivial.

Proof-Trees are sequent representations of the proof. The classical sequent-calculus style proof-tree is a tree whose nodes are made of:

- A sequent to prove, made of a goal and a list of hypotheses.
- The rule used to prove the sequent. Every rule is a primitive rule of the logical framework.
- One subproof for each premise of the rule.

In Coq the notion of rule is replace by that of *tactic*. A tactic can be either *primitive* (if it corresponds to a primitive rule of the logical framework) or *defined*. If it is defined, it must be considered as a sort of macro that, once applied, generates a new proof-tree (made

[7]XPath in theory provides a formalism rich enough to identify every element in an XML file without having to use unique identifiers. Nevertheless, our XML files are peculiar, since they are huge and extremely vertical; it is not unusual to have several hundreds depth levels. The length of XPaths to identify a deep node is $O(n)$, where n is its depth. Moreover, every node of the tree can be referenced in our files (for example to add inner types). So the choice of introducing `id`s for every node is much more convenient.

of both primitive and defined tactics). When every defined tactic is expanded, the result is necessarily a tree of primitive tactics. Thus progressive defined tactic explosion naturally allows structured browsing of the proof-tree.

The DTD is conceived to preserve the distinction between primitive tactics (`Prim`) and defined tactics (`Tactic`). For every defined tactic, its expansion in terms of other tactics is also stored. Thus we have the following elements:

Prim Primitive tactics have a list of subtrees that provide the proof for every premise of the tactic. The sequent to prove is omitted.

Tactic Defined tactics have children elements to describe the sequent they prove. The sequent is made of a `Goal` and a list of `Hypothesis`. They also have a subproof that corresponds to their expansion. The `script` attribute stores the exact user-provided text used to invoke the tactic[8].

Both elements also have an *of* attribute that is the identifier of the node of the lambda-term that was generated by the tactic. Interesting applications are made possible by this attribute. For example, it often happens that a tactic produces a huge sub-term in the proof-object. Once the undesired sub-term is identified, it can be replaced with a smaller one simply identifying the tactic that produced it and providing a new proof-tree to replace the tree rooted in that tactic.

Finally, the `ProofTree` element has an `of` attribute that is the URI of the proof-object produced by the proof-tree.

5 Design and Implementation

Since the code of Coq is evolving daily, the help of the Coq developers in maintaining and updating the exportation module is fundamental. As a consequence we decided to develop the Exportation Module using the Coq CVS system. The whole development is performed on a CVS branch[9], so that the Coq team is free to decide if the module will be part of the next standard distributions of the system.

The exportation process comprises three phases:

1. **Information retrieval.** The relevant information must be extracted from Coq data structures. The new toplevel commands and the "`-xml` " flag must be added to start the exportation process.

2. **Transformations and new information synthesis.** The retrieved information must be mapped to a new format that reflects our DTD. For example, some applications must be converted to explicit named substitutions. New information must be synthesised. For example, Coscoy's double type inference algorithm must be recursively applied to compute the inner-types.

3. **XML generation.** The collected information must be output to XML.

[8]This attribute is useful for pretty-printing since Coq's input language allows syntactic variations to invoke a tactic. We would like to keep the exact user-provided text as much as possible

[9]The name of the branch is *mowgli*.

The main issue faced was to insulate the module as much as possible from Coq changes. Coq data structures are likely to change often. On the contrary, our DTD, that reflects the theory only, will be much more stable. Thus an important issue is to define two intermediate information representations. The first one must just be a collection of all the relevant information, using Coq data structures. It is basically the output of the information retrieval phase. The second one is a representation of the same information that is basically isomorphic to the DTD. Note that, at this level, all the links between the different data must have been made explicit.

As a consequence, the code is naturally factorized into three clusters of functions. The first one are those functions that performs the retrieval phase. These functions rely on a big subset of Coq functions defined outside the kernel (and whose interface is thus quite unstable). The second cluster is made of those functions that transform, assemble and link the data together. They constitute the largest part of the new code and they mainly rely on Coq re-typing functions (which are still defined outside the kernel, but whose interfaces essentially stable, unless the theory is extended). Finally, for the XML generation phase, we have developed a minimal programmer-friendly XML library to describe and output XML structures.

The second main issue was how to associate data to subterms. In fact, in order to improve memory usage and performance, terms are shared in Coq as much as possible. Sharing prevents the possibility to associate data to shared instances in different contexts, unless the whole context is remembered. Adding the context information to terms would imply changing the term structure, since any other way of associating context to subterms just gives back the original problem. Changing the term structure prevents the usage of all the functions developed in Coq and thus it is utterly unfeasible. Thus the only possibility left is to destroy sharing during the retrieval phase. Since most of Coq types, includes that of terms, are abstract data types and since Coq abstract constructors ensure sharing, we were forced to modify the Coq kernel to implement an unsharing function.

The implementation of the unsharing function, used every time a term is retrieved from the system data structures, did not solve completely the problem. In particular, we are interested in exporting both the proof-tree and the generated proof-object, keeping links between corresponding nodes of the two structures. So we were unable to retrieve the two data structures independently and map them to XML, since there is no way to retrieve the linking information. As a consequence we had to reimplement the Coq function that maps proof-trees to proof-objects, in order to record the linking information. The new implementation must unshare the subterms as soon as they are recursively produced. The resulting proof-object, and not the one stored in the Coq library, is the object we have to export[10].

Once the previous two design issues were solved, the whole implementation required approximately 2 man months and produced about 2600 lines of code, comments excluded. The time spent reflects the overall effort: understanding the Coq data structures, synthesizing new information and reimplementing some of the Coq internal functions are a complex job, which requires a lot of interaction with the Coq developers. The work to print the information to XML is essentially marginal (and requires just about 700 lines of code).

Overview of the Exportation Module modules

We now briefly sketch the roles of the several implemented modules.

[10]Of course the two objects must be equal, up to sharing and physical equality.

`Acic`

The `Acic` module defines the two intermediate data structures for proof-objects; their name is `obj` and `aobj`. The first one is a concrete data type with a constructor for each one of the `DOCTYPE`s of our DTD. For the terms argument of the constructor we use Coq term data type. Metavariables already have the form required by our DTD, while explicit named substitutions are introduced only in the next structure. Since the type used for terms is the Coq one, unique identifiers are also missing.

The second structure, `aobj`, almost reflects the DTD. The only missing data are the `sort` attributes of the terms.

`Acic2Xml`

This module is responsible of pretty-printing `aobj`s and inner-types to `Xml.xml token Stream.t`, which is a lazy data structure used to represent XML trees. Inner-types must be provided by means of an hash-table to map subterms to `DoubleTypeInference.types` structures. Another hash-table must be provided to map subterms to their sorts.

The only exported functions are `print_term`, `print_object` and `print_inner_types`, whose semantics is reflected in their names.

`Xml`

This module provides a data structure to represent XML forests as streams of XML nodes. It provides constructors to create `CDATA` and `ELEMENT` nodes. `ELEMENT` nodes are given a name, a list of attributes and the forests of their children. It also provides a pretty-printing function to print an XML textual output either on a file or on standard output.

`Cic2acic`

This module is responsible of mapping `obj` structures to `aobj` structures, introducing unique identifiers, computing the inner-sorts (using Coq typing functions), the inner-types (using the function provided by the module `DoubleTypeInference`) and mapping applications to explicit named substitutions where required.

The main exported function is `acic_object_of_cic_object`.

`DoubleTypeInference`

This module provides the function `double_type_of` whose arguments are an unshared Coq term, its optional expected type and the metasenv, environment and context in which it is defined. The function computes the inner-types of every subterm of the given term and returns an hash-table that maps every subterm node to a `types` structure, which holds both the synthesized type and the expected type (if different).

`Proof2aproof`

This module reimplements the `extract_open_pftreestate` function of Coq that maps proof-trees to proof-objects. The two main differences w.r.t. the one of Coq are the bookkeeping of links between the proof-tree nodes and the terms they produce and the production of existential variables for open goals instead of metavariables. The output is a tuple of several arguments:

- the produced proof-object, equal to the one provided by Coq up to sharing and physical equality.
- a new *metasenv*, which includes not only the existential variables of Coq, but also the ones generated to replace Coq metavariables.
- an unshared version of the proof-tree given in input. This is the proof-tree that is the domain of the following two hash-tables.
- an hash-table to map the unshared proof-tree nodes to the corresponding subterms in the proof-object.
- an hash-table to map the unshared proof-tree nodes to the proof-tree nodes of the flattened proof-tree. The flattened proof-tree is an expanded version of the proof-tree where non-primitive tactics are replaced by the proof-tree that proves them using just primitive tactics. The flattened proof-tree is the proof-tree that is actually exported to XML.

`ProofTree2Xml`

This modules provides a function, `print_proof_tree`, whose arguments are basically those provided by the `Proof2aproof`. The function traverses the unshared proof-tree and its flattened version and generates the XML representation for it. Actually, as for the `Acic2Xml` module, the output type is `Xml.xml token Stream.t`.

`Xmlcommand`

This is the module that, together with `Proof2aproof`, is responsible of the retrieval phase. It provides the `print` and `show` commands that are invoked using the `Print XML` and `Show XML Proof` syntax. It also provides another function, `activate_xml_exportation`, which implements the behaviour of the "`-xml`" flag.

Basically it works in this way: when one of the printing commands is issued, it retrieves from Coq internal structures the environment, context, metasenv and proof-object and, if possible, also the proof-tree. Some other information is also computed at this stage. Then, if the proof-tree is available, it uses the `Proof2aproof` module to compute the proof-object to export; otherwise it uses the one provided by Coq, after unsharing it. This ends the retrieval phase. The transformation and synthesis phase consists in using the `Cic2acic` module to retrieve the `aobj` and the inner-types. To complete the last phase, it retrieves all the XML structures by means of `Acic2Xml` and `ProofTree2Xml` (if the proof-tree was available) and outputs them using the `Xml.pp` pretty-printing function.

`Xmlentries`

This module extends the Gallina syntax with the new `Print XML` and `Show XML Proof` commands, implemented in the `Xmlcommand` module.

Since the code and the function types are daily evolving[11], for a more detailed description

[11]until the current Coq development release freezes

of the interfaces the reader is invited to download the Exportation Module code from Coq CVS and look directly to the ocaml interface files.

The whole module is now reasonably bug-free and is successfully used to export the whole standard libraries of Coq and all the available contribs compatible with the current Coq CVS version. The exportation process requires 3h 35m on a 1.8Gh Pentium IV processor with 512Mb of RAM and produces 1.13Gb of XML compressed files.

A The CIC DTD

```
<?xml encoding="ISO-8859-1"?>

<!-- DTD FOR CIC OBJECTS: -->

<!-- CIC term declaration -->

<!ENTITY % term '(LAMBDA|CAST|PROD|REL|SORT|APPLY|VAR|META|IMPLICIT|CONST|
                  LETIN|MUTIND|MUTCONSTRUCT|MUTCASE|FIX|COFIX|instantiate)'>

<!-- CIC sorts -->

<!ENTITY % sort '(Prop|Set|Type)'>

<!-- CIC sequents -->

<!ENTITY % sequent '((Decl|Def|Hidden)*,Goal)'>

<!-- CIC objects: -->

<!ELEMENT ConstantType %term;>
<!ATTLIST ConstantType
          name       CDATA      #REQUIRED
          id         ID         #REQUIRED>

<!ELEMENT ConstantBody %term;>
<!ATTLIST ConstantBody
          for        CDATA      #REQUIRED
          params     CDATA      #REQUIRED
          id         ID         #REQUIRED>

<!ELEMENT CurrentProof (Conjecture*,body)>
<!ATTLIST CurrentProof
          of         CDATA      #REQUIRED
          id         ID         #REQUIRED>

<!ELEMENT InductiveDefinition (InductiveType+)>
<!ATTLIST InductiveDefinition
          noParams NMTOKEN #REQUIRED
          params   CDATA   #REQUIRED
          id       ID      #REQUIRED>

<!ELEMENT Variable (body?,type)>
<!ATTLIST Variable
          name CDATA #REQUIRED
          id   ID    #REQUIRED>
```

```
<!ELEMENT Sequent %sequent;>
<!ATTLIST Sequent
          no  NMTOKEN #REQUIRED
          id  ID      #REQUIRED>

<!-- Elements used in CIC objects, which are not terms: -->

<!ELEMENT InductiveType (arity,Constructor*)>
<!ATTLIST InductiveType
          name      CDATA        #REQUIRED
          inductive (true|false) #REQUIRED>

<!ELEMENT Conjecture %sequent;>
<!ATTLIST Conjecture
          no NMTOKEN #REQUIRED
          id ID      #REQUIRED>

<!ELEMENT Constructor %term;>
<!ATTLIST Constructor
          name CDATA #REQUIRED>

<!ELEMENT Decl %term;>
<!ATTLIST Decl
          name CDATA #IMPLIED
          id   ID    #REQUIRED>

<!ELEMENT Def %term;>
<!ATTLIST Def
          name CDATA #IMPLIED
          id   ID    #REQUIRED>

<!ELEMENT Hidden EMPTY>
<!ATTLIST Hidden
          id ID #REQUIRED>

<!ELEMENT Goal %term;>

<!-- CIC terms: -->

<!ELEMENT LAMBDA (decl*,target)>
<!ATTLIST LAMBDA
          sort %sort; #REQUIRED>

<!ELEMENT LETIN (def*,target)>
<!ATTLIST LETIN
          id   ID     #REQUIRED
          sort %sort; #REQUIRED>

<!ELEMENT PROD (decl*,target)>
<!ATTLIST PROD
          type %sort; #REQUIRED>

<!ELEMENT CAST (term,type)>
<!ATTLIST CAST
          id   ID     #REQUIRED
          sort %sort; #REQUIRED>

<!ELEMENT REL EMPTY>
<!ATTLIST REL
          value  NMTOKEN #REQUIRED
          binder CDATA   #REQUIRED
          id     ID      #REQUIRED
          idref  IDREF   #REQUIRED
          sort   %sort;  #REQUIRED>

<!ELEMENT SORT EMPTY>
<!ATTLIST SORT
          value CDATA #REQUIRED
          id    ID    #REQUIRED>

<!ELEMENT APPLY (%term;)+>
<!ATTLIST APPLY
          id   ID     #REQUIRED
          sort %sort; #REQUIRED>

<!ELEMENT VAR EMPTY>
<!ATTLIST VAR
          relUri CDATA  #REQUIRED
          id     ID     #REQUIRED
          sort   %sort; #REQUIRED>

<!-- The substitutions are ordered by increasing DeBrujin  -->
<!-- index. An empty substitution means that that index is -->
<!-- not accessible.                                       -->
<!ELEMENT META (substitution*)>
<!ATTLIST META
          no              NMTOKEN #REQUIRED
          id              ID      #REQUIRED
          sort            %sort;  #REQUIRED>

<!ELEMENT IMPLICIT EMPTY>
<!ATTLIST IMPLICIT
          id ID #REQUIRED>
```

```
<!ELEMENT CONST EMPTY>
<!ATTLIST CONST
          uri  CDATA  #REQUIRED
          id   ID     #REQUIRED
          sort %sort; #REQUIRED>

<!ELEMENT MUTIND EMPTY>
<!ATTLIST MUTIND
          uri    CDATA   #REQUIRED
          noType NMTOKEN #REQUIRED
          id     ID      #REQUIRED>

<!ELEMENT MUTCONSTRUCT EMPTY>
<!ATTLIST MUTCONSTRUCT
          uri      CDATA   #REQUIRED
          noType   NMTOKEN #REQUIRED
          noConstr NMTOKEN #REQUIRED
          id       ID      #REQUIRED
          sort     %sort;  #REQUIRED>

<!ELEMENT MUTCASE (patternsType,inductiveTerm,pattern*)>
<!ATTLIST MUTCASE
          uriType CDATA   #REQUIRED
          noType  NMTOKEN #REQUIRED
          id      ID      #REQUIRED
          sort    %sort;  #REQUIRED>

<!ELEMENT FIX (FixFunction+)>
<!ATTLIST FIX
          noFun NMTOKEN #REQUIRED
          id    ID      #REQUIRED
          sort  %sort;  #REQUIRED>

<!ELEMENT COFIX (CofixFunction+)>
<!ATTLIST COFIX
          noFun NMTOKEN #REQUIRED
          id    ID      #REQUIRED
          sort  %sort;  #REQUIRED>

<!-- Elements used in CIC terms: -->

<!ELEMENT FixFunction (type,body)>
<!ATTLIST FixFunction
          name     CDATA   #REQUIRED
          recIndex NMTOKEN #REQUIRED>
```

```
<!ELEMENT CofixFunction (type,body)>
<!ATTLIST CofixFunction
          name     CDATA   #REQUIRED>

<!ELEMENT substitution ((%term;)?)>

<!-- Explicit named substitutions: -->

<!ELEMENT instantiate ((CONST|MUTIND|MUTCONSTRUCT),arg+)>
<!ATTLIST instantiate
          id ID #IMPLIED>

<!-- Sintactic sugar for CIC terms and for CIC objects: -->

<!ELEMENT arg %term;>
<!ATTLIST arg
          relUri CDATA #REQUIRED>

<!ELEMENT decl %term;>
<!ATTLIST decl
          id     ID     #REQUIRED
          type   %sort; #REQUIRED
          binder CDATA  #IMPLIED>

<!ELEMENT def %term;>
<!ATTLIST def
          id     ID     #REQUIRED
          sort   %sort; #REQUIRED
          binder CDATA  #IMPLIED>

<!ELEMENT target %term;>

<!ELEMENT term %term;>

<!ELEMENT type  %term;>

<!ELEMENT arity %term;>

<!ELEMENT patternsType  %term;>

<!ELEMENT inductiveTerm  %term;>

<!ELEMENT pattern  %term;>

<!ELEMENT body  %term;>
```

B The Inner-Types DTD

```
<?xml encoding="ISO-8859-1"?>

<!-- DTD FOR INNER TYPES: -->

<!ENTITY % cicdtd SYSTEM "cic.dtd">

%cicdtd;

<!ELEMENT InnerTypes (TYPE*)>
<!ATTLIST InnerTypes
          of   CDATA      #REQUIRED>

<!ELEMENT TYPE (synthesized,expected?)>
<!ATTLIST TYPE
          of   NMTOKEN #REQUIRED>

<!ELEMENT synthesized %term;>

<!ELEMENT expected %term;>
```

C The Proof-Trees DTD

```
<?xml encoding="ISO-8859-1"?>

<!-- DTD FOR INNER TYPES: -->

<!ENTITY % cicdtd SYSTEM "cic.dtd">

%cicdtd;

<!ENTITY % tactic '(Prim|Tactic)'>

<!ELEMENT ProofTree %tactic;>
<!ATTLIST ProofTree
          of  CDATA #REQUIRED>

<!ELEMENT Prim (%tactic;)*>
<!ATTLIST Prim
          name CDATA #REQUIRED
          of   NMTOKEN #REQUIRED>

<!ELEMENT Tactic (Goal,Hypothesis*,%tactic;)>
<!ATTLIST Tactic
          name   CDATA   #REQUIRED
          script CDATA   #REQUIRED
          of     NMTOKEN #REQUIRED>

<!ELEMENT Goal %term;>

<!ELEMENT Hypothesis %term;>
<!ATTLIST Hypothesis
          name CDATA #REQUIRED
          id   ID    #REQUIRED>
```