From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls

Claudio Sacerdoti Coen

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
sacerdot@cs.unibo.it

Abstract. When we try to extract to an open format formal mathematical knowledge from libraries of already existing proof-assistants, we must face several problems and make important design decisions. This paper is based on our experiences on the exportation to XML of the theories developed in Coq and NuPRL: we try to collect a set of (hopefully useful) suggestions to pave the way to other teams willing to attempt the same operation.

1 Introduction

The formalization of interesting and useful fragments of mathematics and computer science requires the development of a lot of elementary theories. In particular, we recently got more and more evidence of the fact that complex chapters of mathematics may be required even to justify the correctness of simple algorithms [8, 1].

Current tools for the development of formal mathematical knowledge, such as theorem-provers and proof-assistants, have been successful in the last decade only in encoding small parts of mathematics and computer science, starting from the elementary ones. The lack of sharing between the different systems resulted in many theories formalized again and again, each time using a different tool. Moreover, since all of these systems are application-centric and use proprietary formats, the formalized knowledge is not even accessible, if not by means of the system itself. Finally, the tasks of visual rendering mathematical libraries and making them searchable and reusable are complex issues that are usually under-developed in these tools.

Several groups and projects[1] tried to improve the situation, providing two general and largely orthogonal solutions.

The first solution consists of *making the systems communicate* to each other, developing some kind of lingua franca (e.g. OpenMath [12]) and standardizing software interfaces (e.g. the MBase mathematical software bus [9]). This is the solution that less impacts the systems, since it is enough to provide a module

[1] The Calculemus Network, the MKM Network, the OpenMath Society, the HELM Project, the MoWGLI Project, the FDL Project, the MathWeb Project and many others.

(called Content Dictionary in OpenMath) to query other systems and answer to queries, mapping the lingua franca to their logical dialect. This solution works well for computer algebra systems and first order theorem provers, which are all based on a very similar logic and which are quite flexible on the notion of correct mathematical results: since the logic is more or less the same, a result proved by another system can be reasonably considered correct. Fewer applications have been proposed for making proof-assistants cooperating. The reason is that traditionally these systems do not attempt proof-search for undecidable theories and thus are allowed to exploit more complex logical frameworks (Higher Order Logics, Pure Type Systems, Calculus of Constructions, Martin-Löf Constructive Type Theory) with more emphasis on the foundational issues and on the notion of correctness[2]. Since every proof-assistant adopts a different logic, proofs from other systems can not be reused nor trusted, unless encodings of one logic into the others are provided[3]. Finally, note that this solution is successful in avoiding proving the same things again or implementing the same decision procedures, but does not address other problems, in particular those related to the publishing and rendering of the developed theories.

The second solution, pursued by other projects [2, 4], is to develop standard formats for long-term storing of the formal mathematical knowledge and to use them to build distributed libraries of documents. In a previous paper [3] we explained why XML-based formats are a promising choice. Once the standard formats have been defined, tools can be independently provided to manage, render, index, transform, create, check and data mine the documents in the library. Since we must face the multilingual environment of the many logical frameworks, we can not have just one format, but we need at least one for each logic. Nevertheless, those operations which are independent of the specific logic can still be implemented once and applied to every document, either directly or after a transformation that maps the logic level format to a Content Level one (such as MathML Content or OpenMath); the transformation prunes out the encoding details of the rigorous mathematical notion into the logical framework, obtaining an informal representation that is more suitable for searching, rendering and exchange (by means of Phrasebooks) with other systems that are based on a different logic. Instead, logic dependent tools such as proof-checkers, program extractors and decision procedures can work directly on the format that corresponds to their logic, without any need of importing or exporting the documents.

[2] Axioms are (almost) never introduced in these systems, so that the consistency of the theory developed is reduced to the consistency of the logical system. As a consequence, everything must be proved, here comprising very primitive notions such as the fact that 0 is different from 1.

[3] Note that a notion P in the encoding of a system A in a system B is generally different from the same notion P given directly in system B. For example, 0 is defined in set theory as the empty set and in type theory as a constructor of an inductive type. Given that set theory can be encoded into type theory, we can define a new 0 as the (encoded) empty set; still we will not be able to use it in place of the original 0, since its type (and properties) are different

There is an alternative to defining a new XML format for each logic: OpenMath, being an extensible format, can be used to encode any logical system, simply defining in OpenMath a symbol for each constructor of the logic. In this way, though, we do not get any real benefit, since there is no sharing of constructors with other logics; so logic dependent tools will have Phrasebooks defined on disjoint subsets of OpenMath and will still not be able to share information, unless a transformation to a Content Level is provided. The only tools that could be shared are parsers and tools to manage the distribution of the documents, but with no real benefit. Parsing is not made easier with respect to standard XML parsing, since the logic level tools will have to interpret the parsed structures to recognize the constructs of the logic and reject other constructs; ad-hoc Document Type Definitions (DTDs), instead, at least relieve the system from this rejection phase. Distribution tools do not get any benefit either, since they do not need to interpret the data[4]. On the contrary, at the Content Level OpenMath provides already developed libraries for the management and exchange of the information, and may enhance comunication by means of modularly designed Phrasebooks.

While the first solution does not require big modifications in the existent tools, the second approach is much more invasive (but also much more promising), because it implies for the existent tools the change from their application-centric architecture to the document-centric one. Since this is a slow process, the suggested shortcut is to provide modules for the existent systems to extract the mathematical knowledge from their libraries and save it in the new standard format. This way we achieve two different and equally important goals: we can start developing the new tools around an already existent library of interesting documents and we preserve the already formalized content, avoiding redoing the formalization and, in the long term, improving the way it can be accessed and exploited.

In 1999 we wrote for project HELM (Hypertextual Electronic Library of Mathematics) a DTD for a variant of the Calculus of (Co)Inductive Constructions (CIC), which is the logical system of the Coq[5] proof-assistant, and an exportation module from the Coq compiled library files (".vo" files) to our XML format. Several tools were then developed to work on the exported library: a distribution and publishing system; stylesheets to render proofs and definitions in MathML Presentation or HTML; dependency analyzers and renderers for studying the relations among items of the library; a query language based on automatically generated metadata to summarize relevant information extracted from the statements of the theorems; a brand new stand-alone incremental proof-checker; a prototype of a proof-engine. Of the previous tools, all but the last two are logic independent (which implies DTD-independent). All the other tools can be reused on any other DTD used to encode any other logical framework. The

[4] They may inspect metadata, but metadata are provided using the standard RDF format and are completely logic independent.

[5] Another proof-assistant based on a different variant of the same calculus is Lego.

development of our tools helped us in identifying wrong decisions in the design of the DTD and in the implementation of the export module.

In 2002 we became members of the European IST Project MoWGLI (Math on the Web, Get it by Logics and Interfaces), which extends the goal of the HELM project and adds new requirements. Sustained by our previous experience, we are redefining the DTD for the Calculus of (Co)Inductive Constructions and we are reimplementing the exportation module, starting from the first version and adding several new features.

In the meantime in the HELM project we are now trying to export also the library of the NuPRL system, which is based on a variant of the Martin-Löf constructive type theory.

In the rest of the paper we will try to outline the principles that we have learnt in the last three years and that we are now trying to apply to our new implementations of exporting modules.

2 Tips and Pitfalls

The main self-assesment criteria for an exportation module is that the exported information must be 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.

Thus we must face at least the following issues: What information should be exported? What is the granularity of the generated files? How is the information described? (i.e. how should a DTD be designed)

2.1 What Classes of Information Should Be Exported?

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 you 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.

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). An usual operations on proofs that do not require additional information is proof improvement: 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 [6].

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, the advice is to

Make implicit information explicit.

– 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 [10]. Usually, though, they are not provided inside the system or they are just given as unstructured comments.
– A system dependent history of the operations that lead to the creation of a definition or theorem. For example, 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 often discharged during the lexical analysis of the input to the system; So they are likely to be unavailable inside the system itself.
– Parsing and pretty-printing rules. The existent systems only perform 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.

  To have the tactics into the library we need first to formalize the language (and the libraries) in which they are implemented and this is surely a complex task.

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 advice is

Develop one DTD for each class you 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[6]. 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. Thus, according to our first suggestion, we should split the functions.

2.2 How Should Files Be Organized?

The need to clearly identify the atomic components, addressed in the previous sections, is related to our next suggestion:

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

[6] 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 requested, it must be explicitly stated by means of explicit links.

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, just the proof objects of the Coq libraries and their inner-types, once saved as compressed XML files, require about 260Mb. It is not unusual to have single theorems that, exported in XML and compressed, require 24Kb. Thus parsing (or even lexically analyze) these files is an expensive operation. Of course we want to avoid as much as possible wasting 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 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 given suggestions 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.

2.3 What Theory Should Be Described?

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 the first time, we decided to be quite close to the implementation, even if some parts of what we exported were partially not understood. When we developed our proof-checker first and our proof-assistant later, we realized that we had no clear foundation for the theory we were 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 we wanted to experiment with other solutions. Let's see in detail a couple of examples.

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, a hole (called *metavariable* in the literature [11]) 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[7]. 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 first time we exported from Coq, we did not change the Coq representation. Later on, when implementing our own proof-assistant, we found out that our choice forced us to a treatment of metavariables similar to the one of Coq. Moreover the distinction between the two kinds of metavariables did not allow any progress in the proof[8]. So we defined a different internal representation of metavariables, following the second solution above, and we decided to change the DTD.

Now, which is the right format for describing metavariables in the 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.

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:

```
Definition H1 : Prop -> Prop -> Prop := [A:Prop ; B:Prop]A/\B.
```

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

8 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.

```
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[9].

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 longer 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.

If we stop reflecting on the real cause of the problems given by discharging, we will easily identify another severe problem that must be faced when exporting

[9] This is what we 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.

the information: 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 this 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, avoids 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.

3 Further Discussion and Future Work

The previous sections should make evident that there is a conflict between the two design principles "*what is expensive is worth saving*" and "*what is computable can be disregarded*". Indeed it is often the case that the computations that can be disregarded are also the expensive ones.

We recall that the two strong motivations for the second rule are: redundant information requires consistency checks; redundant information may require unwanted parsing of unuseful data. Our suggestion is to look for a compromise preferring to avoid redundant information when possible. When this is not the case, the expensive computed information should be stored in a different file, so that we can grant at least internal consistency, i.e. consistency of the data in the same file. Internal consistency is often enough for our needs. Unwanted parsing is also minimized in this way.

Future work will consist in trying to apply our guidelines to other proof-assistants and theorem provers, with the aim of integrating them in our distributed library.

So far we have not tried to apply our guidelines to export information from a Computer Algebra System (CAS). Our feeling is that some of the issues we faced would be irrelevant for Computer Algebra Systems. For example, the theory behind these systems is quite uniform and better understood, just being some extension of a first-order setting. Determining the information classes to export should present more or less the same problems. Probably, though, other issues may get much more severe and new ones will certainly arise: CAS are more imperative than proof-assistants and theorems prover, and a notion of library is much more difficult to define. Since the power of a CAS mainly derives from the procedures it implements and not from a library of pre-computed results, exporting a description of these procedures to the library could be compelling.

Since our experience derives from a long process of committing mistakes and finding solutions, we want to conclude with a final few advices: be ready to go back on your steps; and be always creative and flexible:

Sometimes it is better to avoid good suggestions and look for compromises!

4 Conclusions

We presented a collection of simple rules-of-thumb that we learned in the previous three years from small successes and lots of mistakes. Each rule was justified with small examples; their only aim was to give a grasp on the set of problems that must be faced by a designer and implementor of an exporting procedure from a proof-assistant to a distributed library of mathematical documents.

To our knowledge, this is the first paper in the literature with this aim, even if many other exporting procedures have been developed for several other systems in the past few years [7, 5].

Even if many of the suggestions may appear obvious, the temptation of sticking to the internals of the system when exporting the information is great. The experiences of the HELM project strongly suggest that the design phase of the DTDs requires a lot of time and a lot of thought; but the result is worth the time spent, since wrong decisions will greatly slow-down further developments.

Only time and further experiences can have the final word on our guidelines and solutions. Still we believe we have provided at least a comprehensive list of the common problems that must be faced.

References

1. M. Agrawal, N. Kayal, N. Saxena, "PRIMES in P", unpublished, August 2002, `http://www.cse.iitk.ac.in/users/manindra/primality.ps`.
2. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena, "Mathematical Knowledge Management in HELM", in On-Line Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), RISC-Linz, Austria, September 2001.
3. A. Asperti, L. Padovani, C. Sacerdoti Coen, Schena, I., "HELM and the semantic Math-Web". Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2001), 3-6 September 2001, Edinburgh, Scotland.
4. A. Asperti, B.Wegner, "MoWGLI - A New Approach for the Content Description in Digital Documents", in Proceedings of the Ninth International Conference on Electronic Resources and the Social Role of Libraries in the Future, Section 4, Volume 1.
5. O. Caprotti, H. Geuvers, and M. Oostdijk, "Certified and Portable Mathematical Documents from Formal Contexts", in On-Line Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), RISC-Linz, Austria, September 2001.
6. Y.Coscoy. "Explication textuelle de preuves pour le Calcul des Constructions Inductives", Phd. Thesis, Université de Nice-Sophia Antipolis, 2000.
7. A. Franke, M. Kohlhase, "MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems", to appear in Journal of Symbolic Computation.
8. J. Harrison, "Real Numbers in Real Applications", invited talk at Formalising Continuous Mathematics 2002 (FCM2002), 19th August 2002, Radisson Hotel, Hampton, VA, USA
9. M. Kohlhase, A. Franke, "MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems", Journal of Symbolic Computation 23:4 (2001), pp. 365 - 402.
10. Learning Technology Standards Committee of IEEE, "Draft Standard for Learning Object Metadata", July 15th 2002.
11. C. Muñoz, "Un calcul de substitutions pour la représentation de preuves partielles en théorie de types", PhD. Thesis, University Paris 7, 1997.
12. The OpenMath Esprit Consortium, "The OpenMath Standard", O. Caprotti, D. P. Carlisle, A. M. Cohen editors.