2007

Case Studies in Proof Checking

Robert Kam
San Jose State University

Running Head: PROOF CHECKING

Case Studies in Proof Checking

Robert C. Kam

San José State University

Introduction

The aim of computer proof checking is not to find proofs, but to verify them. This is different from automated deduction, which is the use of computers to find proofs that humans have not devised first. Currently, checking a proof by computer is done by taking a known mathematical proof and entering it into the special language recognized by a proof verifier program, and then running the verifier to hopefully obtain no errors. Of course, if the proof checker approves the proof, there are considerations of whether or not the proof checker is correct, and this has been complicated by the fact that so many systems have sprung into being.

Dr. Freek Wiedijk made a list in 2006 of all systems that had been "seriously used for this purpose" of proof checking and that had at least one attribute that distinguished them from the rest. It contains seventeen proof systems: HOL, Mizar, PVS, Coq, Otter/Ivy, Isabelle/Isar, Alfa/Agda, ACL2, PhoX, IMPS, Metamath, Theorema, Lego, Nuprl, Ωmega, B method, and Minlog (Wenzel & Wiedijk, 2002, p. 8).

The Flyspeck Project by Dr. Thomas Hales was one of the first required applications of the proof checker idea. Hales' proof of Kepler's conjecture in 1998, the statement that the "grocery store" stacking of spheres is the optimal way to conserve volume, required computer verification in parts of the proof. The referee committee checked much of it by hand, but due to the nature of the proof, it was time-consuming. After five years (not of actual checking, but overall time), the leader, G. Fejes Tóth, would only say that he was "99% certain" that the proof was correct (Hales, 2005, p. 1). In this instance, having a computer verify the entire proof instead would have produced quicker, more satisfactory results.

The two main challenges in using a proof checker today are the time needed to learn the syntax and general usage of the system and the time needed to formalize a proof in the system even when the user is already proficient with it. As mathematicians are not yet using proof checkers regularly, we wanted to evaluate the validity of this reluctance by analyzing these main obstacles. Judging by Dr. Wiedijk's *Formalizing 100 Theorems* list, which gives an overview of the headway various proof systems have made in mathematics, Coq and Mizar are two of the most successful systems in use today (Wiedijk, 2007).

Mizar has been around longer than Coq, having had its first version appear in 1975, but Coq currently has a larger active user base (Wiedijk, 1999, p. 14). We used Mizar version 7.6.02, with version 4.60 of the Mizar Mathematical Library (MML), and Coq version 8.1.

I simultaneously formalized two fairly involved theorems in these two systems while I was at approximately the same level of familiarity with each. I kept track of my experiences with learning the systems and analyzed their comparative strengths and weaknesses. The analysis and summary of experiences should also give a general idea of the current state of computer-aided proof checking.

1. The orbit-stabilizer theorem in Coq

When I embarked upon this first formalization, I would have described my skill level with Coq as "novice." I had proved a few simple things in Coq previously, so I had some experience with basic syntax. I knew the basic *tactics*, the interactive commands that one types into Coq to gradually reduce the goal (in Coq, a synonym for the current statement one is trying to prove) to completion: `elim`, `inversion`, `auto`, `simpl`, `red`, `split`, `intro`, `induction`. But there were many deeper facets of Coq which this formalization would be the introduction to: the various forms of the axiom of choice in the Coq standard library, the difference between `sigT` and `exists`, and the importance of `Opaque` and separating out sublemmas to deal with huge expressions in a goal, to name a few.

I took careful notes to chart my progress at learning this proof system. Afterwards, to analyze in some way the time and effort spent learning, I tried to classify the time spent into categories. I tried to separate out the basic tasks one engages in during a moderately complex formalization. Being a computer science major with a moderate mathematical background, I had experience with programming and university level mathematics, which helped, but I lacked knowledge in the theory and background of proof checker systems.

One aim of the analysis was to look at the system from the perspective of an ideal user. That is, if this particular proof checker became the standard for mathematicians, well known and regularly used, about how much time and effort would mathematicians expect to spend to formalize a theorem? We want to distinguish the time the system requires for the actual formalization from the time spent by a new user in learning the system. The analogy would be that no matter how experienced a C programmer is, she still must name her variables, allocate her data structures, and work out the logic of how to translate her human idea of processing the data into official C commands and syntax. However, unlike a novice programmer who is just learning C, she does not have to spend time flipping through tutorials and books; she is past that point thanks to her previous experience. She also does not have to write, for example, a quicksort function, as programming libraries will provide such a standard piece of code.

This programming analogy gave me my first few categories. One category would try to encompass just the rote, laborious work of converting one's mathematical ideas into the proper syntax that the Coq interpreter understands. Another clearly different category would be related to learning the system itself, figuring out the syntax. Yet another would try to capture the portion of time that I spent finding and learning how to properly make use of the work of others before me. Instead of a quicksort algorithm, I would be searching for and trying to comprehend theorems and definitions upon which I would build my own proof of the orbit-stabilizer theorem.

1.1. Categorization explanation

We describe the seven categories which capture the essential tasks a new user of Coq faces in more detail. Much of the explanation applies to the Mizar categorization also; we will note the differences.

1.1.1. Rote work

This is the simplest category and also the largest, in the sense that it takes up the largest proportion of total time in the formalization. As mentioned above, this category is meant to contain any busy, tedious work that does not involve much thought on the user's part. Translating human mathematical statements into Coq syntax is one part of this. For example, to state that a set is countable, to a person we might say something like "there is a surjection from the natural numbers onto this set," but in Coq we will assign specific names to all variables, convert loose English phrases into exact keywords like `exists`, `->`, and `Ensemble`, and fix simple syntax errors.

```
Definition cntabl(U:Type)(C:Ensemble(Ensemble U)):=
exists f:nat->Ensemble U,forall c,C c->exists i:nat,f i=c.
```

Figure 1.1.1.1. A definition of countability in Coq.

At first, this kind of work might seem to the reader to be fairly complex, not really falling under the category of "rote" or mindless work. The reader would be correct for the initial stages. It certainly takes thought to write statements in Coq, especially debugging the errors that arise when trying new syntax for the first time. However, once one has used a piece of syntax several times and figured out all the nuances, just as in a programming language, the translation of human thought to Coq statement (when the syntax is familiar) is really second nature. At a certain point, it is only fair to start putting most of this formulation and translation work into the rote work category. So, throughout this categorization process, I made judgment calls as time progressed as to what kind of work went where. The first time I tried writing functions with Coq's `fun` syntax, I ran into hosts of problems and it was certainly hard to figure them all out. But months later, writing a `fun` expression was like using `printf` in C, and I viewed it as busy work, with more difficult things to worry about.

Another major subcategory of rote work is the portion concerning Coq tactics themselves. Again, Coq tactics are the interactive commands that one types in to reduce goals of the proof to simpler and simpler forms until they are completely satisfied by known facts or assumptions. At first, my use of tactics consisted of blindly flailing at the goal with them, trying random commands in the hope they would miraculously solve the problem. I learned a few simple scenarios where this tactic worked or that one, and eventually I built up knowledge of the commonly used tactics to the point where I had mastered them. So the same scenario plays out; during this first major proof of the orbit-stabilizer theorem, it seemed as if an interminable time was spent grappling with these tactics, but as time passed, they became less daunting and more like busy work: just typing them in and doing away with goals as quickly as I could hammer out the tactics sequence. The first portion of time would go into a category of time spent learning tactics; the second belongs in this rote catch-all category.

Fixing common errors, formulating human reasoning in Coq syntax, and doing away with easier lemmas using tactics one is proficient with – those are the main types of rote work one does in Coq.

In Mizar, the analog to typing out the tactics as quickly as one can in easy Coq proofs would be the writing out of simple logical chains of statements and citing all the proper theorems. It is not hard work, but it is slow. One segment of rote work in Coq that has no Mizar analog is the act of, having completed typing out a definition or a statement of a lemma, sitting back and verifying that the written Coq syntax actually is what I intended or needed to write. This is because Coq syntax is simply more complex and deep than Mizar's, often having the writer delve deep into several layers of definitions in the middle of a statement, then rise back up and reaffirm himself with the bigger picture. This does not occur in Mizar, as statements are more flat and concise, dealing with one concept at a time, and the syntax itself is structured more similarly to English sentences, or at least logical statements, than Coq's syntax.

1.1.2. Learning Coq syntax

The category of learning Coq syntax is fairly broad. It encompasses the main ways I generally learn syntax, which is looking for an existing model, trying to adapt that model to my own uses (in other words, copying it but replacing the variables to fit my own situation), and finally fixing errors, trying tweaks, and just basically learning through trial and error how the syntax works. Again there is a difference in this category between Coq and Mizar. In Coq, we also include browsing the internet for helpful information people have written about using particular pieces of syntax, as this information exists for many constructions in Coq. I have used papers, tutorials, read others' forum posts and mailing list threads, and as a last but invaluable resort, asked people personally for help with Coq syntax. Mizar, however, has very little documentation beyond a handful of tutorials, the most helpful of which were notes from an introductory lecture in Dagstuhl by Piotr Rudnicki in 1997 and the two tutorials *Mizar: an Impression* and *Writing a Mizar article in nine easy steps* by Dr. Freek Wiedijk.

Note that this category is restricted to Coq *syntax*; user-defined terms, user-defined libraries, and the tactics are not included here.

For certain parts of the Coq system, determining whether or not they fell into this category was tricky. The result is that I define Coq syntax as the set of Coq primitives that one is likely to encounter in every user-defined library as well as every proof one sets out to formalize. They are essential and used often. We are getting a little into the nitty-gritty of Coq here, but, for example, I consider the term `sigT` part of Coq syntax, as well as the many formulations of the axiom of choice set out in the Coq standard library. Technically, both of these terms are user-defined; they are defined by members of the official Coq committee that designed and approved the Coq standard library. However, they are so basic that I see them having more in common with, say, the proper way to structure a `Fixpoint` definition, than with learning how to use the group theory lemmas in Dr. Loïc Pottier and Jasper Stein's combined Algebra and Linear Algebra library. (That is the user-contributed library on which I built the orbit-stabilizer theorem; it contains a lot of results of group theory and a predicate definition of subsets. From now on, I will refer to it as Pottier-Stein.)

On the other hand, terms like `Ensembles`, `Union`, and `Complement`, fundamental pieces of set theory defined in the Coq standard library, do not in my view fall into the category of syntax. While they are primitive, and integral to any theorem built on the set theory foundations defined in the Coq standard library, they are still set-theory-only terms, and unique to the Coq standard library. A user library that is formalizing some type of mathematics not based on ZF set theory, or that wishes to define its own formalization of set theory (as Pottier-Stein and the Constructive Coq Repository at Nijmegen, known as C-CoRN, library both did, predating this version of the Coq standard library), will not use these terms or any theorems associated with them. Also, the Coq committee has a history of revamping their standard library. Overall, learning these basic set theory notions has more in common with learning any other user-defined library's definitions and theorems than it does with figuring out basic syntax like `Fixpoint` definitions and `sigT`.

The main purpose of clarifying the distinction between spending time learning Coq syntax and spending time learning user-defined Coq libraries is to separate out the work that an experienced Coq user would have to do (the latter category) from the work a novice Coq user would have to do (both categories).

1.1.3. Experimenting with Coq tactics

The interactive portion of Coq is best understood through an analogy with programming. While most proof verifiers, including Mizar, are like structured programming in that the user prepares as much input as she wishes before running the Mizar compiler to point out errors, a Coq user enters commands, the tactics, into an interpreter that gives feedback after every command in real-time. As touched on before, the main ways I learned how tactics work are simply trying all of them until one worked (only in the cases where there are not too many options for arguments to pass), reading documentation on the internet, and searching for hints and ideas in others' usage of them.

I also include in this category any portion of the hashing out of a proof that did not strike me as rote and second nature, as this kind of extra-tactical thinking still centered around how to lay out the logical argument in terms of tactics. For example, I learned how to use a theorem called `NNPP` to do proof by contradiction in Coq. `NNPP` states that for a proposition `A`, (not not `A`) implies `A`. While `NNPP` is not a Coq tactic per se, it required a new way of using the interactive interpreter to accomplish a goal. I knew the idea of proof by contradiction before Coq, but the process of replacing the current goal with a goal of `False` to signify trying to prove a contradiction, and then applying theorems that implied `False` (negative statements) was a new way of looking at this old concept.

1.1.4. Browsing the user-supplied library

We now get to one of the main parts of formalizing a proof, building upon others' previous work. For example, in formalizing the orbit-stabilizer theorem, we first sought out a user-defined

library that defined groups and basic results of group theory. To do this, we had to browse the current repository of Coq user submissions, figure out which ones dealt with groups, and review them to select the best one.

This category distinguishes determining whether a branch of mathematics or a particular theorem has been formalized at all from the process of learning how to use such user-defined terms and theorems. It is meant to gauge the proportion of time in doing a formalization that one spends just browsing and interpreting others' work. How hard is it to search the global repository of Coq knowledge, at the moment?

In contrast to Mizar, searching the user-defined libraries for definitions in Coq employs reading what others have written in articles, mailing lists, and forums on the internet. In Mizar, to find out if a definition or theorem exists, we do text searches on the actual Mizar code of the entire database of user-submitted Mizar formalizations (the MML), because there is little human-language documentation written about Mizar on the internet or elsewhere. Though this is not the most user-friendly way of locating things, Mizar syntax is still more readable than Coq and lends itself better to straight text searching (such as `grep` in Unix and `findstr` on Windows). Browsing Coq user libraries is difficult, and any information that other people have written explaining what a library contains is invaluable.

1.1.5. Familiarizing with user-defined library

This category takes care of the act of learning how to use others' definitions, theorems, and lemmas for the first time. That means interpreting their variable names and choice of data structures, which may entail sitting and staring at complex expressions for several minutes, as well as, for tactile learners, loading in their file and instantiating the definitions and terms and experimenting with them. A useful technique is to state some very basic results that should be true about the terms if one's interpretation is correct, and doing a quick throwaway proof to verify one's intuition – "Is this supposed to be the definition of integral? Let me see if I can quickly prove that the integral of the zero function is zero."

If one were completely familiar with a particular library, from long experience or if one were the original author, this section would be completely empty, showing the advantage of experienced users and heavily-used standardized libraries.

1.1.6. Planning out proof strategy

The final category is sort of a rogue. For this orbit-stabilizer theorem as well as Markov's inequality later, I wrote a pencil-and-paper version of the proof beforehand, based on an example of the proof I found online. Carefully going over each step to be sure I fully understood the reasoning took time. But as I meant to simulate a mathematician who was already familiar with the proof she was trying to formalize, it would not be fair to confuse the time and work I spent

formalizing the proof in Coq with the time I spent learning the proof I was already supposed to know. With that in mind, I did not expect to spend much time on this kind of "human mathematical reasoning" at all once beginning the actual formalization. It turned out that I did, not for the details of the human reasoning proof, but for the details of the human reasoning proof properly processed into a form that could be entered into Coq.

To better explain what I mean, one example of this was when I had to define simple functions to create the notion of a limited Lebesgue integral to state Markov's inequality. In normal mathematics, the definition of a simple function is short: it is just a function with a finite number of values in its range. However, to put this notion into Coq, I had to define all the intermediate data structures involved. That means, since the simple function is essentially a partition of its domain into a finite number of pieces, each piece mapping to a specific single range value (real number), I had to create a finite list mapping some identification tag (natural number) to each partition piece, and a finite list linking those tags to their respective range values. Then I had to come up with the logic, the restrictive facts about these lists, that would make them actually represent simple functions.

So there was a significant amount of actual mathematics-related thinking involved, the fleshing out and filling in minute details of the conceptual proof written down on paper. Since the part relating to these data structures, and the associated work of planning out proofs involving them, was a direct result of the necessity of having to be extremely detailed and precise when formulating ideas to a computer, I considered this worthy of being included in an analysis of the work one has to do to formalize a proof in Coq, and thus gave it its own category. This also seems to me to be a category of work which does not depend on the user's expertise with Coq or any other proof system, at least in the sense that no matter how well-versed one is with the system, one will still have to do this kind of detailed formulating of and thinking about data structures for any proof one wishes to check.

1.1.7. Housekeeping

The final category is a miscellaneous category to catch various logistics of setting up and running the Coq computer program itself. Installing and configuring Coq on one's personal computer, updating it, and downloading user-defined libraries and properly installing them comprise most of this category.

1.2. Breakdown of formalization time

For our first analyzed formalization, we chose a medium-sized theorem, the orbit-stabilizer theorem of group theory. When talking about a *left action* of a group G on a set S, a function

from $G \times S$ to S that satisfies certain properties, group theorists define two sets for an element p of S. The orbit of p is the set of all possible results of the left action when it is applied to an element of G and p. The stabilizer of p is the set of elements of G that when applied with the left action to p, produce p again. The orbit-stabilizer theorem states that

$$|G| = |\text{the orbit of } p| \bullet |\text{the stabilizer of } p|.$$

(I borrow the notation used by Rahbar Virk in his lecture notes.) This is actually the statement of the theorem when G is finite, which is the version we formalized, on the assumption that restricting to finite groups was simpler (an idea this formalization cured me of).

We kept an approximate log of the time spent in various tasks during the formalization; the consolidation of it into several broad categories is listed below. The entire formalization process took about seven and a half weeks, working on average three to four hours a day.

Category of work	**Approximate work (hrs.)**	**Percentage of total work**
Learning Coq syntax	18	9%
Experimenting with Coq tactics	27	14%
Browsing the user-supplied library	22	11%
Familiarizing with new structures, terms of library	60	31%
Actual formalizing	48	25%
Planning out actual proof strategy	14	7%
Housekeeping (installing and setting up Coq)	3	1%

Table 1.2.1. Breakdown of Coq formalization time.

We can consolidate these categories into the three main categories with which a new Coq user would be concerned. How much time is spent on learning a new Coq library (the kind of work a professional user of Coq would have to do also) versus learning the syntax and user interface of the Coq software itself? Do these two areas take up the bulk of the time to complete the proof, with the actual formalization being a negligible amount of time?

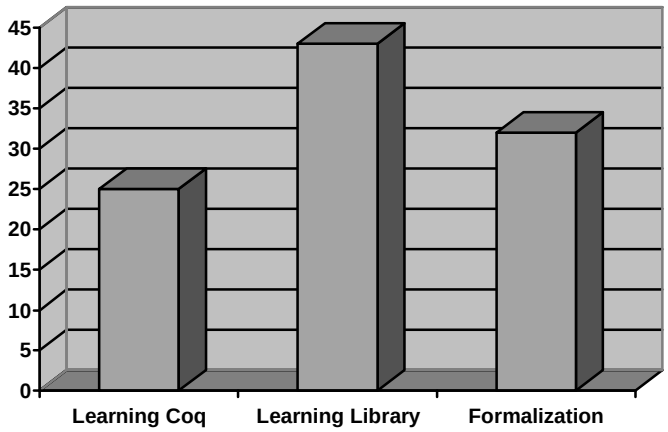

Figure 1.2.1. Simplified breakdown of Coq formalization time.

At this stage of Coq experience, formalization remains a significant part of the time invested, almost a third according to the record. However, formalization seems to go faster and faster the more tactics of Coq one becomes familiar with. Likewise, getting used to Coq's style or paradigm of operation, the "learning Coq" portion, is probably a one-time cost. An advanced user probably spends the bulk of his time in a new formalization with browsing the new user-supplied library, getting used to its structure, new terms, and just what lemmas are available. After getting up to speed with the new library (usually done ad hoc, by experimenting with sections of the library as they become necessary in each step of the proof), the actual formalization process goes by quickly. In fact, not only the time is divided in this way, but often the actual thinking is mostly concentrated in figuring out the library too. Formalization becomes mechanical after a while – kind of like the difference between writing a computer program and debugging it.

While this formalization did take longer than expected (the human-readable proof, jotted down, took up about a page), afterward, I felt that my comfort level with Coq had increased from "novice" to "intermediate." I felt that this one, fairly sized proof contained enough aspects of Coq formalization that the next proof would go by with fewer hurdles.

1.3. Estimation of amount of work done by the library

To estimate what percentage of the work was done for us by the Pottier-Stein library, we separate the theorem into its component lemmas and classify them according to difficulty. An "A" lemma is basically trivial to prove, "B" is of medium difficulty, and a "C" lemma is arduous to prove; we would really like as many of those to be existing parts of the library as possible.

We also ask of each lemma if it "should be in" the library. This is a fairly optimistic definition of shouldness. We approach answering the question from the perspective: "If the library were comprehensive and 100% ideal, this lemma would exist for us." That is, it is *possible* that a library creator would have thought that this lemma might be useful and general enough to be worth including in a library. That does not mean that in a real-world library the creator should really have included it for reasons of clutter or time. Also note that if the lemma is of size A or B, it can be reasonable to expect a competent Coq user to be able to construct it on the fly.

Likewise, an answer to "should be in" of "no" means that we expected when embarking on the formalization that this would have to be proven by us. We would be very surprised if it actually were already included by a general-library-writing author. An example would be the definition of "left action of a group on a set," one of the foundations of the orbit-stabilizer theorem. This kind of map has applications beyond the orbit-stabilizer theorem, but it is obscure enough that we expect it not to be defined, especially as the focus of the Pottier-Stein library is linear algebra, not group theory.

The point is to get a gauge of how many holes in the proof we had to fill in ourselves. Naturally, we expect to have to fill in *some* holes; otherwise there would be no theorem to formalize. The amount of holes is not to say the library writer is bad or in error. As mentioned, size considerations prevent any library from having the answer to every step at every time. The point of this exercise is simply to gauge or evaluate the "completeness" of a real-world library applied to a real-world problem. For anyone approaching the idea of formalizing a particular theorem in Coq, what amount of help can he expect?

The italicized portions indicate the holes we had to fill in. A lemma is italicized if it were category B or C (had a significant impact on the formalization), and if it "should have been" included.

Name	Category	Should be in?	In library?
Axiom of excluded-middle	A	yes	yes
Axiom of choice	A	yes	yes
Definition of union	A	yes	yes
Definition of subset	A	yes	yes
Definition of power set	A	yes	yes
Membership is compatible (`in_part_comp_l`)	A	yes	yes
$A \subseteq B$ and $B \subseteq A$ implies $A = B$	*B*	*yes*	*yes*
Subset of empty set is empty	A	yes	no
Nonempty iff contains an element	*B*	*yes*	*no*
Set equality is transitive (`Trans`)	A	yes	yes
Natural number addition is commutative	A	yes	yes
Natural number addition is associative	A	yes	yes
Natural number addition is compatible	A	yes	no
Natural number is positive implies predecessor	A	yes	yes
Conversion of type (full) (`seq_set_n_element_subset`)	*B*	*yes*	*yes*
Conversion of type (full) for finite set	*B*	*yes*	*no*

Table 1.3.1. Lemmas of basic set theory.

Name	Category	Should be in?	In library?
Cardinality definition	*B*	*yes*	*yes*
Cardinality is zero iff empty	*B*	*yes*	*yes*
Cardinality is unique (`has_n_elements_inj`)	*C*	*yes*	*yes*
Cardinality is same implies bijection	*B*	*yes*	*no*
Cardinality decremented by removing an element	*B*	*yes*	*no*
Cardinality of subset is not greater	*C*	*yes*	*yes*
Map constructed from function definition (`Build_Map`)	*C*	*yes*	*yes*
Map restricted to subset of domain	*B*	*yes*	*yes*
Map is injective implies it has inverse	*B*	*yes*	*no*
Map composition definition	*B*	*yes*	*yes*
Map exists of set onto subset	*C*	*yes*	*no*
Inclusion-exclusion principle	*C*	*yes*	*no*
Sum of a sequence definition	*B*	*yes*	*no*
Sum of a sequence is compatible	*B*	*yes*	*yes*
Sum of a constant sequence equals multiplication	B	no	no
Sum of a sequence with one element changed	*B*	*yes*	*no**
Sum of a sequence ignores zeroes (`sum_omit_zeroes`)	*B*	*yes*	*yes*
Pairwise disjoint definition	A	yes	no
Pairwise disjoint implies cardinality of union is sum	*B*	*yes*	*no*

Name	**Category**	**Should be in?**	**In library?**
Partition definition	*B*	*yes*	*no*
Partition union incremented by adding element	B	no	no
Sequence element has an index	A	yes	yes
Sequence elements have index after a removal	*C*	*yes*	*yes*
Conversion of type (range of sequence) (`seq_set`)	*B*	*yes*	*yes*
Conversion of type (same length) (`cast_doesn't_change`)	*C*	*yes*	*yes*
Change an element of a sequence definition (`modify_seq`)	*C*	*yes*	*yes*
Change an element is compatible	A	yes	yes
Change an element can be undone	*B*	*yes*	*yes*
Change an element affects head (`modify_hd_hd`)	A	yes	yes
Change an element affects tail (`modify_hd_tl`)	*B*	*yes*	*yes*
Change an element affects only that index (`modify_seq_modifies_one_elt`)	*B*	*yes*	*yes*
Head tail definition	*B*	*yes*	*yes*
Head of a sequence in range of sequence	A	yes	no
Head not equal to and injective means element in tail	*B*	*yes*	*no*
Remove an element maintains injectivity (`omit_preserves_distinct`)	*C*	*yes*	*yes*
Remove an element and injective means entirely removed	*C*	*yes*	*yes*
Remove head leaves sequence (`Seqtl_to_seq`)	*B*	*yes*	*yes*
Remove head means elements that remain are in tail	*B*	*yes*	*no*
Remove an element means range is subset (`omit_seq_in_seq_set`)	*B*	*yes*	*yes*
Finite-domain map has finite range	*C*	*yes*	*no*
Finite set has bijection with some natural number	*C*	*yes*	*no*

Table 1.3.2. Lemmas related to sequences and cardinality.

*How changing one element of a sequence of natural numbers changed the sum of the sequence was proved as `sum_modify`, but since it was defined to work on elements of `Abelian_Group`, and cardinality was defined on `Nat`, we had to write a simpler version specifically for sequences of `Nat`.

Name	Category	Should be in?	In library?
Left action definition	A	no	no
Left action compatible	A	no	no
Left action regular	B	no	no
Group operation definition	*C*	*yes*	*yes*
Group operation associative	A	yes	yes
Group operation identity	A	yes	yes
Group inverse property	A	yes	yes
Definition of orbit	A	no	no
Definition of stabilizer	A	no	no
Definition of $H(x)$	A	no	no
Union of $H(x)$s $\subseteq G$	B	no	no
$G \subseteq$ union of $H(x)$s	B	no	no
$H(x)$ • stabilizer in stabilizer	B	no	no

Table 1.3.3. Lemmas directly related to the orbit-stabilizer theorem.

Category of lemma	Total number	Number already done	Percentage already done
B	26	15	58%
C	13	9	69%

Table 1.3.4. Estimate of amount of work done by the library.

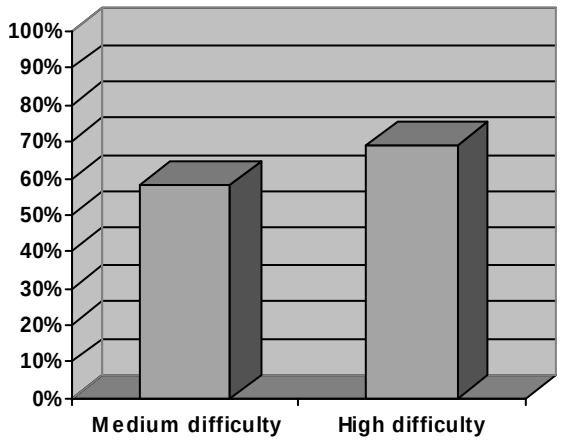

Figure 1.3.1. Percentage of lemmas already completed by the library.

My personal impression was that I was very satisfied with the Pottier-Stein library in terms of which lemmas were provided to me, especially dealing with sequences. The slightly lower percentage of medium difficulty lemmas which had been completed for me reflects the toolbox philosophy of library construction, where a minimum of lemmas are actually completed, making

it easier for a new user to browse the library and get a general feel for the material it covers. Lemmas are building blocks of theorems, rather than databases meant to instantly satisfy many similar cases. The analogy would be a library of basic Unix file commands like `fopen()`, `fseek()`, `fread()`, `fwrite()`, and such, rather than `fcompress()`, `freadfirsthalf()`, or `fseparatefileintopieces()`.

Again, real-world considerations mean that one cannot really expect a library to satisfy over 90% of the "high difficulty" lemmas of a proof immediately, or even provide a good percentage of the building blocks that would make this high difficulty lemma feel more like a medium difficulty one. If it did, it might mean that one's theorem was nearly already proven, and as proof formalizers we naturally direct our attention toward theorems which have not been addressed yet. The only theorem which I was a little disappointed had not been included was the inclusion-exclusion principle, or a simplified version of it (we used the simplification of only working on partitions). However, again I note that this library focused on linear algebra.

1.4. One difficulty for a new user of Coq: inability to look at old proofs for hints

A large portion of the work for a new Coq user is learning the basic strategies of attack to a goal. Some of these come very quickly, for example, figuring out that `elim` is the proper tactic to use to "instantiate" a variable locked up in an `exists` hypothesis. This is a basic tool and would appear in Chapter 1 or 2 of a Coq textbook. However, in general, the process of finding these basic strategies of attack is usually through trial and error – from essentially copying and pasting the strategies used by other proof authors in published work.

One example is when I was trying to define the orbit of a point *p* in a set *S*. I had invoked `Build_Predicate` (the operator for defining new `Predicates`) and now the goal was reduced to "`pred_compatible` [something]." I did not know what that term meant or how to prove it. But I was well-versed enough in Coq to know about the usefulness of `unfold`. So that was my first idea, and I `unfold`ed `pred_compatible`. That produced a new goal of the form "`exists g,` [something]." At the time I did not know how to solve this goal either. I knew I wanted to explain to Coq, "Yes, I have an element *g* that satisfies what you want," but did not know the command to say that. So what was I left with? The time-honored strategy of going to others' work for hints.

In the end, I found the proper tactic by manually typing out the proofs of some other simple `Predicates`, `empty` and `full`, defined by Dr. Pottier himself in the very Algebra library of his I was learning how to use. But this was after some browsing of `.v` files and more or less random selection of existing proofs to try and manually type out for myself: a clumsy way of looking for help. The problem is that there is no way, from browsing any number of `.v` files, to know when in a particular proof the user was actually faced with a goal of the form "`exists` [something]"!

This is unique to Coq, because Coq proofs have the goal "invisible" at all times. A proof in Coq is a sequence of commands to the interpreter, with no information about what the environment looks like at each step in the sequence (Wiedijk, 1999, p. 13). Mizar, on the other hand, does not use this interactive style. Statements are asserted one by one, each a refinement of the previous one, until the final statement which is the theorem itself. It is thus easy to do a text search of Mizar files to find a similar quandary to one's current situation, and then see exactly what a more experienced user did in that situation.

One solution would be to have a command-line utility that does what `coqc` (the command-line Coq compiler) does, but even more. It would compile the proofs in a `.v` file, but at each step, save a snapshot of what the goal looked like at that time. Then it would simply output all this data to another text file. Of course, the result would be huge, but if it is feasible to have an entire library, or useful section of library, snapshotted in this way, one would be able to text search the result a là Mizar.

Instead of text search, it may be possible to automatically pattern-match one's current goal against one of these huge text files. The `auto` tactic itself is a pattern-matcher, so this would be more or less an extension of `auto` applied to a very large "library" of goals to match against. However, this has the same concern as any other pattern-matching algorithm: exponential running speed.

Rather than get too fancy, why not let the human user retain some of the work. If the text search adopted the simple criterion of "match two keywords in my goal in any order," the text search would return perhaps a few hundred results, but the user could filter out the useful results by inspection fairly quickly. For example, if the user faced the goal of `in_part (subtype_elt a) A`, she could search on the two keywords `in_part` and `subtype_elt`. She would mentally toss out results such as `subtype_elt x =' y /\ in_part b B` and zero in on a true match. This kind of search is already possible with `grep` and similar utilities.

1.5. Strict typing in Coq leads to library incompatibility

As I surveyed the database of user-supplied libraries and compared them with the growing "standard library" produced by the official Coq designers, it struck me that most user-supplied Coq libraries were already obsolete in the sense of being compatible with the Coq standard library.

Coq has a centralized standard library overseen by the designers of Coq itself. Throughout Coq's history, each revision has added significant portions to this standard library. For example, the most recent revision from 8.0 to 8.1 added finite sets and lists. The Pottier-Stein library is now incompatible with the current standard library. It defined its own concepts of `Predicates`,

Setoids, and seqs as a necessary foundation. The Coq standard library only recently added Ensembles and sequences (FSetLists) to serve essentially the same purpose.

For another example, the extensive C-CoRN (the Constructive Coq Repository at Nijmegen) library, which includes a formalization of the Fundamental Theorem of Calculus, built its own notion of setoids, CSetoids, from the ground up. A user wishing to use C-CoRN's statement of the Fundamental Theorem of Calculus in verifying another theorem would either have to rewrite C-CoRN's work in terms of Ensembles or avoid using other user-supplied libraries not already based on C-CoRN's CSetoid foundations. Because most Coq formalizations, now that version 8.1 has been released, will now be based on the new Ensembles, in some sense C-CoRN's formalization has been made obsolete.

All proof verification programs face this issue, but Coq especially so. Coq is fundamentally abstract, being based on the Calculus of Inductive Constructions. The building blocks are Props and Sets. One can build practically anything from these building blocks; set theory is merely one use of this foundation. Since a proof in Coq is essentially equating a very complex type with the type True, equality must be defined for many, many different *Type*s. For example, in Pottier-Stein, there is the concept of a setoid S and the setoid (full S). (full S) is the "full" subset of S, the entire set S. However, they have different, incompatible types (S is whatever type S happens to be in the current context, and full S has type part_set S, or "subset of S"), and one cannot even pose the question "Does S equal (full S)?" because of that. The C-CoRN developers noticed a similar problem when defining their own foundations, and as a result bounced between defining their set theory foundations in the Prop domain, then defining them in the Set domain, and finally settling on a hybrid (Karrmann, 2005, para. 4).

Another example I personally encountered was the two formalizations of the cardinality concept: cardinal in Dr. Loïc Pottier's Algebra section and has_n_elements in Jasper Stein's Linear Algebra section. Although Mr. Stein built his Linear Algebra section on top of the Algebra section, he found Dr. Pottier's cardinal definition inadequate, being a resident of the Prop domain with no connection to concrete Sets. He defined has_n_elements to contain an exists statement from which one could "instantiate" a bijection between the natural number *n* and the set with cardinality *n*. Because his definition required the definition of a finite ordinal (fin *n*), which did not exist in the Algebra library, the lemmas referring to cardinal could not be used with statements involving has_n_elements. I had formalized some work using a lemma from the Algebra library called cardinal_image_injective. When I later switched over to has_n_elements, to make use of some of Mr. Stein's lemmas there, I had to write a new version of cardinal_image_injective, as well as adapt all the work I had done that had cardinal_image_injective as a basis.

Mizar, in contrast, assumes from the start that its users are only interested in ZF set theory with the axiom of choice (Wiedijk, 1999, p. 8). Since its first release in 1975, the Mizar designers have not changed this assumption (Matuszewski & Rudnicki, 2007, p. 3). It permits only specific avenues of defining new terms or capabilities: definitions and clusters, primarily. This restriction of viewpoint allows a simpler typing framework (only one notion of equality) and promotes interlibrary compatibility.

In closing, the writer of a Coq library must stay abreast of the latest news in the Coq community to know which libraries to use as his foundation. Just because a theorem has been proved using *some* library does not mean that theorem is conquered for the whole proof verification community: it will only be useful as long as the libraries used to prove it remain in use.

1.6. Interface concerns

We describe issues we faced with the Coq interface as a new user.

1.6.1. Quotations

One small issue that nonetheless led to the loss of a good deal of work was when at one point I thought I had actually crashed the Coq interpreter. It seemed to be "dead" and would not execute commands or print error messages no matter what I typed. I basically saw:

```
proof_name <
proof_name < aaa
proof_name < a
proof_name < .
proof_name < . a. a.
```

Figure 1.6.1.1. Dead interpreter?

Eventually I resigned myself to restarting the Coq interpreter, losing my work for this session (Coq work is not saved until a proof is completely closed off, either by "giving up" by admitting it as an axiom or successfully completing it). It was then that Coq printed `Syntax error: Unterminated string` as I returned to DOS. Perhaps Coq could display a prompt to reflect that one is currently inside a quote, especially as I was not aware that Coq recognized quotation marks and used them for anything.

1.6.2. Refolding "`->False`"

`unfold` is a common and useful tactic, but `fold` is more mysterious. Sometimes it undoes an `unfold` and sometimes it does nothing. One of the cases in which this caused trouble was when I defined a lemma with the notation ~, the shorthand for "`->False`." When faced with a goal of `False` (proof by contradiction), one can only apply a hypothesis that ends in "`->False`," so most lemmas must have their "~" `unfold`ed before they can be used. Unfortunately, the process does

not work in reverse, so I had to rewrite my lemma in unfolded form. Had I been a library writer, this might have meant breaking the work of previous users of my library.

1.6.3. Operator precedence

When using `min`, the keyword for taking the `group_inverse` of an element, I was surprised to find that it had a lower precedence than the group operation. While this is an issue for the library writer, not the designers of Coq, it nonetheless caused me to have to back up and redo the simplification of a long expression because I could not attempt to apply the group inverse property "`min y +' y =' (monoid_unit G)`" (more commonly $y^{-1} \bullet y = 1$) until I had reduced the lengthy expression `f(couple y(f (couple(min y +' subtype_elt h'')p))) =' f(couple y p)` down to this final fact. This is a consequence of the proof-by-reduction style of formalization in Coq, where the user reduces the final, complex goal to trivial statements, rather than introducing many simple statements and tying them together to build up to a complex goal (Wiedijk, 1999, p. 14). At any rate, only at the very end was I able to see that I was trying to prove a false fact: $(y \bullet y)^{-1} = 1$, thus the need to restart.

This is fundamental to the design of Coq, however, and I do not think it is worth adding a new feature just to deal with this. A user simply should be aware of this fact and check the precedence of operators in his expressions before embarking on simplifying a complex expression. However, one feature worth adding to Coq that would lower the chances of running into this pitfall (among others) is the idea of a save point.

1.6.4. Save points

If there were one single user-interface feature I could add to Coq, it would be the ability to create a save point. A save point would be the current location on the Coq command stack plus a snapshot of the current Coq global environment (external `Lemmas` and `Axioms` temporarily added to the environment). It might be a surprise to a non-Coq-user that the command-line interpreter does not have this ability, but in fact the usual mode of operation is to complete the entire proof of a lemma before Coq will return the entire "source code" of the proof, the list of commands to the interpreter that make up the proof, at which point the user can copy and paste that output to a `.v` file for saving. A save point would give the user the peace of mind to experiment, by for example assuming facts temporarily to jump ahead to a later point in the proof, to test whether the current path one is thinking of really does lead to the ultimate goal. After verifying this, the user could then undo everything back to that save point, and not worry about erroneously assumed facts or temporary variables floating around.

This feature would also avoid the problem of formalizations that are too large for the window buffer. There were times I had to redo entire sections of proof because the top portion of my work scrolled off the edge of the DOS window when I was done. (Coq's formatting of user commands adds many carriage returns.)

It would also help in situations like the following. At one time I was working on a lemma dealing with removing an element from a sequence. There were two nested excluded-middle cases, so I had to prove a certain fact for four different cases:

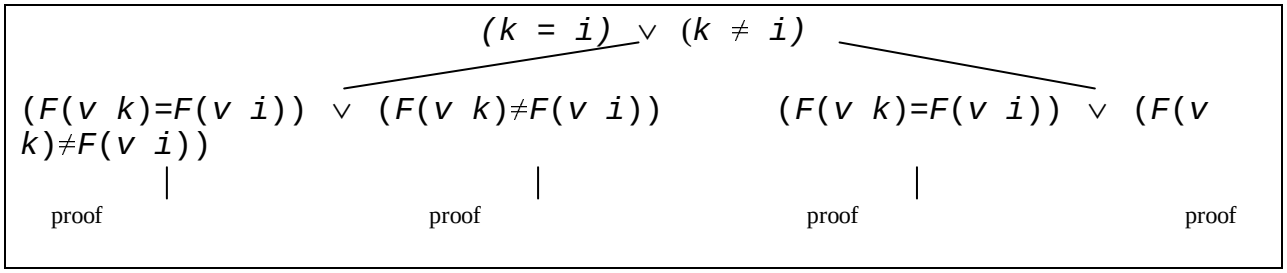

Figure 1.5.4.1. Four similar cases as part of one proof.

However, it turned out that two of these four cases had basically identical proofs! Unfortunately, since I had no way to look at previous work, I had to feel my way through the argument from scratch again.

One may ask why not simply keep a record of one's work from the beginning of the proof in a separate text editor as one works simultaneously in the other Coq window. The problem is that using Coq's interactive commands is often exploratory, and one types a rapid succession of commands just to see if a particular branch of argument will work. Frequently one goes down several paths and must `Undo` away wrong paths before finding one that actually worked. It makes it tedious to keep a copy of the current stack by hand.

2. The orbit-stabilizer theorem in Mizar

I undertook this formalization simultaneously with the Coq formalization, by interleaving my work on this theorem in the two systems.

2.1. Breakdown of formalization time

We do a breakdown of the total time spent on formalizing the orbit-stabilizer theorem in Mizar similar to the previous Coq analysis. This formalization took about four and a half weeks, working two to three hours on average a day. As with Coq, I had had some experience with

Mizar already before starting on the finite orbit-stabilizer theorem. I had formalized the statement that the limit of (*sin x*)/*x* as *x* approaches 0 is equal to 1. From that exercise, I learned how to write a basic Mizar file, compile it, and include files from the Mizar library, the MML. (Because Mizar user-defined libraries, although written by different authors, usually are good about building off the same foundations, one comes to view the MML as a single unit.) I learned the systematic, step-by-step style of Mizar formalizations, but none of the advanced features like defining one's own terms, or even how to do proofs by induction or contradiction.

The categories are mostly the same as those for the Coq formalization. In any case, the intent is the same: to distinguish the time spent on learning Mizar and the time spent on necessary work that even expert Mizar users must do (familiarizing oneself with new theorems and terminology, and the actual typing out of all the proof steps). The categories have similar general intents and meanings to the Coq categories. The "actual formalizing" category is the rote work of typing in steps and fixing common syntax errors. As in Coq, after a while this work does not need much real thought, and has the potential for computer optimizations or automation. "Learning syntax by trial and error" in large part refers to the process of using an example from a user-defined library as a guide for the first time one has to use a new piece of Mizar syntax, for example, how to define a new term referring to the left action of a group on a set and its properties of identity and associativity.

2.1.1. The environment category

One difference from the Coq division of labor, as mentioned before, is the lack of a category related to learning the Coq tactics. Another is the addition of a category specifically related to the Mizar environment.

Although fixing errors related to the Mizar environment – the `vocabularies`, `notations`, `constructors`, `registrations`, `requirements`, `definitions`, `theorems`, and `schemes` lists at the top of every Mizar file – could be grouped under rote formalization time, or seen as part of learning new user libraries, I feel it deserves its own category. Even after working with the environment for a long time, I still do not know how it works – I simply know some strategies for fixing the errors that appear whenever I import new definitions. In Dr. Wiedijk's Mizar tutorial *Writing a Mizar article in nine easy steps*, the section on the environment is the second longest section, taking ten of the paper's fifty-four pages.

2.1.2. Judgment calls and categorization

Although I took careful notes as I proceeded with the formalization, going back and classifying all my work as falling into one category or another at the end required some judgment calls. Suppose I go to use a new term from the MML, `integral` (the Lebesgue integral defined on simple measurable functions), and get the familiar `*103` typing error. (`*103` means Mizar did not recognize the functor, which often means one is supplying the wrong types of arguments to a

functor keyword.) I solve the error without too much difficulty by carefully testing the types of each argument to the `integral` term to narrow down which argument causes the error, and then finding out what type it *should* have (by checking the original definition) and adding steps to cast that argument to the needed type. Do I classify that as rote formalization, or as part of learning the new `integral` term (which would then contribute to the time tallied toward learning the MML)? I actually decided to classify these sorts of activities as rote formalization and not related to the MML, and here is why. The above process was arrived at through experience. With the additional step of, if the types of all arguments turn out to be correct, merging the relevant MML file's environment with my own to ensure that the proper MML files are named in the proper places for use of this term, the process solves 99% of `*103` and `*102` errors in a reliable and systematic way.

So for the purposes of differentiating between rote work where I do not learn anything new about Mizar and exercises and experiences that force me to learn something new that I have never encountered before, the aforementioned process clearly falls in the first category.

2.1.3. Results

Now let us look at the observed time breakdown for the orbit-stabilizer theorem in Mizar.

Category of work	**Approximate work (hrs.)**	**Percentage of total work**
Learning Mizar syntax by trial and error	14	16%
Searching libraries for examples of new syntax	8	8%
Familiarizing with user-defined libraries	16	18%
Searching for existence of terms or theorems	11	13%
Actual formalizing	26	30%
Fixing the Mizar environment	4	4%
Logically planning out the proof itself	10	12%

Table 2.1.3.1. Breakdown of formalization time in Mizar.

One interesting point is that I spent a significant amount of time (13%) simply searching the MML to see if a term or theorem is defined already. Again, since searching the internet for hints on what has been defined in the MML does not yield much, the best method is `grep` search with a dash of creativity, done over the whole MML (Wiedijk, 1999, p. 33). In searching for a theorem that says "the subset of a finite set is finite," one might search the whole MML for the string `A c= B implies`, and if that fails, check if there is an actual keyword `finite` that an author may have used to write his theorems about finite sets. As a last resort, one can simply scroll through the entire MML file list, pick out good candidate files based on their eight-digit

DOS name (here `CARD_FIN` seems good), and browse them from top to bottom, looking at and interpreting theorem statements and definitions manually.

Searching for examples of Mizar syntax, on the other hand, can be a bit trickier. When I was looking for an example of how to define a new operation, the left action of a group on a set, there was not an exact Mizar keyword I could search for like `define_binary_operator`. Eventually I realized that the normal binary operation on groups itself would be a good model: a binary operator from a group to itself, with the properties of having an identity element and associativity, was pretty similar. For ideas on how to make the operation take operands of two different sets (as the left action does, while the group operation does not) I could go to a familiar operation that works on two different sets: raising a real number to a natural number power. (In this case, as often, a definition I had seen in passing while looking for some *other* definition or theorem many days ago would come to mind, and be of use, later.)

Now we take a simplified look at the division between the three categories of work, that is, the work to learn Mizar itself, the work to familiarize oneself with a new section of the MML, and the rote formalization work:

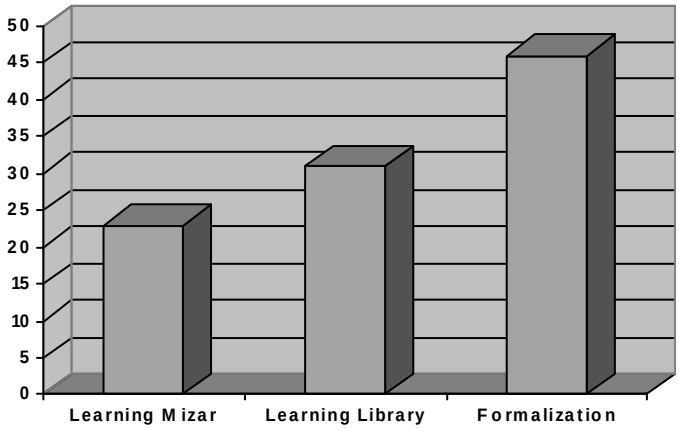

Figure 2.1.3.1. Simplified breakdown of Mizar formalization time.

This is a relative graph, of course, and does not mean that the pure formalization aspect is more time-consuming in Mizar than in Coq. This graph simply reflects that Mizar has a more limited, simpler syntax than Coq. The number of "general categories" of syntax and techniques one needs to learn to use Mizar feels more limited and manageable. Besides very basic syntax like ending statements with semicolons, surrounding all proofs with `proof` and `end` brackets, and the *mizar_file*:*theorem_number* syntax for citing a theorem, the major techniques one needs to learn are few. Important ones are the `deffunc` keyword for defining an "out of the set theory" function, the `Function` keyword for defining a function "in the underlying set theory" (Wiedijk, 1999, p. 4), and the proper structuring of proofs by induction and contradiction (mostly getting one's `end`s to line up with one's `per cases` and `let` blocks). Coq, by contrast, has the feeling of great depth. After struggling with a new concept in Coq, slowly gaining a working understanding of it, and finally applying it, it often seemed that that only opened a new door to a

new, equally difficult, concept, that I could not even begin to work with until mastering the previous layer.

Although this is a subjective criterion, it speaks to the difficulty of learning Coq on one's own that I sought help from real users of the Coq system on several occasions: seven to be exact, by requesting help through email, mailing lists, and forums. I kept this a last resort to keep my experience of the two systems as uniform as possible, but here I had no choice. In each of these cases, I literally felt that I could not progress on my own without external human help in answering a particularly confusing error message or roadblock related to the underlying theory of Coq. At least, I could not figure it out without spending an indeterminate amount of time. With Mizar, although there were several tricky problems involving syntax and the environment, the available papers, internet searches, and trial and error eventually produced a solution.

A further way the simplicity of Mizar syntax helps the new user is shown in my experience of one of the most tricky parts of Coq, the `if-then` construction. Unable to find information about others' experience with this piece of syntax, and unable to figure it out on my own, I had to start emailing people for help. The problem was that even if there were some help online for this construction, I would not find it because of the commonality of the words "if" and "then." With Mizar, although I would too have difficulty trying to find help for the `if-otherwise` construction for `deffunc`, Mizar syntax is deducible enough that I would eventually be able to figure it out by trial and error and deduction. This is just a microcosm of the continual difference in syntax difficulty I find in Coq and Mizar, and its consequences.

Perhaps the highest-level comparison of interest we can do is the total time spent on each formalization of this same theorem. We can look at it from the perspective of a new user, who takes into account time spent on learning the Coq and Mizar systems themselves, and from an expert user with and without the benefit of being familiar with the relevant user-defined libraries.

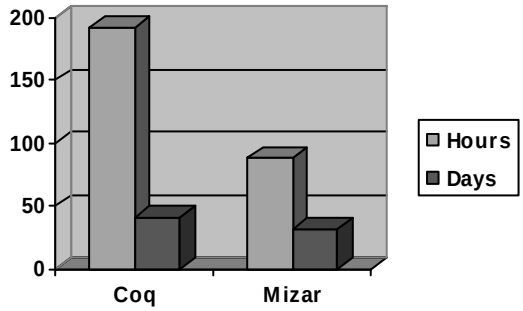

Figure 2.1.3.2. Formalization time in Coq and Mizar (new user).

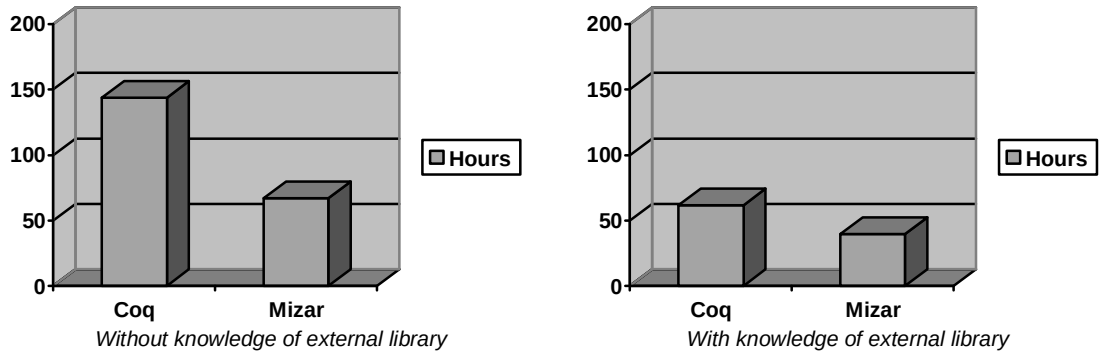

Figure 2.1.3.3. Formalization time in Coq and Mizar (experienced user).

When we attempt to separate out just the formalization aspect, the time difference between Coq and Mizar narrows. The experience of the rote formalization portion is again very different between the two, and the perceived difficulty in Coq is higher, but in the end the two systems are comparable.

2.2. Estimation of amount of work done by the Mizar MML

We estimate the proportion of work done by the Mizar MML in the same way we did for the Coq user-supplied libraries, with the same general classification of lemma difficulty. However, what made a particular lemma difficult was slightly different in the Coq world than in the Mizar one. In Coq, difficulty usually stemmed from the extremely unwieldy and complex expressions that would unfold, and some rather art-like tactics of dealing with them. In Mizar, most of the difficulty was due to learning the syntax, as many proofs seemed to go down to the nuts and bolts of the definition, no matter how many helpful lemmas were applied to simplify the task. Yet the exercise of sorting out all the details to massage one's hypothesis and assumed variables into the exact form desired by the Mizar theorem was more time-consuming than mentally taxing; proving a difficult Coq lemma was more the other way around.

Name	**Category**	**Should be in?**	**In library?**
Definition of union	A	yes	yes
Definition of subset (`TARSKI:def 2`)	A	yes	yes
Definition of power set (`bool`)	A	yes	yes
Definition of family of subsets	A	yes	yes
Function exists for any coherent operation (`BINOP_1:sch 3`)	*C*	*yes*	*yes*
Function composed with identity function is the function (`FUNCT_1:38`)	A	yes	yes
Set has zero elements iff empty (`XBOOLE_0:def 1`)	A	yes	yes
Set defined by a function; all elements have preimage	*B*	*yes*	*no*
$A \subseteq B$ and $B \subseteq A$ implies $A = B$ (`XBOOLE_0:def 10`)	*B*	*yes*	*yes*
Subset of a finite set is finite (`FINSET_1:13`)	*C*	*yes*	*yes*
Natural number is greater than or equal to zero	A	yes	yes
Natural number exponentiation definition	*B*	*yes*	*yes*
Natural number raised to zero power is one	A	yes	yes
Real number inequality preserved by addition (`XREAL_1:8,10`)	A	yes	yes
Real number multiplied by zero is zero	A	yes	yes

Table 2.2.1. Lemmas of basic set theory.

Name	Category	Should be in?	In library?
Equipotent definition	A	yes	yes
Equipotent sets have a bijection between them (`WELLORD2:def 4`)	*B*	*yes*	*no**
Identity map is bijection	*B*	*yes*	*yes*
Cardinality definition	*B*	*yes*	*yes*
Cardinality of set and the set are equipotent	*B*	*yes*	*yes*
Cardinality of finite cardinal is the cardinal	*B*	*yes*	*no*
Cardinality is natural number for finite set (`CARD_4:4`)	*B*	*yes*	*yes*
Cardinality of natural number is the natural number	*B*	*yes*	*yes*
Cardinality finite for natural number	*B*	*yes*	*yes*
Cardinality is positive iff nonempty	A	yes	no
Cardinality greater than one implies two distinct elements	C	no	no
Sequence definition	*B*	*yes*	*yes*
Sequence length equivalent to domain (`EULER_1:1`)	*B*	*yes*	*yes*
Sequence length greater than natural number implies in domain (`AFINSQ_1:1`)	*B*	*yes*	*yes*
Sequence implies exists function with domain its cardinality (`RLVECT_1:def 12`)	*B*	*yes*	*yes*
Sequence exists consisting of +/- `Card_Intersection(k)`	*C*	*yes*	*yes*
Finite number of finite sets' union is finite	*C*	*yes*	*yes*
Sum definition	*C*	*yes*	*yes*
Sum of sequence of zeroes is zero	*B*	*yes*	*no**
Sum of constant sequence equals multiplication	A	yes	yes
Sum of sequence consisting of +/- `Card_Intersection(k)` equals cardinality of the union (`CARD_FIN:67`)	*C*	*yes*	*yes*
Pairwise disjoint definition	A	yes	yes
Pairwise disjoint implies cardinality of intersection is zero	*C*	*yes*	*no*

Table 2.2.2. Lemmas relating to cardinality, sums, and pairwise disjointness.

*These theorems were proved in the MML, but in a significantly different format. Massaging the variables on hand and the statement of the existing theorem until they matched was time-consuming enough to be classified as a difficult lemma.

Name	Category	Should be in?	In library?
Left action definition	B	no	no
Left action associativity	B	no	no
Left action identity property	B	no	no
Group operation definition	*C*	*yes*	*yes*
Group operation associative	*B*	*yes*	*yes*
Group operation identity	*B*	*yes*	*yes*
Group inverse property	*B*	*yes*	*yes*
Definition of orbit	B	no	no
Definition of stabilizer	B	no	no
Definition of *H*(x)	B	no	no
Union of H(x)s $\subseteq$ G	B	no	no
G $\subseteq$ union of H(x)s	B	no	no
Stabilizer $\subseteq$ G	A	no	no
H(x)s and orbit are equipotent	B	no	no
H(x)s are pairwise disjoint	C	no	no

Table 2.2.3. Lemmas directly related to the orbit-stabilizer theorem.

Again, the italicized lemmas are the ones that could have been reasonably included in the MML given that a Mizar article devoted to that area of mathematics exists. We see that Mizar indeed has a fairly comprehensive database of theorems available, at least for proving this finite version of the orbit-stabilizer theorem.

A difference from Coq was that some of the most difficult portions of a Mizar proof (which were done for us) were the definitions of new terms, as observed by the complexity of the code in the MML. In Coq, the most difficult portions were lemmas.

Category of lemma	Total number	Number already done	Percentage already done
B	19	15	79%
C	8	7	88%

Table 2.2.4. Estimate of amount of work already completed by the MML.

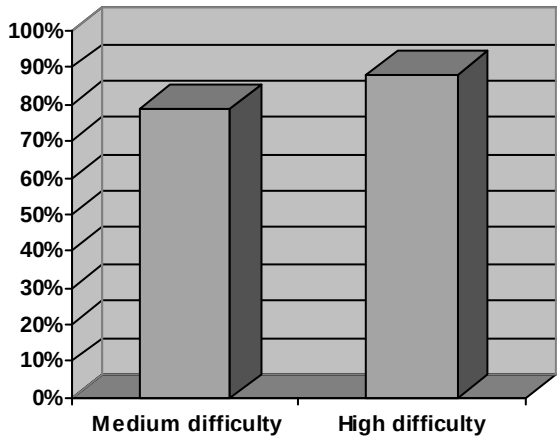

Figure 2.2.1. Percentage of lemmas already completed by the MML.

It is well-known that Mizar (partially due to being one of the first theorem verifiers in existence) has an extensive database of work already compiled in it. We stress, however, that this particular metric, the analysis of the raw number of lemmas already proved for us, does have a Mizar bias. The lack of theorem names or descriptions shifts a little of the work toward *finding* the theorem as opposed to using it, so simply having every theorem one could possibly want is not the end-all. Also, although it sounds odd to say it, even the longest and most difficult of Mizar proofs were not as mentally taxing as learning the syntax itself. The most difficult parts of the entire proof, subjectively, were learning how to use the `scheme` functionality (how to create a function object from a declaration of the computation of that function) and comprehending the statement of the inclusion-exclusion principle. So for a novice Mizar user, at least, perhaps having almost all of the smaller lemmas already proved to him is not his biggest concern.

2.3. Analysis

2.3.1. Issues of the Mizar environment

Dr. Wiedijk suggested that "if you are completely mystified by a Mizar typing error, start thinking 'cluster'!" (1999, p. 16). Indeed, `102: Unknown predicate` and `103: Unknown functor`, two of the most common Mizar compilation errors, are nearly always environment-related. The most effective way to handle these is to use the `findvoc` command-line program to locate the originating file of that term or keyword and then simply add that file's name to various categories of the environment. If that solves the error, great. Otherwise a more painful merge of the environment is necessary, which consists of transplanting the entire environment – all eight lists of MML references – of the originating file, removing duplicates, and seeing if that fixes the

problem. Usually it does, and then it is a good idea to follow up by removing excess unnecessary files that were added. This is because order and inclusion matters in some environment directives, especially `notations` (Wiedijk, 1999, p. 18). I did encounter cases where I had to reorder or remove extraneous references from environment directives to fix errors.

One wonders why Mizar does not adopt the simple package model of modern programming languages such as Java. Perhaps the complicated environment system is needed to keep some consistencies with the underlying theory of Mizar. However, one aspect of the package model that would benefit Mizar is the concept of prefixing package names.

The issue here is that Mizar supports operator overloading. In actual mathematics, one pays little attention to the "types" of variables, but in Mizar, as in most programming languages, types are paramount. For example, the usual notation for power in Mizar is `|^`, and is defined in different files for natural numbers, real numbers, and complex numbers. In mathematics one does not need to worry as much about making sure one's variables are the proper types when using the power operation with them, but in Mizar, of course, one has to have all the types match up for a proof to compile. This can cause subtle difficulties since Mizar does not identify "which" `|^` it is using in a compilation. Is my `Unknown functor` error occurring because I have not imported the power operation properly or is it because the types of my variables are wrong? It would be nice to be able to write `Real.|^` in a Mizar formalization, and perhaps this should even be forced upon Mizar proof writers for their own good.

I encountered this problem when trying to apply `CARD_FIN:67`, a statement of the inclusion-exclusion principle by Karol Pak.

```
theorem Th67:
  for Fy be finite-yielding Function,X st dom Fy=X
    for XFS be XFinSequence of INT st
          dom XFS= card X &
          for n st n in dom XFS holds XFS.n=
          ((-1)|^n)*Card_Intersection(Fy,n+1)
      holds
   Card union rng Fy=Sum XFS
```

Figure 2.3.1.1. The inclusion-exclusion principle in `CARD_FIN.MIZ`.

In originally attacking the problem, I browsed the MML to learn about this new keyword `Sum`, referred to in the last line of the theorem statement. `GR_CY_1.MIZ` (a Mizar article by Dr. Dariusz Surowik about cyclic groups) defines a keyword `Sum` on a structure known as `FinSequence of INT`.

```
definition let F be FinSequence of INT;
 func Sum(F) -> Integer equals
  addint $$ F;
  coherence;
end;
```

Figure 2.3.1.2. Definition of Sum in GR_CY_1.MIZ.

There are many redefinitions of Sum in the MML, so when I saw this one, FinSequence looked close enough to XFinSequence that I did not press on and try to find an exact match, assuming XFinSequence was a subtype of FinSequence (the statement of CARD_FIN:67 indicates that the argument to Sum is a variable of type XFinSequence). The two types are not actually related as far as Mizar is concerned. I spent several hours following this wrong track, delving into the definitions of FinSequence and the $$ operator referred to in GR_CY_1's definition of Sum, before I finally came back and tried to apply my results to the original theorem, which is when I found the typing error. If the author of CARD_FIN had been able to notate CARD_FIN.Sum or XFinSequence.Sum, to indicate that he was using his own redefinition of the keyword, I would have avoided that red herring.

2.3.2. Theorem location systems

Clearly, a grep search is not the best way to go about finding out if a theorem or definition has already been proved for us, in either Mizar or Coq. There has already been research into more sophisticated systems of theorem searching. The Alcor assistant for Mizar uses a latent semantic indexing (LSI) algorithm and has been fairly successful in finding theorems using a kind of fuzzy logic. Alcor's search algorithm returns multiple results based on the structure of the theorem query, instead of a simple yes or no answer for an exact match (Cairns & Gow, 2006, p. 9). This would be helpful in locating theorems or definitions whose name is not universally agreed on. For example, when searching to see if there were a Mizar article treating pairwise disjoint sets, I had to try multiple iterations of text search: "empty" cross-referenced with "intersection," "pairwise-disjoint," and "pairwise" and "disjoint" separately, which eventually found that Mariusz Giero had defined the concept as mutually-disjoint in TAXONOM2.MIZ.

Another example I encountered was when trying to find the indicator functions referenced by Wikipedia in the proof of Markov's inequality. I looked fairly extensively for the definition of this concept and did not find it. Later, after having defined the term myself, I was looking for some other definitions when I happened across a strange Ch keyword, which I realized was characteristic function, the term that MML writer had used to define the concept of indicator functions. Characteristic function is a well-known alternate name for the idea and I should have known to search for it, but for some reason it did not occur to me during the original search.

A final example was SetSequences. In this case, there is not even an agreed-upon name for this concept, that of numbering the collection of partition pieces of a simple function. I did not think the concept would have its own term, and the keyword is_simple_func_in, through which I found most of the material related to simple functions, did not reference it. I again found the

term by accident while searching for something else, after already writing the definition of the concept myself and proving results about it. The idea of proving that a function has a finite number of possible values by actually *instantiating* the finite list of partition pieces themselves is beyond the interest of normal mathematicians. How, as a library writer, am I to indicate to potential users that I have created such a thing, since even adding comments will not help as they will have no clue how to begin searching for it?

Dr. Michael Beeson suggests a solution based on the Mathematica® scientific computing software. Mathematica®'s Help Browser is a menu system for locating theorems (or definitions) with an intuitive tree structure. Most mathematicians will agree what category a particular theorem or definition falls in even if they do not agree on what to name it, or how exactly to word its statement. This would help in the case of `SetSequence` or characteristic functions, because even if the name were unfamiliar to the searching user, Help Browser would present the concept to them incidentally since they are in the right place to see it. The category for `SetSequences` would be simple functions, for example. This kind of system also separates a library writer from the problem of not being able to put her theorem, definition, or lemma in the expected MML file because that file is already finalized or because she was not the author. That does happen in Mizar, and as Dr. Wiedijk says, theorems "can be in unexpected places" as a result (Wiedijk, 2006b, 33).

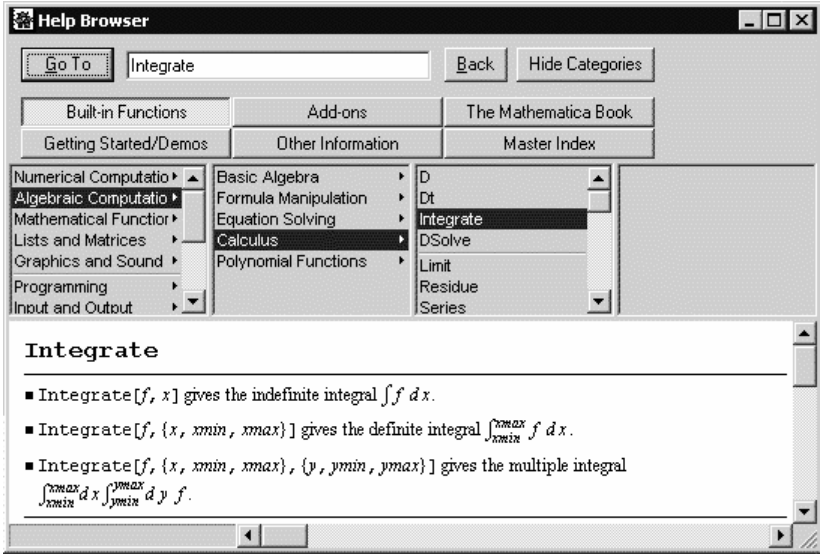

Figure 2.3.2.1. Mathematica® Help Browser.

2.3.3. Mizar file documentation

Most Mizar files restrict their comments to little more than a one line description of the contents of the file. For example, `XREAL_0.MIZ`, an invaluable file for any proof involving real numbers, has the title `Introduction to Arithmetic of Real Numbers` at the top, and a single comment labeling where in the file begin the definitions of `min & max`.

Assuming the lack of a more intelligent system for locating a particular theorem or definition, a short title for each theorem and definition would go a long way to both helping a new user get up to speed on the contents of a file and facilitating text searches. The "central" theorem of a file, or very well known ones, actually do often have a heading of this sort. For example, Karol Pak labeled his statement of the inclusion-exclusion principle:

```
:: The principle of inclusions and the disconnections

theorem Th67:
  for Fy be finite-yielding Function,X st dom Fy=X
    for XFS be XFinSequence of INT st
          dom XFS= card X &
          for n st n in dom XFS holds XFS.n=
          ((-1)|^n)*Card_Intersection(Fy,n+1)
      holds
   Card union rng Fy=Sum XFS
```

Figure 2.3.3.1. The inclusion-exclusion principle.

Even a more readable statement like `A c= B & B is finite implies A is finite` (`FINSET_1:13`) could benefit from having the description `The subset of a finite set is finite` above it.

For complex statements like the inclusion-exclusion principle, once located, the task becomes then to understand what each part of the statement is referring to. A significant part of the "familiarizing with new terms and theorems" portion of the work in Table 2.1.3.1 was decoding the statement of this theorem and learning what each piece meant. The Mizar statement of a theorem is usually more detailed than the colloquial version, and yet each variable usually has a one or two letter name, due to the need to keep expressions fitting on one or two lines. I spent several hours drawing pictures to represent all the layers of bijections between cardinalities and subsets, before I could begin writing any Mizar code involving this theorem.

In fact, there were several instances where I guessed that a certain lemma had already been proved in the MML, but because proving it myself would be fairly straightforward, I preferred to reinvent the wheel instead of assembling text searches and comprehending lemma statements.

3. Formalizing Markov's inequality in Coq

The other major theorem I formalized in Coq and Mizar was Markov's inequality, which provides a loose bound on the probability of a random variable taking on values greater than or

equal to some fixed constant. If For a nonnegative random variable *f* and a real number *a*, Markov's inequality states that

$$\text{the probability that } f \geq a \quad \leq \quad \frac{\text{the expected value of } f}{a} .$$

We do the same analysis of categories for this formalization.

Category of work	Approximate work (hrs.)	Percentage of total work
Learning Coq syntax	8	9%
Experimenting with Coq tactics	10	11%
Browsing the user-supplied library	12	13%
Familiarizing with new structures, terms of library	6	7%
Actual formalizing	43	48%
Planning out actual proof strategy	11	12%
Housekeeping (installing and setting up Coq)	<1	negligible

Table 3.1. Breakdown of Coq formalization time.

The major difference from the proof of the orbit-stabilizer theorem is the cutting down of time spent on learning the new library I needed for this proof, the Coq standard library. Although I was using it for the first time, I used only a small portion, mainly the axioms of and results about real numbers, and the `Ensembles` section. I used the standard library more often as a model for learning how to write definitions, or ideas for how to work with functions and sequences, than a place from which to take definitions of needed terms. This is because a large part of the proof of Markov's inequality *was* writing the definitions of the basic terms of Lebesgue integrals like sigma-algebras and simple functions, as the Coq standard library does not include these.

3.1. Coq file documentation

Coq libraries mostly try to label their theorems with descriptive names, and the authors of the C-CoRN, Pottier-Stein, and Coq standard libraries included some helpful comments as well. Overall, Coq user libraries seem to have a little more documentation in-file than Mizar MML contributions. This is especially useful for concepts with canonical names. For example, what is a good `grep` search to find a Mizar lemma stating "for reals x,y,z, x≤y and y≤z implies x≤z," knowing that variable names and whitespace may be different, and the Mizar author may have

used the `holds` keyword instead of `implies`? In Coq this theorem can be located instantly by searching for "trans" and picking out the one related to reals, `Rle_trans`.

For concepts without canonical names, the descriptive theorem titles still help. What I think of as injective the Pottier-Stein library terms `distinct`, but when searching the Pottier-Stein library, all theorems relating to injectivity have `distinct` somewhere in the name.

3.2. Library incompatibility

The reason we moved to the Coq standard library for Markov's inequality, abandoning the previous two libraries we learned, was a current problem with the Coq contribution repository. Even though the C-CoRN library I knew had deep support for real numbers, and the Pottier-Stein library had definitions and support for the basic concepts I would use to build measure theory and Lebesgue integrals, including sequences, functions, and predicate treatment of subsets, I ultimately could not use any portion of either in my work.

To do measure theory, I needed a formulation of the real numbers (usually measure theory is defined using the extended reals; however, since we were considering Markov's inequality as it applied to probability spaces, the normal reals would do). Pottier-Stein does not treat reals at all. On the other hand, C-CoRN lacks the notion of predicate subsets, which allows one to define countably infinite collections of sets, a concept needed to define sigma-algebras, in a natural way. (In a predicate subset foundation, a subset is implicitly defined by a predicate that acts like a filter; the sets that pass through the filter, from the original set, comprise the intended subset.) I could not see a clean way to do this with C-CoRN's `CSetoid`-based class hierarchy. In other words, both libraries lacked one essential piece. The Coq standard library has good support for predicate subsets in its `Ensembles` files as well as a treatment of reals. I also noticed that it has the Riemann integral defined, which would be useful as a model for how to define the Lebesgue integral later. But this would still mean learning an entirely new library.

The unfortunate part was that I could not simply pull out the predicate subset portion of Pottier-Stein and the real number portion of C-CoRN. I needed to have *one* library which had support for both concepts, because of the incompatibility of each library's foundations. As mentioned before, each of these three libraries build their foundations for set theory (setoids and how to create subsets) from the ground up: `Setoid` versus `CSetoid` versus `Ensemble`. Trying to combine theorems from different ones would be like trying to write a program in C and asking if I could pick a few of my favorite functions from a Java package, and a few from a Visual Basic library. It just does not work; I have to start in C with what support there is for what I need to do, and write the whole program based on that.

3.3. Finding out which axioms we've used

One of the main strengths of Coq, and one of its important differences from Mizar, is its abstract foundation. Mizar has a broad scope in starting from ZF set theory, but Coq goes one level more abstract by beginning with only the most fundamental rules of first-order logic, set theory being just one application of this foundation. An advantage of this is being able to do things like either assume the axiom of choice or not. One simply chooses to invoke the axiom of choice in one's proofs or not, through a variety of formulations like `choice`, `constructive_choice`, and `constructive_definite_description`. The only issue is that it is not easy to figure out when looking at others' results and theorems, whether or not *they* used the axiom of choice. To know for sure, one would have to recursively backtrace every lemma or theorem used in a particular proof, searching for any use of one of the axiom of choice's many formulations.

This problem, from the standpoint of the design of Coq, is an easy one to fix. Coq needs a utility that will step through an entire proof and run tree recursion on theorem calls all the way down to component axioms, and see if any of them are a form of the axiom of choice. This process could even assemble the complete list of axioms the theorem depends on. This process may actually tie into Coq's normal compilation algorithm in a natural way.

3.4. Coq is more difficult to read

One of the realizations that came to me after the formalization of Markov's inequality in Coq was the striking difference between Coq expressions and Mizar syntax: how differently expressions written in these two systems "read."

The Calculus of Inductive Constructions' idea of proofs as types has something in common with the programming language Lisp: rapid growth of nested parenthetical expressions, though the mechanism is different. Let me give an example.

Consider a sigma-algebra `F`, a "nice" family of subsets of the universe set `X`. Now, in Coq, if we want to say that a set `t` is an element of `F`, `Element of F` in Mizar parlance, we do this by simply supplying a little proof that `t` is in `F`: probably trivial or assumed as a hypothesis, but necessary to have explicitly named. Suddenly the set `t` has become a pair `(t,t')` where `t'` is the proof that `t` is indeed in `F`, and now that pair can be thought of as the set `t`, but tagged as type `Element of F`. Now consider a subset of the sigma-algebra `F` called `T`. (This is a common setup in many of the lemmas I wrote during Markov's inequality.) So `T` is also a family of subsets of the universe `X`; in fact, `T` is a subset of our sigma-algebra on `X` known as `F`. Now consider our set `t` from `F` again; only this time, we even know that `t` is in `T`, not just in `F`. So once again, the pair `(t,t')` where `t'` is now the proof that `t` is in `T`, can be thought of as a variable whose value is `t`, but has type `Element of T`.

Now, suppose I am moving along in a proof, proving various things about `t`, and I come to a point where I want to use some theorem that applies to arguments of type `Element of F`; that is, this theorem is generally meant to be used on elements of the sigma-algebra. Well, in Mizar thinking, or object-oriented programming language thinking if the reader prefers, since `F` is a "parent" of `T`, then obviously if `t` is in `T` then `t` is in `F`! Should we even have to tell Coq this? Well, in Mizar or most OOP languages one does not, but one must in Coq. Moreover, the proof that `t` is in `T`, which we *do* have at the moment, has no obvious way to be "converted" into a proof that `t` is in `F`, at least not a nice looking way.

The only way is to create a "proof converter theorem" that takes a proof that something is in `T`, and outputs a proof that that thing is in `F`. In the end, the exact Coq syntax one has is `(T' t t')`, where `T'` is the proof converter, and this triplet is the proof that `t` is in `F`. Then, to get the actual item `t` "cast" to type `Element of F`, we have `(t,(T' t t'))`. That is, we have the original set `t`, accompanied by a proof that `t` is in `F`. Any element of `T` cast to type `Element of F` will look similar: `(s,(T' s s'))` or `(v,(T' v v'))`.

This is the most direct and proper way to do a type cast in Coq. It is a consequence of the highly structured and exact nature of the Coq proof verifier, and is probably the single biggest culprit in making Coq expressions (and thus, the interactive Coq proof process) difficult to read and follow for humans.

3.4.1. A more detailed example

For another, more detailed example of this phenomenon, let us look at what happened later in Markov's inequality. One step in the proof required me to show that for a constant *c*,

c×integral of indicator function = integral of (*c*×indicator function).

My statement of this fact in Coq:

```
forall(c:R)(c':0<c),lebint_s _ _ _ _ _ u u' u'' u''' n(fun i:
nat=>v i*c)w(simple_P9 _ n v w S0 S1 S2 S3 c c')(simple_P10 _ n
v w S0 S1 S2 S3 c c')(simple_P11 _ n v w S0 S1 S2 S3 c c')(
simple_P12 _ n v w S0 S1 S2 S3 c c')(simple_P13 _ _ F' F'' F'''
n v w S0 S1 S2 S3 M c c') E E' =
c*lebint_s _ _ _ _ _ u u' u'' u''' n v w S0 S1 S2 S3 M E E')%R.
```

Figure 3.4.1.1. Formulation of a property of Lebesgue integral.

The bold portion of the lemma is somewhat decipherable as the asserted fact. `lebint_s` is the Lebesgue integral for a simple function. `v` is the finite sequence representing the list of possible range values, and `w` is the associated list of partition pieces. The lengthy expressions `simple_P9` through `simple_P13` are what I did not realize I needed until actually sitting down to write this statement. They are proof converter lemmas, saying that multiplying a simple measurable

function by a constant still preserves the four qualifications for a function to be simple and the one description that says a function is measurable. In other words, these long expressions popped in unexpectedly – when I thought I would be writing simply a statement about a constant and a Lebesgue integral, I failed to see that I would need to include lengthy expressions related to proper typing of arguments.

Imagine what would happen if writing out *those* lemmas necessitated filling in the blanks of more expressions as we broke each hypothesis down to its component basic assumptions, and having to verify each one in turn. This cascade of expressions is, again, the mechanism by which Coq expressions quickly become very large. It happens to a greater degree when one enters the interactive proof of a theorem and begins to unfold the various definitions in the statement to prove and use facts about them. Anyone who uses Coq for a nontrivial application runs into unmanageably large expressions. Figuring out how to do anything with these is one of the most difficult hurdles to overcome in learning how to use Coq. (One way to deal with this issue is by *going around* these large expressions, stepping out of the current situation and defining helper lemmas that simplify pieces of the expression until it is whittled down to a manageable size, at which point one can proceed with the original proof.)

Mizar, like a structured programming language, can hold information about current variables and types invisibly in the current environment or scope and avoids generating large expressions that in a sense try to include the whole environment within themselves. However, this is because Mizar does not seek to support the more powerful and fundamental idea of proofs as types in the Calculus of Inductive Constructions.

3.4.2. Implicit arguments

Coq actually does have an attempt at a solution for these large expressions, implicit arguments. After interacting with it, I feel that this is not the final solution, as it creates its own issues. For an example, here is the syntax of the `ifdec_right` theorem.

```
Theorem ifdec_right :
  forall (A B:Prop) (C:Set) (H:{A} + {B}),
    ~ A -> forall x y:C, ifdec H x y = y.
```

Figure 3.4.2.1. The `ifdec_right` theorem from the Coq standard library.

The idea is that it simplifies one level of an `if-then` expression. Even to a seasoned Coq user, the statement and arguments are difficult to interpret, and I needed to do some testing to figure out exactly what to pass as arguments. However, the use of implicit arguments in this theorem (specified by an earlier directive) makes it extremely hard to use the normal procedure of deducing argument types by guessing and checking. In Coq, learning how to use a theorem usually means invoking it and supplying one's best guess for the arguments, which Coq responds to with typing errors, which one reads to get hints on what to try next. Depending on the

theorem, this might take one round of guesses or dozens. This process, while tedious, is at least predictable and methodical, and works.

It does not work as well when the theorem uses implicit arguments. The implicit arguments functionality will fill in certain arguments for the user based on the arguments the user supplies, but Coq's algorithm for deciding which arguments to fill in and when is hard to figure out. It is often not clear what place a particular argument one supplies is going in when implicit arguments are involved. The effect is to multiply the number of combinations of arguments one has to try as well as make the returned error messages harder to take hints from. I remember trying about forty iterations of different attempts at argument lists to the function.

The point of implicit arguments is to let a seasoned user of the theorem input a few less arguments, thus shortening the expression and saving space and typing time. However, without author-written documentation on how to use the theorem, the added frustration for new users is unacceptable. In the end, I actually gave up and wrote a version of `ifdec_right` without the implicit arguments specifier, adapting their proof of `ifdec_right` to work with the change in convention.

4. Formalizing Markov's inequality in Mizar

We use the same Mizar categories as for the orbit-stabilizer theorem, and see if the proportions of time change as we expect.

4.1. Breakdown

Category of work	Approximate work (hrs.)	Percentage of total work
Learning Mizar syntax by trial and error	7	8%
Searching libraries for examples of new syntax	2	2%
Familiarizing with user-defined libraries	7	8%
Searching for existence of terms and theorems	8	10%
Rote formalizing	48	56%
Fixing the Mizar environment	4	5%
Logically planning out the proof itself	10	11%

Table 4.1.1. Time breakdown of Markov's inequality in Mizar.

As expected, the proportion of time for rote work has increased significantly from the previous formalization of the orbit-stabilizer theorem. This makes sense given my increased experience with many aspects of Mizar, including creating skeletons for inductive proofs and proofs by cases, as well as using Mizar "schemes" and defining functors with `deffunc`. By a certain point in this formalization, I had solved all those problems several times, so even if I needed to look back at some old examples, or do a little trial and error to refresh my memory, I was confident that I could write the portions of code involving these pieces of syntax without significant trouble. Thus, I file much of the time associated therewith under rote formalization for Markov's inequality, while they were partially or completely new to me during the orbit-stabilizer theorem.

A category that ended up taking the same proportion of time as it did with the orbit-stabilizer theorem was the searching for the existence of terms or theorems in the MML. Again, we note that this category reflects only searching for the term or theorem itself. For example, we might be asking, "Okay, is there a definition for Lebesgue integral?" Or, "Is there a theorem that tells us that every set in a sigma-algebra is necessarily a subset of the universe set *X*?" Looking up these answers is purely the action of doing `findstr` or `grep` text searches, employing `findvoc`, or browsing MML files. To be pedantic, once I went into the realm of trying to *use* or get familiar with the new MML term or theorem, I felt that I had gone into a different category, related to the ease or difficulty of getting up to speed with some new Mizar entity *once I had already found it*. This division allows us to separate two issues with the Mizar system: that of a user-friendly search capability and that of the ease or difficulty with which one can interpret and make use of others' Mizar creations.

It makes sense that the proportion of time should be roughly the same, as my skill with searching out things in the MML plateaued a while ago. This is a skill that one learns early on in one's Mizar career. There is actually a slight drop in the proportion of time; this is either experimental error or a result of being familiar with a slightly larger section of the MML after completing the orbit-stabilizer theorem. That theorem gave me a lot of experience with the MML files about finite sequences and real numbers, which I used often in Markov's inequality.

Looking at the other categories, the results show that the time spent learning Mizar has approximately been cut in half. The two main types of work related to learning Mizar, all of which can be viewed as a one-time cost, are the time spent figuring out how to use new Mizar syntax, which in this formalization mainly consisted of writing term and functor definitions, and the time spent reading and getting familiar with user-defined MML terms and theorems. We notice that the time spent on these categories dropped from 16% to 8% and from 18% to 8%. This is good news, as these percentages are becoming small enough that if one could find a way to partially automate the rote work portions of a formalization, it would have a large effect on the total time required.

For the total time, although the orbit-stabilizer theorem and Markov's inequality turned out to be very different styles of proofs – the orbit-stabilizer theorem was almost pure logical reasoning, while Markov's inequality contained a significant amount of work related to simply defining new terms – the total time was about the same. Markov's inequality tallied 86 hours while the orbit-stabilizer theorem tallied 89 hours.

At this point, I expect that further exercises with Mizar will have diminishing returns toward improving my proficiency with Mizar. Although there are a few terms and structures of Mizar syntax that I have not yet tried, such as defining records or schemes, I am certain that I can deduce the proper syntax for those with the same trial and error method that has served me well. As for the proofs themselves, I have pretty much established the basic style of approach. First, I try to understand the proof conceptually, writing out a proof in normal mathematical notation. Then I peruse the MML to see what previous Mizar users have proved and defined already for me, browse the associated lists of theorems to get a feel for what facts I already have, and then write a skeleton of the entire proof in Mizar to hash out all the environments, definitions, and typing errors. Once the skeleton is complete, one can then simply pick out lemmas or parts to begin the rote formalizing work, and slowly fill in the proof, gap by gap, until the blessed moment when one runs `mizf` for the last time and no `*4` errors appear in the checker, meaning that as far as the checker is concerned, one's proof is completely valid.

4.2. Markov in Mizar analysis: syntax and interface

We cover a few issues and potential areas of improvement discovered during this formalization of Markov's inequality in Mizar. We begin with syntax- and interface-related issues.

4.2.1. Piecewise functions must be named

I am not sure if this is inherent in set theory, but in Mizar, defining a piecewise function is a tricky process. The best way is to use the `func` keyword to create a functor (also known as a metalanguage function, as it exists outside of the set theory universe in which Mizar formalizations live) and then use the `if-otherwise` construction for the separation of cases. (One cannot use the `if-otherwise` construction outside of a `func` definition.) Then one must convert this functor into a `Function` in the Mizar set theory universe by use of `FUNCT_1:sch 3`. I used this process to define indicator functions as well as two auxiliary piecewise functions that might as well have been nameless, being only intermediaries to details of the proof. My slight quibble with this is having to take up keyword space to name these minor functions.

4.2.2. Mizar theorem numbering

Mizar requires all theorems to be numbered. However, authors of MML files often must revise their submissions as time passes, and insert or delete theorems.

When deleting a theorem, Mizar allows one to replace it with a `canceled` statement. That single keyword takes the place of the theorem, and aside from having to remove all references to that theorem by other files, one needs do nothing else as the placeholder `canceled` preserves the original numbering of all other theorems. However, it would probably be cleaner to, instead of using the `canceled` keyword, simply keep the theorem around and change its statement to something meaningless like `0=0` or `not contradiction`, so that one can do text searches and simply count the number of times the word `theorem` appears to get to a specific theorem number.

There is currently no good way to insert a theorem into a finished MML file since Mizar theorem numbers are integers; no Dewey Decimal System here. Again, this contributes to newer theorems cropping up in odd places. One is used to going to `FINSEQ_1` for all one's finite sequence needs, and then finds that a useful singleton sequence theorem is located in `CONVFUN1`.

I see no problem with the idea of numbering theorems in general, as it cuts down on typing long theorem names repeatedly, as is sometimes needed in Coq. Comments could take the place of the descriptive theorem names. To alleviate the theorem insertion problem, MML writers could start numbering theorems by tens, as BASIC programmers of old were taught to do.

4.2.3. `findvoc` does not cover redefinitions

The `findvoc` utility is good at finding the original definition of a term, quicker than `grep` or `findstr`. However, it cannot find redefinitions, and using `grep` or `findstr` in this way requires fairly sophisticated regular expressions. Again, Mizar would benefit here from a feature of modern programming environments where an intelligent piece of software compiles a list of all definitions and redefinitions of a particular term and displays them when the user right-clicks the term. One model is Microsoft Visual Studio's dynamic popup interface.

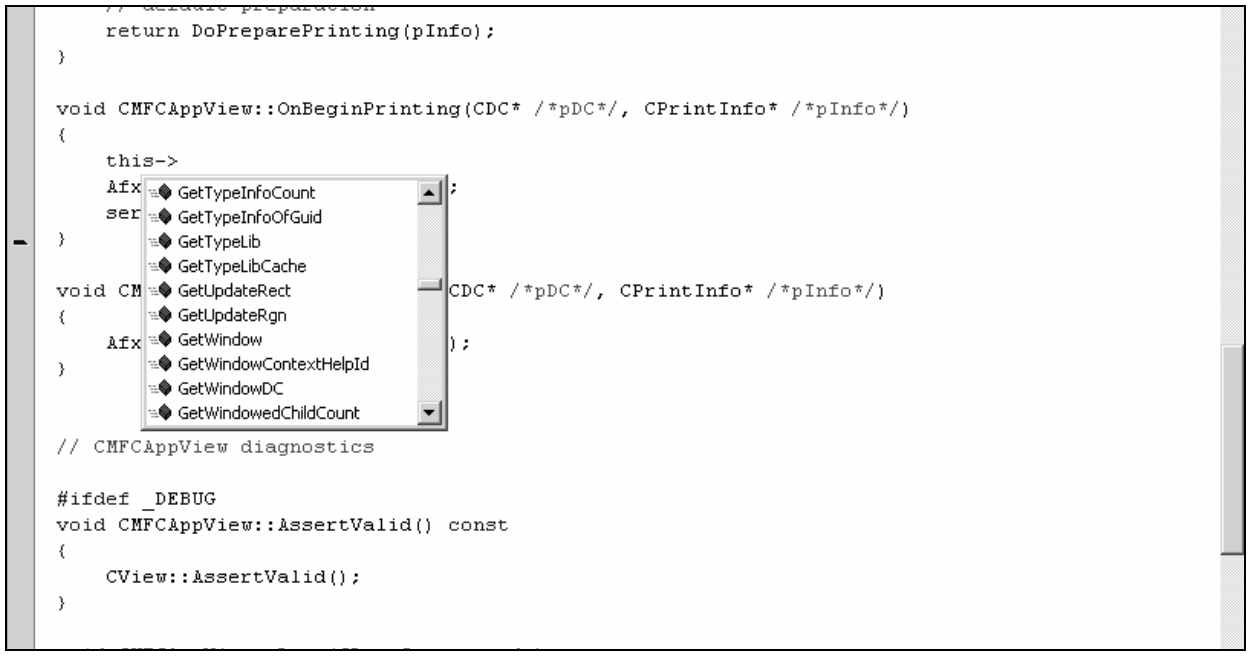

Figure 4.2.3.1. Microsoft Visual Studio dynamic list of all member functions.

Even better would be if that piece of software determined *which* particular redefinition of the term is being used in the context. Indeed, it is often not clear to the Mizar user which version of a term Mizar is interpreting a particular usage as. The most common example, mentioned earlier, is `Sum()`, which has been defined on many different types over the years. Another example is the innocuous <=, which while only being defined on two major types, reals and extended reals, due to the nature of reals being a subset of extended reals causes many typing errors.

4.2.4. Order of hypotheses

One odd discovery was that combining two logical statements with the "`&`" operator is not necessarily symmetric when expanding a definition. This was surprising because Mizar is usually robust in this regard, for example, when combining multiple statements to draw a conclusion. Let us look at an example involving the `union` term.

Here is the definition of `union` in `TARSKI.MIZ` (`Def4`):

```
definition let X;
  func union X means
   x in it iff ex Y st x in Y & Y in X;
 correctness;
end;
```

Figure 4.2.4.1. Definition of `union` in `TARSKI.MIZ`.

Compare the following four usages of the term, only one of which generates an error when compiled. These are direct applications (either unfoldings or collapsings of the exact definition). (The reader can verify these results using a bare-bones environment with just `TARSKI` in `vocabularies`, `constructors`, `notations`, and `theorems`.)

```
for x,Z be set st x in union Z holds
ex Y be set st x in Y & Y in Z by TARSKI:def 4;

for x,Z be set st x in union Z holds
ex Y be set st Y in Z & x in Y by TARSKI:def 4;
::>                *4

for x,Y,Z be set st
x in Y & Y in Z holds x in union Z by TARSKI:def 4;

for x,Y,Z be set st
Y in Z & x in Y holds x in union Z by TARSKI:def 4;
```

Figure 4.2.4.1. Reversing the order of hypotheses causes an error with `union`.

Notice that the order of the two logical parts of the definition of union – that for `x` to be in the union of `Z`, there must exist an intermediate set `Y` in `Z` and `Y` must contain `x` – matters when we are unfolding the definition, but not when we are collapsing it. This would be a good candidate for inclusion in a FAQ about Mizar for new users, if there were such a document. (Coq has one and was a useful reference at all stages of my learning it.)

4.2.5. Order of `notations`

One troublesome aspect of the Mizar environment lists is that not only must one make sure that one has all the needed files in the lists, but one must occasionally play around with the order of the files. The phenomenon seems to be related to redefinitions. In one occurrence of this issue, a file I was adding to the `notations` list had to be placed before another specific file to prevent typing errors. This does not bode well for the MML persisting as a single database of all mathematics. With enough layering of redefinitions and cross-referencing of MML files, two files from the MML might become completely incompatible, unable to both be used for a given proof.

Here are the essential portions taken from my larger Markov's inequality file, with which the reader can reproduce the error. The statement about `X`, `f`, and `x` at the end is not true; it is merely there to showcase the error. The error is that, if `FUNCT_1` occurs after `SUPINF_2`, one gets a `*102` typing error, which is probably because Mizar is interpreting the expression `f.x` as returning the default type `set`, which does not make sense with the `<=` (less than or equal to) operator. If `FUNCT_1` is placed before `SUPINF_2`, Mizar returns a `*4` logical error, correctly as the test statement is not true, and the fact that Mizar got to this stage of error shows that Mizar accepts the statement as well-typed. In other words, the typing processed properly, and Mizar knows

that `f.x` returns something of type `ExtREAL` (extended real) for which the `<=` operator makes sense.

```
environ
vocabularies PARTFUN1,SUPINF_1,FUNCT_1;
notations XXREAL_0,PARTFUN1,SUPINF_2,FUNCT_1;
constructors MESFUNC3,MEASURE6;
registrations SUPINF_1,RELSET_1,NAT_1;
requirements NUMERALS,SUBSET;
begin

for X be set,f be PartFunc of X,ExtREAL,x be set
holds 0<=f.x;
::>     *102
::>
::> 102: Unknown predicate
```

Figure 4.2.5.1. Consequences of the order of the files in the `notations` directive.

I believe this error is related to redefinitions because `SUPINF_2` redefines the `.` operator (function notation) to work intelligently on functions that output extended real numbers. This redefinition lets Mizar know automatically that such functions will output objects of type `ExtREAL` (without the redefinition, the Mizar user would have to do a little proof each time she uses the function to show that the output is, indeed, of type `ExtREAL`). Hence, placing `FUNCT_1` after `SUPINF_2` in notations overrides `SUPINF_2`'s redefinition, and so Mizar goes back to the original `FUNCT_1` definition of `.` where `f.x` is of type `set`, which makes no sense with `<=` (thus the typing error).

```
  func f.x -> set means
:Def4: [x,it] in f if x in dom f otherwise it = {};
  existence by RELAT_1:def 4;
  uniqueness by Def1;
  consistency;
end;
```

Figure 4.2.5.2. The definition of the `.` operator in `FUNCT_1.MIZ`.

```
definition
   let X be non empty set;
   let Y be non empty Subset of ExtREAL;
   let F be Function of X,Y;
   let x be Element of X;
   redefine func F.x -> R_eal;
coherence
proof
   F.x in ExtREAL by TARSKI:def 3;
   hence thesis;
end;
end;
```

Figure 4.2.5.3. The redefinition of the `.` operator in `SUPINF_2.MIZ`.

4.2.6. `reconsiders` do permanent damage

I recall this as one of the most difficult idiosyncrasies of Mizar to uncover overall. The initial error was the common typing-related error `*103`.

At least in the environment I was working in, if one `reconsiders` a variable from type `non empty set`, to `Element of S`, then back to `non empty set`, the variable has been permanently changed. Where previously it was valid in a particular expression, after the change-to-and-back, it generated a `*103` in that same expression.

Here is the actual occurrence in my work, pared down to only what is necessary for the reader to duplicate the error. This code also needs a `MARKOV.VOC` vocabulary file containing the single line of text `Ofsepseq` to be placed in the `\dict` directory, for the definition of the `fsepseq` term. The different situations have been bolded: before the `reconsider` to and back from a different type, there is no typing error, only a logical error. Thus, the statement is parsing correctly with the appropriate types. But after the `reconsider` to and back, we get a typing error, meaning something has irrevocably changed in the variable `X`. The first part of the code is the environment, the second is the definition of the custom term `fsepseq` which seems to have a property that makes this strange behavior possible, and the third is the test expression and illustration of the different situations before and after the `reconsiders`.

```
environ
vocabularies MARKOV,PARTFUN1,SUPINF_1,MEASURE1,RELAT_1,FUNCT_1,
 ORDINAL2,PROB_1,MESFUNC2,BOOLE,INTEGRA1,RLVECT_1,MEASURE6,ARYTM_3,
 TARSKI,COMPLEX1,ABSVALUE,FINSEQ_1,MESFUNC1;
notations FUNCT_1,XBOOLE_0,XXREAL_0,ORDINAL2,RELAT_1,PARTFUN1,PROB_1,
 SUPINF_2,MEASURE1,REAL_1,SUBSET_1,FUNCT_2,MESFUNC2,TARSKI,MESFUNC3,
 SUPINF_1,EXTREAL1,MEASURE6,RELSET_1,NAT_1,COMPLEX1,FINSEQ_1,MESFUNC1;
constructors MESFUNC1,MESFUNC3,MESFUNC2,EXTREAL1,MEASURE6,REAL_1;
registrations SUPINF_1,RELSET_1,NAT_1,SUBSET_1,NUMBERS,XREAL_0;
requirements NUMERALS,SUBSET,BOOLE,REAL,ARITHM;
theorems PROB_1;

begin
definition
let X be non empty set;
let S be SigmaField of X;
let M be sigma_Measure of S;
let f be PartFunc of X,ExtREAL such that
 for x be set st x in dom f holds 0<=f.x;
let a be Real such that a>0;
let n be set;
func fsepseq(X,S,M,f,a,n) -> set equals :Def4:
 {x where x is Element of X:f.x>=a} if n=1
 otherwise {x where x is Element of X:f.x<a};
coherence;
consistency;
end;

for X be non empty set,
S be SigmaField of X,
M be sigma_Measure of S,
f be PartFunc of X,ExtREAL,
a be Real
st (for x be set st x in dom f holds 0<=f.x) & a>0 holds 0=0 proof
let X be non empty set;
let S be SigmaField of X;
let M be sigma_Measure of S;
let f be PartFunc of X,ExtREAL;
let a be Real;
assume A0:(for x be set st x in dom f holds 0<=f.x)&a>0;
fsepseq(X,S,M,f,a,0)=0;
::>                  *4
reconsider X as Element of S by PROB_1:43;
reconsider X as non empty set;
fsepseq(X,S,M,f,a,0)=0;
::>   *103
0=0;hence thesis;
end;
::>
::> 4: This inference is not accepted
::> 103: Unknown functor
```

Figure 4.2.6.1. A `reconsider` followed by its reverse causes a permanent change in a variable.

This is another example of an error that I resolved through trial and error, but for which I could not figure out the underlying reason. It may have something to do with the fact that the original

type, `non empty set`, is a two-part type: it contains a main type, `set`, and an attribute, `non empty`. Perhaps some attribute information is lost through the two `reconsiders`.

4.3. Markov in Mizar analysis: matters of style

4.3.1. Syntactic sugar

The MML seems to define many new types and predicates just to shorten Mizar phrases or make Mizar statements more closely resemble English mathematical statements. In some cases, user-defined types and predicates do clarify code. `are_Re-presentation_of` and `is_simple_func_in` have a clearer purpose than an ordered list of subsets of the universe set `X` and an ordered list of extended-real values.

But what about `Function of A,B` and `PartFunc of A,B`? Why not stick with the already existing terms `dom` and `rng` and say `f is Function & dom f = A & rng f c= B`, if that is precisely what `Function of A,B` means? That is completely clear and needs no new looking up of definitions and flipping back and forth between files.

One reason for the proliferation of terms may be that Mizar has trouble linking multiple logical steps together in a single justification. Mizar's automatic reasoning can occasionally surprise, but overall I have learned to break arguments down into the smallest possible steps. Let us look more closely at what Mizar can and cannot do.

```
consider x,X,Y,Z be set;
A1:x in X;
::>     *4
A2:X c= Y;
::>     *4
A3:Y c= Z;
::>     *4

B1:X c= Z by A2,A3;
::>     *4
B2:X c= Z by A2,A3,XBOOLE_1:1;
B3:x in Y by A1,A2;
B4:x in Z by A1,A2,A3,XBOOLE_1:1;
::>     *4
```

Figure 4.3.1.1. Mizar's automatic reasoning capability.

We examine the Mizar subset operator, `c=`, and what Mizar implicitly knows about it and how well it reasons with it. We have three hypotheses, `A1`, `A2`, and `A3`, and try a few test statements,

B1, B2, B3, and B4. The failure of the first statement, B1, may be surprising to a new Mizar user. Transitivity of the subset operation is not implicitly known by Mizar; it is proved in the theorem XBOOLE_1:1. So B2 does not generate a logical *4 error, while B1 does. B3 is an example of the kind of two-step reasoning that Mizar *can* do. Beyond a trivial logical step like replacing a term with its exact definition (a common single step in Mizar), here Mizar actually combines two separate facts in a creative way. Or does it? If one looks at the definition of c= in TARSKI.MIZ, one sees that Mizar is simply directly applying the definition of c= here, nothing creative.

```
definition let X,Y;
  pred X c= Y means
   x in X implies x in Y;
  reflexivity;
 end;
```

Figure 4.3.1.2. The definition of c= in TARSKI.MIZ.

B4 is a true attempt to get Mizar to make a two-step logical jump. We know that XBOOLE_1:1 combined with the facts A2 and A3 is enough to justify that X c= Z in this example. Adding A1 as another hypothesis produces the exact same scenario of directly applying the definition of c= that Mizar knows without needing to cite any special theorem in B3. However, trying to do both of these steps at once fails: B4 gets a *4 error.

Seeing the level of detail Mizar needs explains why Mizar proofs are so long. This may be the true reason for having a lot of terms. The terms are not to make Mizar look like English mathematical writing; they are for unifying the many small hypotheses and variables of a mathematical concept under a single term such as “simple function” so that we may then write *clusters* about simple functions. Clusters cannot operate on groups of many small hypotheses; they are one-to-one relationships. If s is a simple function, then this cluster says that it has some_property. If s is a simple function, then this other cluster says that s has some_other_property. But are clusters really worth it?

4.3.2. Clusters considered harmful

Clusters are the only way to expand Mizar’s ability to reason *implicitly*. They do so in two ways: they use a hierarchical structure similar to inheritance in object-oriented programming, giving the illusion that Mizar knows more about transitivity that it really does (we saw an instance of its limits in the above example), and they outright tell Mizar rules for making logical jumps.

For the hierarchical structure, consider that a FinSequence (finite sequence) is a Function, and a Function is a Relation. When we start applying theorems that ostensibly deal with Relations directly to FinSequences, it feels like Mizar is employing logic on multiple fronts at once, reasoning about transitivity (knowing that FinSequence is a Relation) while at the same time applying a theorem. This is an example of how clusters allow Mizar to seem to elide many steps into one step. But really, Mizar is doing nothing special. That theorem about Relations

only applies to my FinSequence because I was asking something about dom and rng, basic properties of Relations, and FinSequence, if that term had never been defined, would just be another conglomerate of a dom and rng and some pairs – a Relation is exactly what it is. Mizar needed no logic to do this; it only seems like Mizar is reasoning because we have labeled identical things with different names.

Where clusters really *do* elide multiple steps in one is when the third type of cluster is used (Wiedijk, 1999, p. 10). This kind of cluster is essentially a theorem that has been stamped with a note to Mizar to try to use this theorem automatically whenever it sees the type on the left-hand side. For example, the cluster sigma_Field(C) -> sigma-additive compl-closed non empty from MEASURE4.MIZ allows one to state that one's sigma-algebra is non empty in one's code without any justification. Now we see why clusters operate only on specific types, and hence the ultimate reason for types existing at all. They are to give Mizar an easier time with knowing when to try to apply which clusters. If we did not have clear, succinct labels of Function and FinSequence, Mizar would have to try to apply hundreds of clusters on every statement, and it would run very slowly (Wiedijk, 1999, p. 10).

So clusters require the existence of types. Types, as a side effect, do eliminate a significant amount of writing in any proof. If types like Function were not defined, and we kept long assumption lists such as f is Function & dom f = X & rng f c= Y in their place, whenever we applied a theorem we would take up many lines simply citing to Mizar that, indeed, every single one of these twenty-seven or so hypotheses is satisfied by the environment we are currently in, before we could get around to actually using the theorem.

Still, Mizar proofs are already long and detailed, and having to type in even more steps, tedious as it sounds, is preferable to the trouble of looking up a dozen new definitions several times, until the definitions are finally burned into memory from sheer repetition, for every MML file one builds upon in one's proof.

Another argument against types is that they obscure the nuts and bolts of mathematical concepts, making it harder to convert between types. One must spend time unrolling layers of definitions, proving equality of underlying basic parts, and then recompiling the basic parts into the new type. This happened when I failed to find the SetSequence type on an initial search for helping me define the partition piece list for a simple function. I found the SetSequence type later, by accident, after already defining the partition piece list myself. I considered trying to write a lemma that would convert my creation into a SetSequence, so that I could use a useful theorem on SetSequences, but after some analysis I judged that it would be easier to write the theorem for my own partition piece list from scratch. The proof was fairly long, but it was still comparable to the work I would have had to do to convert to the SetSequence type.

A problem with clusters themselves is that they introduce unreliability. At one time, I was trying to figure out how to show that if x is real, then |.x.|, the absolute value of x, is real also. Searching the MML, I found a cluster in COMPLEX1:

```
registration let z be complex number;
  cluster |.z.| -> real;
```

Figure 4.3.2.1. A cluster from COMPLEX1.

Since any Real is also a complex number, I attempted to use reconsider to cast x to complex number, and then presumably the cluster would take care of the rest. However, trying to cast x to complex number created environment errors, and I did not feel like modifying my environment at that point, having recently resolved issues with ordering some of the lists. I went to look for a workaround.

Now imagine if Mizar were really bare-bones: no clusters, no types, just predicates. This particular step would have been simple. There would be a theorem that said for x st x in REAL holds x in COMPLEX and a theorem that said for z st z in COMPLEX holds |.z.| in REAL, and these kinds of modus ponens applications never fail in Mizar. It would be nice if every time one went to apply some logic involving a cluster, one had that degree of assurance.

The idea of this simplified system comes from Coq, where one works with basic predicates and equality almost exclusively, and one gets used to applying many small obvious logical steps to minute parts of data structures, instead of thinking in broader human terms that resemble Mizar clusters.

4.3.3. Operator overloading and mistaken identities

The single largest avoidable issue that I encountered during this formalization was the issue of operator overloading. Convenient in programming languages, in proof verification it seems to create a lot of trouble with little benefit.

Here is a particularly confusing example. In measure theory, we have the idea of a universe X and a sigma-algebra S which is a set of “nice” subsets of that universe X. From here we can talk about “nice” functions which are S-measurable. That simply means for any subset of the range of the measurable function, the preimage of that set will definitely be in S. Put another way, all preimages are well-behaved.

Some measure theorists apply the term measurable to sets as well to mean that the set is a member of the sigma-algebra, in keeping with the idea that the sigma-algebra contains all sets that behave properly with the measure function. The authors of MESFUNC1 define the binary predicate is_measurable_on to mean this, with the first argument any set and the second argument the sigma-algebra.

```
definition
  let X be set;
  let S be SigmaField of X;
  let A be set;
pred A is_measurable_on S means :Def11:
A in S;
end;
```

Figure 4.3.3.1. The first definition of `is_measurable_on`.

Later, they overload `is_measurable_on` to have a different meaning when the first argument is a function from `X` to the extended reals, the expected meaning that the supplied function is `S`-measurable. What if a user intending to use the second definition of `is_measurable_on` accidentally supplies a non-`PartFunc` argument, perhaps because he got his variable names mixed up? Normally, Mizar would generate a typing error and immediately alert the user to his mistake, but since the original definition takes `sets` for the first argument, and everything in Mizar has the type `set`, Mizar will accept the phrase as written. Mizar will probably report a logical error though, because whatever the user is formalizing here is probably talking about measurable functions, not set membership. I made this very mistake in my code, and seeing the logical error, I spent considerable time trying to track down the failure in logic, when all along the real problem was that Mizar was not using the version of `is_measurable_on` that I thought it was. It defeats the point of predicates that take arguments of specific types, if when one puts in a bad type, Mizar cannot report that one did so. This is one way operator overloading can be detrimental.

Another serious instance of mistaken identity that arose from operator overloading occurred with the overloading of the `*` operator, used to denote composition of both functions and relations. This may also say something about the inherent difficulties with having many Mizar cooks contributing to the same broth: the author of the MML files on relations was not the same person who wrote the MML files on functions, but both wanted to use `*` to represent composition, and in different ways.

The original snippet of code that uncovered the problem:

```
M is Function of S,ExtREAL;
p is Function of Seg 1,S;
p*M is PartFunc of Seg 1,ExtREAL;
M*p is PartFunc of Seg 1,ExtREAL;
```

Figure 4.3.3.2. The composition operator `*`.

Ignoring the difference between `Function` and `PartFunc` for now (it does not cause problems here), as well as the meanings of `Seg 1`, `S`, and `ExtREAL` (in this example, they are just sets), which of the third and fourth statements should compile correctly? Given that `M` and `p` are functions, usual mathematical notation reverses the order of application, so since `p` outputs an element of `S`, and `M` accepts elements of `S`, the proper notation would be `M*p`.

Mizar actually compiles the third line as correct and the fourth as wrong. It turns out that the author of the MML file on relations, RELAT_1, defined the * operator to denote composition when applied to relations, but in the opposite order that the writer of FUNCT_1 defined it. Since functions are relations (Relation is a supertype of Function), one might ask the question of how Mizar decides which * operator to use when faced with two functions applying it. Might it be that Mizar chooses whichever definition came first in the environment lists (in other words, whichever definition Mizar "loaded" first)? No, because swapping the order of RELAT_1 and FUNCT_1 in the environment does not move the error from the fourth line to the third. In the end, only a fix I found from another MML file that also used both RELAT_1 and FUNCT_1 solved the problem, and I still am not sure exactly how:

```
notation let f,g be Function;
synonym g*f for f*g;
end;
```

Figure 4.3.3.3. Reconciling the two definitions of the composition operator *.

This seems to beg the question of how Mizar would decide on one version over another when faced with the * operator being applied to two functions. Resolving this issue was both confusing and time-consuming (looking for special code to reconcile the two versions of * was not part of my typical plan of attack for typing errors), and the issue would not have arisen if Mizar forced authors to adopt unique notation for composition of relations and composition of functions. (This is another benefit that Coq demonstrated; its type semantics prevents these kinds of confusion with operator overloading.)

4.3.4. Operator overloading and syntax

Accommodating operator overloading can lead to bizarre constructions. In this example, we use MEASURE6:16 to convert an extended real number to a real number when we know that the extended real number is sandwiched between two finite reals.

```
theorem
      for x,y,z being R_eal holds
   x is Real & z is Real & x <= y & y <= z implies y is Real
```

Figure 4.3.4.1. Theorem for converting extended real to real.

Notice that it uses the <= operator to show the sandwiching, and that the arguments are type extended real (R_eal). This leads to difficulties trying to apply the theorem. One must ensure that the x <= y and y <= z preconditions one supplies are using the proper version of <=, the version that has been overloaded to work with R_eals. Since Reals are R_eals (every real number is an extended real number), this would seem to be straightforward, and often is in similar situations, but for some reason, this particular operator overloading combined with the

structure of MEASURE6:16 required a unique construction. Here is the simplest solution I could find that works:

```
for i be Nat st i in dom F holds(M*F).i is Real
& 0<=(M*F).i & (M*F).i<=1 proof
 let i be Nat;
 assume C0:i in dom F;
 C1:F.i in S by B1082,C0;
 C2:(M*F).i=M.(F.i) by B1120,C0;
 M is nonnegative by MEASURE1:def 11;
 then 0. <= M.(F.i) by C1,MEASURE1:def 4;
 then C3:0 <= M.(F.i) by SUPINF_2:def 1;
 reconsider Zer=0 as R_eal by SUPINF_1:10;
 reconsider One=1 as R_eal by SUPINF_1:10;
 M.(F.i) <= M.X' by MEASURE1:62,C1;
 then M.(F.i) <= 1 by A0;
 then 0<=(M*F).i & (M*F).i<=1 by C3,C2;
 then Zer<=(M*F).i & (M*F).i<=One;
 then (M*F).i is Real & 0<=(M*F).i & (M*F).i<=1 by MEASURE6:16;
 hence thesis;
end;
```

Figure 4.3.4.2. Instantiation of extra variables.

The instantiation of the Zer and One R_eal wrappers for the numbers 0 and 1, bolded above, are necessary. This was the only time in Mizar that I needed to create wrapper variables to resolve a typing error.

4.3.5. Operator overloading and user-friendliness

A final way operator overloading can create problems for users is in blurring users' abilities to easily see what is going on. Mizar always has a better handle on the current types of variables than its users do, because users often declare and modify the types of their variables (with reconsider) in places far off from their current section of code. A Mizar author can use the reserve keyword to localize type declarations in a standard place, but still must flip back and forth to remind herself of the types of variables. reserve also cannot completely overcome the localization of type information problem, because reconsiders are often the most direct way to fix typing errors. (This may be an argument for removing the current ability of Mizar to reconsider the *original* variable to a different type, instead only allowing the secondary mode of reconsider, which creates a new variable with the desired type and leaves the original one untouched.)

With Mizar as it is now, users have no confidence when looking at an expression that they can interpret it properly. They must always search throughout theirs' and others' code for type-modifying statements, and use test statements often to refresh their memory. This is solely because operator overloading allows a keyword or symbol to change meanings depending on its

arguments, and arguments are only ambiguous in the first place because type information is invisible lexically. Again the type system interacts with a feature of Mizar (operator overloading currently, previously clusters) to create an unfriendly environment for the user.

For an illustration, here is another sample from my code for Markov's inequality:

```
reconsider a=Seg 1-->0. as Function of Seg 1,ExtREAL by FUNCOP_1:57;
A90:a is FinSequence of ExtREAL proof
 dom a = Seg 1 by FUNCOP_1:19;
 then reconsider a as FinSequence by FINSEQ_1:def 2;
 rng a c= ExtREAL by RELSET_1:12;
 hence thesis by FINSEQ_1:def 4;
end;
then reconsider a as FinSequence of ExtREAL;
A91:Sum(a)=0. proof
 a = 1|->0. by FINSEQ_2:def 2;
 then a = 1|->0 by SUPINF_2:def 1;
 then B0:a = <*0*> by FINSEQ_2:73;
 then reconsider a as FinSequence of REAL;
 Sum(a) = Sum<*0*> by B0;
 then Sum(a) = 0 by RVSUM_1:103;
 then B1:Sum(a) = 0. by SUPINF_2:def 1;
 reconsider a as FinSequence of ExtREAL by A90;
 Sum(a) = 0. by B1;
::>           *4
 hence thesis;
end;
```

Figure 4.3.5.1. Unfriendly code.

Most of the above code is background and the reader can ignore it; the important lines are in boldface type. Just from a user-friendliness standpoint, a proof checker should not say to its user, "The fact that `Sum(a) = 0.` does not imply that `Sum(a) = 0.`." (`0.` is the symbol for zero with the type extended real.) The reason, of course, is that the two `Sum(a)`s are actually different expressions, with different meanings, as one is the sum of a sequence of reals and the other is the sum of a sequence of extended reals. Note the `reconsider` changing `a` from `FinSequence of REAL` to `FinSequence of ExtREAL`.

Besides emotional distress, this particular piece of code led me to embark on a chain of work that turned out at the end to be irrelevant. As shown, I was working to show `Sum(a)=0.`, a necessary fact for a different part of the proof. This should not be too difficult, I thought, since `a` is a singleton sequence containing `0.`. I found a chain of theorems: `0 = Sum<*0*>` and `<*0*> = 1|->0` and `1|->0 = Seg 1-->0.`, which is exactly what I defined `a` to be in the first line of the sample code. But writing this out, it was only at the last link that the error appeared; only at the end did I notice that the theorems in my chain were dealing with `FinSequence of REAL`s, not `FinSequence of ExtREAL`s. All along the `Sum(a)` I was equating to was not the `Sum(a)` I needed. If the authors of the MML had not overloaded `Sum()`, I would have known from the start that I needed to find theorems about finite sequences of extended reals.

4.3.6. The legacy of MML authors

In a way, the problems with operator overloading are not the fault of operator overloading itself, but related to a general issue with proof checker systems. After all, when mathematics textbooks or literature set up their conventions of terminology and notation, the worst that can happen is that the reader must learn to think in an unusual notation for the duration of this textbook or paper. Once the user has absorbed the meaning of the paper, she is free to build on that knowledge using whatever notation she prefers. But in formal proof checker systems, future users are more or less stuck with adopting in their own work any previously created notation. If one wishes to use a different notation, one either has to rewrite each file one uses from the MML to use that notation, or write some conversion theorems that can mediate between logical statements in one's preferred notation and statements formatted in existing notation. Either way, this is too much to ask.

4.3.7. Mathematics is not uniformly canonized

That concerns different notations for the same concept, but what about the same notation for different concepts? What if an MML author's definition of a foundational piece differs from a future proof writer's concept of it? The writer must first understand that he has found someone's attempt to define whatever particular term or theorem he is looking for, and then he must convince himself without the benefit of human interaction that the piece he is about to use is indeed the same piece that he originally thought he would be using.

The example that came up in my own experience was researching the existing definition of Lebesgue integral. The definition in `MESFUNC3` had the unusual property that it did not specify over which subset of the measure space's universe to take the integral. (This is analogous to being unable to take a Riemann integral over the closed interval [0,1] instead of the entire real line.) This definition of Lebesgue integral loses no generality, because any subset of a measure space is a measure space. It should be able to be used, with a little massaging, in any case in which the more common definition could be.

Nevertheless, this reminds us that one cannot simply add a comment "Lebesgue integral" next to a definition and have that be the end of the story. There are different definitions of Lebesgue integral, measure space, measurability, and just about any other complex concept in mathematics, and without actually looking at the exact axioms or hypotheses described when an MML author defines their terms, one cannot be sure if one's proof is on the foundation one thinks it is.

For another example that arose in this formalization, let us look at the extended real numbers. Pop quiz: does 0 multiplied by infinity equal 0? I believe this is not necessarily true of all definitions of extended real numbers. Some systems have this hold and others leave the operation undefined. At any rate, I found myself in a situation where I needed to prove this, and automatically I went the usual route of searching for a theorem that would prove this (assuming it was derived from basic axioms), before it occurred to me that it might be an axiom in the definition of the `ExtREAL` set itself, which it turned out to be.

Now, although this particular example did not take that long to figure out, this idea might cause problems more generally. Suppose some mathematicians are using measure theory to prove a theorem, and they proceed happily using theorems out of `MESFUNC3.MIZ`. Suddenly, they hit a snag in a particular lemma. Puzzled, and having trouble finding out exactly why the logical step is failing, they trace backwards through the relevant terms, unrolling definitions, until they finally find the reason – the writers of `MESFUNC3.MIZ` had used a different set of assumptions to define their measure theory to begin with! In other words, because the basic assumptions of measure theory have not been canonized, one runs the danger of trying to use a library or set of theorems that are not talking about what one really needs. The upshot is that one must verify each library and its assumptions before use. Since libraries build upon other libraries, the farther afield one's topic is from set theory's fundamentals, the more work this is.

4.3.8. Reinventing the wheel

This is not the theorem you're looking for. – Obi-Wan Kenobe, if he were a user-defined proof checker library

When browsing MML files, it becomes apparent that comments really do go a long way. It is true in computer programming, but it is even more pertinent in Mizar because variable names are more terse and obscure. MML writers have to think up names for all their intermediate variables, and due to referring to these variables many times over the course of a proof, just as in normal mathematical writing they choose short, often one-letter names. For example, when I was trying to find out if there were a theorem stating that the sum of a sequence equals the sum of that sequence with all zeroes removed, I ran into new notation that I had never seen before. In addition to the alphabet soup of variables names themselves, the terminology was mostly new: `Ser()`, `vol()`, `SUM()` (all caps, not the `Sum()` I knew about).

Experience has taught me that sometimes it is easier to reinvent the wheel than to learn other authors' uncommented, mysterious syntax well enough to determine if they have proven a theorem I need. Depending on how difficult the needed theorem is, one has to make a judgment call of whether it is likely to be worth examining MML files for an hour or two.

A policy of casual documentation would help; a few sentences on the meaning of each variable and term should be enough to communicate the intents of each to a human reader. As noted above, though, this is more for a user roadmap; it does not get around the problem of authors having different ideas of what mathematical terms mean.

5. Comparison

We discuss a few high-level differences between Coq and Mizar.

5.1. Underlying theory

The semantics of Mizar is fairly straightforward. Mizar is about ZF style set theory with first order logic
–Dr. Freek Wiedijk, Mizar: an Impression, p. 10

Coq is a proof tool based on a type system with inductive types called the Calculus of Inductive Constructions (CIC). Through the Curry-Howard isomorphism, proofs are identified with terms and proof-checking with type checking; the construction of a proof then becomes simply the interactive construction of a term which is at the end type-checked. –Dr. Luís Cruz-Filipe, Formalizing Real Calculus in Coq, p. 2

As I have alluded to before, understanding the underlying theory of Coq will be one of the main challenges for a new user. The Calculus of Inductive Constructions, the general concept of constructive logic, and the Curry-Howard isomorphism are not widely known outside the computer proof checking community. In contrast, any mathematician seeking to check his proofs by computer will be relatively familiar with set theory and first-order logic, which is all one needs to know about the underlying theory of Mizar.

Besides the main concept of "proofs as types," the Calculus of Inductive Constructions theory has a lot of subtleties, each of which was a challenge for me to get used to. One involves the idea of equality. Suppose we have two sets, each containing the natural numbers 1, 2, and 3. However, the first set we build up in the order 1, 3, 2 (perhaps because we apply the `if-then` construction of Coq in that order) and the second set we build up in the order 1, 2, 3. In that case, according to Coq, the two sets are not equal, since their structure is clearly different. Even though we could prove that the two sets contain exactly the same elements, this is immaterial from the viewpoint of Coq's "=" operator. Of course, most mathematics defines equality for sets based on whether or not they contain the same elements regardless of how the sets were "constructed" (indeed, in most mathematics there is not even the idea of "constructing" a set – this is analogous to the issue in computer programming that every group of data has an order to it, the order that the data is laid out in memory, whether or not the programmers want to think about the data as being ordered at all). In each of the three major libraries I used, there is a provision for allowing Coq to have a more traditional idea of equality. C-CoRN defines an equals operator `[=]` for all its types, and Pottier-Stein similarly defines multiple versions of the `Equal` predicate. The Coq standard library has the `Extensionality_Ensembles` axiom to essentially modify the equality operator to accept sets which have proofs that they are subsets of each other as equal, as opposed to only accepting sets which have exactly the same structure as equal.

5.2. Constructivity

Being able to enter a fully constructive proof into Coq and have it generate an algorithm for computing an object stated to exist by the theorem (extraction) was one of Coq's main goals from the beginning (Filliâtre, 2000, p. 2; Letouzey, 2003, p. 1). Kleene first described this idea of generating algorithms from proofs, called realizability, in 1945, although the idea exists in Gödel's work dating as far back as 1932 (2000, van Oosten, pp. 2-3).

One of the main aspects of constructivity new Coq users must wrap their heads around is the difference between being able to state that something exists and actually being able to compute it. A basic rule of constructive logic is that, if one has proved that for all x there exists a y such that property $A(x,y)$ holds, then that means one has a process by which one can, given an x, produce the actual y satisfying the property $A(x,y)$. In classical logic, the statement of existence does not inherently bring along a way to specify or describe the y; in other words, classical logic is looser and less stringent about letting mathematicians say things exist.

The real numbers provide an interesting case study in constructivity. A common way to constructively define the reals is as infinite sequences of rationals that approach a certain value (the real number) with increasing accuracy. One valid way to mandate this accuracy is that for a real number r, any two rationals in the sequence r_n and r_m must satisfy the condition $|r_n - r_m| < 1/n + 1/m$. From this, logically two reals r and s are indistinguishable, and thus equal, if every pair of terms from the sequences satisfies $|r_n| - |s_n| \leq 2/n$. Since the sequences are infinite, we can never finish a computation to conclude that every pair from the sequences satisfies this rule, and this lack of computability equates to a lack of a proof of equality in pure constructive mathematics.

A new Coq user will likely first run into this difference between constructive and classical logic in Coq when he first encounters the wall between the `Prop` domain and the `Set` domain. Although it is not obvious unless one reads the manual and learns some of the underlying workings of Coq, there are two "modes" of proof writing in Coq, `Prop` and `Set`. A major separation occurs when one looks at the `exists` keyword, which talks about existence in the `Prop` domain (any statement of the form "`exists` [something]" in Coq is an object in the `Prop` domain), and the `sig` keyword, which is its counterpart in the `Set` domain. One may, as I did, get used to using the `elim` tactic to convert an `exists` statement into the claimed object itself, which is the only way to use existence hypotheses to produce new existence theorems. This works fine until one attempts to use an existence hypothesis from the `Prop` domain to build a proof in the `Set` domain. The `Set` domain is the true constructive arena we talked about above. The minute one jumps into trying to build a proof in the `Set` domain – perhaps because one is starting to use a new user-defined library, or section of library that works in that domain – one finds that all one's storehouse of knowledge built on `exists` keywords is moot. New users must be cautious of this to avoid building a large database of theorems in the `Prop` domain and suddenly having to redo it due to not entering the `Set` world early enough.

5.3. Readability

A major difference between Mizar and Coq is apparent when browsing the user-defined contributions of each system. Coq's expression syntax is harder for humans to parse than Mizar's. Consider the following statement of a well-known theorem in Coq. First are the variables and preconditions, followed by the statement.

```
Variable J : interval.
Variable F : PartIR.
Hypothesis contF : Continuous J F.
Variable x0 : IR.
Hypothesis Hx0 : J x0.
Hypothesis pJ : proper J.
Variable G0 : PartIR.
Hypothesis derG0 : Derivative J pJ G0 F.
Let G0_inc := Derivative_imp_inc _ _ _ _ derG0.

Theorem : forall a b
 (H : Continuous_I (Min_leEq_Max a b) F) Ha Hb,
 let Ha' := G0_inc a Ha in
 let Hb' := G0_inc b Hb in
 Integral H [=] G0 b Hb'[-]G0 a Ha'.
```

Figure 5.3.1. A Coq theorem statement.

Despite some descriptive term names like `Derivative`, `Continuous`, `interval`, `PartIR`, (partial function from the reals to the reals) and `Integral`, it is difficult to interpret this. The meaning of arguments with respect to their terms is not as clear as in Mizar. For example, why is the only argument of the `Integral` expression a single precondition `H`? Does not an integral require a function and an interval at minimum? (`Integral` actually does require those; the explanation is that all the needed information is contained implicitly in `H`.) At any rate, this is Barrow's rule, a formulation of the fundamental theorem of calculus, the main goal of the C-CoRN library.

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

Figure 5.3.2. The real Barrow.

Let us look at the Mizar formulation (`INTEGRA5:13`):

```
for f being PartFunc of REAL,REAL st A c= X & f is_differentiable_on X &
f`|X is_integrable_on A & f`|X is_bounded_on A holds
integral(f`|X,A) = f.(sup A)-f.(inf A)
```

Figure 5.3.3. A Mizar theorem statement.

If I mention that `f`` is the derivative of `f`, this statement is almost conventional mathematical writing. This is pretty typical of Mizar. As mentioned in the Mizar analysis, Mizar authors tend

to create new terms for most concepts, here including differentiability, integrability, and boundedness, not to mention integral, supremum, and infimum, and the ` notation for derivatives. If we add and rename some terms in the Coq version to improve readability, the result is still murky:

```
Variable J : interval.
Variable F : PartFunct_Real.
Hypothesis contF : is_continuous J F.
Variable x0 : Real.
Hypothesis Hx0 : J x0.
Hypothesis pJ : proper_interval J.
Variable Fprime : PartFunct_Real.
Hypothesis is_deriv : Derivative J pJ G0 F.
Let included_in :=
 Is_deriv_implies_interval_is_included
 _ _ _ _ is_deriv.

Theorem : forall F:PartFunct_Real,
 forall a:Real, forall b:Real,
 (H : continuous_on (interval_well_defined a b) F) Ha Hb,
 let Ha' := Is_deriv_implies_interval_is_included a Ha in
 let Hb' := Is_deriv_implies_interval_is_included b Hb in
 Integral H [=] Fprime b Hb'[-]Fprime a Ha'.
```

Figure 5.3.4. Coq statement of Barrow's rule with renaming.

The unorthodox placing of the arguments to terms such as `included_in`, `Is_deriv_implies_interval_is_included`, and `continuous_on`, and the lack of a clear operator for function evaluation make this about the best one can do.

6. The Book proof

There is a book by Martin Aigner and Günter M. Ziegler called *Proofs from THE BOOK* (Wikipedia, 2007). This is a reference to a saying of Paul Erdős about how God has a Book which contains perfect, the "most elegant," proofs of all mathematical facts. It makes sense to try to come up with an elegant, succinct handwritten proof before trying to convert it into Mizar or Coq. Optimizing the human proof by minimizing the number of steps should decrease the formalization time proportionally.

I used this strategy in the formalizations described in this paper. First I browsed basic explanations of the theorems, and then sat down and wrote a complete proof by hand, polishing it until I felt it elegantly captured the essentials and nothing more. Let us look at how this optimization bore out during formalization.

6.1. Cons

One inevitably makes changes to one's Book proof as one progresses through the formalization. Some logic that is easy to understand for humans is difficult for computers, and vice versa. For example, in the Wikipedia proof of Markov's inequality, the writers state as a single step that, given a random variable f and a particular set of points where f takes on values greater than a constant a, a multiplied by the indicator function for f is less than or equal to f for all points in the measure space universe. With a little thought, the reader can see that this is true where the indicator function is 0, because a is mandated in the assumptions to be greater than 0, and this is true where the indicator function is 1, because by the definition of indicator function, f is greater than or equal to a at precisely those points.

However, using this as a step in Coq or Mizar requires the knowledge of a fact that you can pull out a constant from a Lebesgue integral in general. This fact is lengthy to prove. So, in the formalization, we restricted ourselves to only proving the fact for simple functions. One often chooses data structures or lemma statements that represent easily in the proof checker system over structures or statements that seem elegant or most natural to humans.

Streamlining a pencil-and-paper proof beforehand is still necessary, because it gives a clear understanding of the proof from start to finish. However, it is best not to think about the data structures and minute lemmas when writing the handwritten proof, leaving that for when one begins browsing the user-defined libraries of the proof checker system.

6.2. Pros

A good thing about writing out a Book proof first is that it forces one to decide on what sets of assumptions and versions of used terms one plans to use. I have been tempted to "cheat" when looking too soon at the actual Coq or Mizar terms I will be using in the formalization.

To give an example, another step in the Wikipedia proof of Markov's inequality is that the probability that $f \geq a$ equals the expected value of the indicator function. Where does this fact come from? Might not some mathematicians define "probability" in measure theory to exactly to be the integral of the indicator function of the event? But in that case, a large step of the proof is just a definition! On the other hand, mathematicians might define probability in any number of other ways, perhaps, just to give an example, as the limit of some carefully chosen expression. In that case, I would be in for a very long lemma equating that complex definition to the Lebesgue integral of an indicator function. The reader can see how it can be tempting to choose the easy definition that absorbs most of the work into an axiom.

How about definitions of measurability? Some authors define a function to be measurable if the preimages of every subset of its range are in the sigma-algebra. This is a natural and direct

definition. But some authors choose to define it in a more restrictive way to avoid talking about intractable, badly-behaved sets. They define measurability as the property that all half-lines of its range have preimages in the sigma-algebra. Both definitions are fairly common. So which shall we choose? The first one, of course saves us more work.

6.2.1. "Cheating" in Coq

The first time the idea that I might be "cheating" in some way occurred to me was during formalizing Markov's inequality in Coq.

I was describing the structure of a countable collection of subsets of the measure space, as part of writing the definition of measure. (The sets in the collection can be thought of as intervals of the real line, except that they can be "spotty," and we are interested in the additivity of measure, that is, what the measure of the union of this collection of intervals will be.) For proving certain lemmas in Coq regarding this union of a countable collection, the most convenient way to represent the collection was a serial numbering of the sets, and a separate map taking the serial numbers to the measures of the sets. But it occurred to me that the future user of my statement of Markov's inequality (after all, theorems are meant to be used to prove other theorems) would probably find it more natural for his own work to simply have a map taking each element of the collection to its measure, not this intermediate serial-numbering system.

Thinking about how this idea might apply elsewhere in my proof, I returned to my definition of countability. I had chosen, again, the definition of countability that made my future work the easiest. (I actually did not realize before this proof that there were so many ways to define countability: with injective maps, with surjective maps, and with relationships with subsets of the natural numbers.)

I decided that I should at least go with the two most common formulations of countability, and prove them equivalent. Then I would not be clearly foisting as much work as possible upon the user – I was simply choosing what honestly seemed to me to be the two most accepted formulations and proving their equivalence so that I could use either one without feeling lazy. The proofs of equivalence (both ways, showing that each definition implied the other) were fairly complex and took 45 lines of compressed code, so this decision was not without consequence.

Despite that, in the end I realized that this issue is not so clear-cut. The Coq standard library itself does not contain an official definition of countability. In fact, as far as I could tell, the concept is only used once in the whole standard library. In `Coq.Logic.ConstructiveEpsilon`, Yevgeniy Makarov formulates his own version of countability for a small proof at the end.

When I saw an author of the Coq standard library himself choose a definition of countability that is most natural for his own purposes, exactly engaging in this practice of removing the most work from his proof, it struck me that in some sense this is justified. The author of `Coq.Logic.ConstructiveEpsilon` may know that the Coq standard library will later come up

with proofs of equivalence of all formulations of countability, and perhaps such a basic concept should be kept the domain of the (rest of) the standard library anyway.

6.2.2. Mizar

Of course, this phenomenon is not restricted to Coq. In Mizar, it happened to me at the very beginning of my formalization of Markov's inequality, during the definitions phase. One of the first things needed is the idea of the set of all elements of the measure space universe that *f* maps to something greater than or equal to *a*. In this picture from the Wikipedia proof, it is the two segments at the bottom which correspond to the portion of *f* that rises above the dotted line.

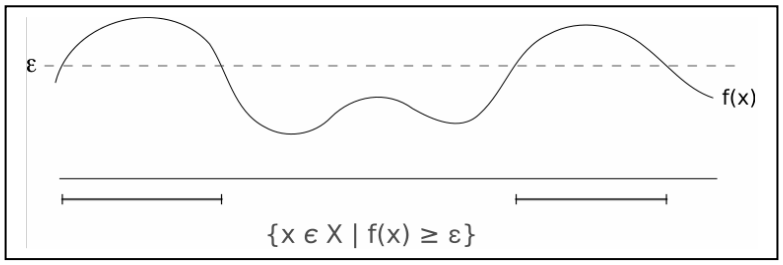

Figure 6.2.2.1. Illustration of the set of all elements that *f* maps to something greater than or equal to ε.

The dilemma was, do I state the theorem of Markov's inequality such that "for all sets that exactly capture this set of points that *f* maps to something greater than or equal to *a*..." or do I say "for a given fixed real constant *a*, there exists a unique set which captures all elements that *f* maps to something greater than or equal to *a*..." On the surface, the main difference is that one way requires me to prove something extra that the other does not, the existence and uniqueness of a certain set. But the important question is, which is the "fair" way to state Markov's inequality? Is it really my problem to establish whether or not such a set exists and is unique, or should I concentrate on just the logic of Markov's inequality itself? If it is true that some standard library should cover this for me, because it is really a fundamental fact and more or less tangential to the real issues being addressed by Markov's inequality, I do not want to duplicate their work when they do get around to it. (The C-CoRN developers saw the consequences such a choice can have for the size of one's proof. In their formalization of the fundamental theorem of algebra, the constructive definition of and basic theorems about the real numbers took up 865 KB of Coq code, while the code directly related to the logic of the theorem's proof was 65 KB (Geuvers et al., 2001, p. 4)).

The situation as it actually arose turned out to have a solution that had its cake and ate it too. It turned out to be almost trivial (as the reader may already have questioned) to say that this set exists and is unique by the axiom of separation. It was not completely trivial because I had to familiarize myself with the Fraenkel operator in Mizar, one of the tougher pieces of syntax, to use this axiom. Still, these questions of fairness and putting work upon others persist in general.

7. Conclusions

We make observations on the future of proof checking in general, based on the experiences of doing these formalizations in Coq and Mizar.

7.1. An eye for detail

Formalizing a proof forces one to look at mathematical details that one otherwise overlooks. One example I encountered concerns the definition of measurable functions. I have a function *f* which is measurable on the sigma-algebra, a basic requirement to take a Lebesgue integral. Since most of the MML deals with `PartFunc` (partial functions), and to be consistent I had typed *f* as a `PartFunc` also, is it necessary that the domain of *f* equals the entire universe of the measure space to know that *f* is measurable on the sigma-algebra? After all, measurable simply means that for every subset of the range, the preimage of that set is in the sigma-algebra. But what exactly does this statement mean when *f* is only partially defined on the universe of the measure space? The preimage will be the intersection of the portion of the universe upon which *f* is defined and the "true" preimage of the range subset. Will the preimage always be contained in the sigma-algebra? In general, it seems not, since the partial function's domain could be any badly-behaved subset of the universe.

The paper formulations of Markov's inequality that I read were mute on the subject. It seems that this concept of measurability as related to partial functions is too obscure for most people to notice, myself included, even in expositions of measure theory intended to be fairly comprehensive tutorials and explorations of the basic definitions. It is the sort of thing that only comes to the forefront when one begins instantiating variables and data structures for a computer formalization of a proof.

In general, the exact nature of computers when formalizing mathematics in proof checker systems trained me to develop an eye for detail. For example, I had to reformulate my four axioms of what it means to be a simple function several times. The concept of a simple function is easy, but I kept uncovering errors and things I had overlooked in my formulations as I used them in later sections of the proof. I learned that it is one thing to understand the concept of a function with a finite number of values in its range, but it is another to be able to formulate that in terms of precise data structures and statements about restrictions on those data structures.

One such mistake involved the axiom describing the finite list of range values. The natural way to represent a finite list of reals in Coq is a function from the natural numbers to the reals. For this list to be a legal representation of the range values of a simple function, we add the restriction that we only care about the portion of this function below a certain fixed natural number *n* representing the end of the list. For clarity and to make future lemmas easier, I mandated that the range values be distinct as well. That is, I did not want to allow the possibility

of the simple function being broken up into partition pieces such that there were multiple partition pieces all mapping to the same real number.

Long after writing this axiom, I was creating an actual simple function, the indicator function of an event *A*. In other words, *A* was a subset of the universe domain *X*, and its indicator function would be a simple function mapping points in *A* to 1 and points outside of *A* to 0. As another simplification to make life easier, instead of mapping 0 to the complement of *A*, and 1 to *A*, and natural numbers from 2 to infinity to some null value, I just mapped 0 to the complement of *A*, and all other natural numbers to *A* – why? Because that required only one use of the `if-then` construction in Coq (it is somewhat cumbersome to work with).

Having created the indicator function, I proceeded on, and suddenly was amazed to hit a brick wall: looking at the current list of known facts and comparing it with the current goal, I saw that I was attempting to prove a falsehood. I had stated some hypothesis incorrectly. After some analysis, I realized the culprit was not the current lemma's preconditions, but my old axiom of simple functions! The current proof depended on the fact that all partition pieces would have a distinct range value. But, in my axiom, I had neglected to add in a note that I only cared that range values would be distinct for the "relevant" natural numbers, that is, the ones below the length of the list! I had left out the phrase "for natural numbers less than or equal to *n*" from the distinctness axiom, again, for simplicity. Since at the moment, all natural numbers from 1 to infinity were mapping to *A*, in terms of the current language I indeed had multiple "serial numbers" of partition pieces mapping to the same range value, contradicting the axiom! It impressed me that an innocuous concept like the partitioning of a simple function could lead to mistakes like this.

If I were to sum up the main difference between pencil-and-paper mathematics and computer formalizations in a single phrase, it would be "instantiation of data structures."

7.2. Subjective experience

Although I find Coq much harder to learn than Mizar, I do want to mention something good about this. One has to come up with creative solutions to problems caused by Coq's many nuances, and this makes the process of becoming proficient with Coq continually new and satisfying. Figuring things out in Mizar, including learning new syntax and user-defined terms, is like debugging a memory dump. One *knows* that one can find the answer, eventually, given enough time and poring over pages of data (or, in Mizar, using many test statements to narrow down the causes of an error). However, this kind of problem solving is not very inventive or rewarding. In Coq, there is always the element of wondering, "Will Coq stymie me this time or will I be able to figure this one out?" After reaching a certain level of experience with Coq, having gotten through enough problems (with help from others or without) and having mastered a significant portion of the system, some of the worries about never being able to figure out this

system disappeared. Feeling a little more confident, it started to become fun learning each new layer, digging deeper into Coq's world. Working on a formalization became something I would look forward to.

Most of this paper has been devoted to criticism and problems with Coq and Mizar, but I do not want to give the wrong impression. After some initial diving into the systems followed by two medium-sized proofs in each, I feel competent enough with Coq and Mizar to formalize theorems in the accepted amount of time (several months for a group to formalize a fairly complex theorem, longer for an individual). So while learning them was definitely challenging and time-consuming, the experiment succeeded.

For the person who wants to pick up a system and begin checking proofs as quickly as possible, I would suggest Mizar. Mizar's learning curve flattens relatively rapidly compared to Coq's. One fairly quickly gets to the point where the majority of one's time is spent on busy work (the necessary work), and not always having to learn new syntax.

7.3. General benefits

The main benefit of proof checking is a standardized method of verifying proofs, especially for ones like Kepler's conjecture which required computer intervention either way. By having a system universally considered reliable, as time goes by and the system maintains its reliability, the confidence of users only increases. It may also take less time than human verification of proofs, as also noted in the Flyspeck project, where the committee of referees felt that verifying the proof by hand, while possible, was too laborious.

Acknowledgements

I would like to thank my advisor for this project, Dr. Michael Beeson, for his encouragement, proofreading, suggestions for revisions, and the general idea. Dr. Beeson also corrected some of my misunderstandings of constructive logic. I also thank Dr. Philippe Audebaud, Dr. Gilles Barthe, Dr. Thierry Coquand, Seokhyun Han, Dr. Christine Paulin, and Dr. Jan Reimann for responding to my inquiries. I would like to thank Lionel Elie Mamane and Nickolay Shmyrev for their extremely helpful answers to my questions on the Coq-club mailing list. I especially wish to thank Russell O'Connor and Jasper Stein for teaching me several of the ins and outs of Coq. I would like to thank the committee members who attended the defense of this paper, Dr. Michael Beeson, Dr. Chris Pollett, and Dr. Mack Stanley. I would also like to thank Deanna Diaz, Dr. Horstmann, and Dr. Louden for their help with administrative issues.

Bibliography

Alama, Jesse. (2007, February 17). Re: referring to unlabeled theorems. Message posted to http://www.nabble.com. Retrieved April 25, 2007, from http://www.nabble.com/Re:-referring-to-unlabeled-theorems-p9022199.html.

Anderson, Robert M. Lecture Notes on Measure and Probability Theory. (n.d.). Retrieved June 26, 2007, from http://elsa.berkeley.edu/users/anderson/Econ204/MeasureTheoryLectureNotesTimeless.pdf.

Ash, Robert B. Some basic techniques of Group theory. (2002, November). Retrieved March 12, 2007, from http://www.math.uiuc.edu/~r-ash/Algebra/Chapter5.pdf.

Audebaud, Philippe, and Paulin-Mohring, Christine. Proofs of randomized algorithms in Coq. (n.d.). Retrieved June 23, 2007, from http://www.lri.fr/~paulin/ALEA/article.pdf.

Barthe, Gilles, Forest, Julien, Pichardie, David, and Rusu, Vlad. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. Retrieved December 27, 2006, from http://www-sop.inria.fr/everest/personnel/David.Pichardie/Publis/genfixpoint.pdf.

Bass, Richard F. Real Analysis. (2007, April 10). Retrieved June 25, 2007, from http://www.math.uconn.edu/~bass/meas.pdf.

Bass, Richard F. Probability Theory. (2001). Retrieved June 25, 2007, from http://www.math.uconn.edu/~bass/prob.pdf.

Bertot, Yves, and Pierre, Castéran. On Well Founded sets and the Axiom of Choice. (2004). Retrieved May 9, 2007, from https://www.labri.fr/perso/casteran/CoqArt/newstuff/notwf.html.

Bogomolny, Alexander. (n.d.). The Inclusion-Exclusion Principle. Retrieved May 7, 2007, from http://www.cut-the-knot.org/arithmetic/combinatorics/InclusionExclusion.shtml.

Bray, Nicholas. (2002, December 3). Subgroup Index. From *MathWorld*--A Wolfram Web Resource. Retrieved March 12, 2007, from http://mathworld.wolfram.com/SubgroupIndex.html.

Byliński, Czesław. Strengthening the Computational Power of the Mizar Checker. (2004, November 1). Retrieved March 30, 2007, from http://www.fnds.cs.ru.nl/typesworkshop/slides/bylinski.pdf.

Cairns, Paul, and Gow, Jeremy. Integrating Searching and Authoring in Mizar. (2006, January 31). Retrieved August 1, 2007, from http://www.uclic.ucl.ac.uk/people/j.gow/papers/alcor-jar.pdf.

Candel, Alberto. (2003). The limit of sin(x)/x as $x \rightarrow 0$. Retrieved January 8, 2007, from http://www.csun.edu/ac53971/courses/math350/xtra_sine.pdf.

C-CoRN -- History. (n.d.). Retrieved December 22, 2006, from http://c-corn.cs.kun.nl/history.html.

The C-CoRN library. (n.d.). Retrieved May 8, 2007, from http://c-corn.cs.ru.nl/downloads/CoRN.tar.gz.

Chen, Beifang. The Inclusion-Exclusion principle. (2005, March 31). Retrieved May 7, 2007, from http://www.math.ust.hk/~mabfchen/Math391I/Inclusion-Exclusion.pdf.

Chicli, L., Pottier, L., and Simpson, C. Mathematical quotients and quotient types in Coq. (2002). Retrieved May 9, 2007, from http://www-sop.inria.fr/lemme/Loic.Pottier/chicli_pottier_simpson.ps.

Chlipala, Adam. (2006, June 16). Equality modulo proofs. Message posted to the Coq-club electronic mailing list. Retrieved July 3, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2006/002404.html.

Chlipala, Adam. Propositional and First-Order Logic. (2006, August 31). Retrieved May 11, 2007, from http://www.cs.berkeley.edu/~adamc/itp/lectures/lecture2.pdf.

Coleman, Mark. Measurable functions. (n.d.). Retrieved June 26, 2007, from http://www.maths.manchester.ac.uk/~mdc/old/341/not4.pdf.

Coleman, Mark. Simple functions. (n.d.). Retrieved June 26, 2007, from http://www.maths.manchester.ac.uk/~mdc/old/341/not5.pdf.

Coleman, Mark. Integration. (n.d.). Retrieved June 26, 2007, from http://www.maths.manchester.ac.uk/~mdc/old/341/not6.pdf.

Coleman, Mark. Integration of measurable functions. (n.d.). Retrieved June 26, 2007, from http://www.maths.manchester.ac.uk/~mdc/old/341/not8.pdf.

Construction of real numbers. (December 22, 2006). In *Wikipedia, The Free Encyclopedia*. Retrieved December 22, 2006, from http://en.wikipedia.org/wiki/Construction_of_real_numbers.

The Coq proof assistant, version 8.0, for Windows. (2006, December 15). Retrieved February 27, 2007, from ftp://ftp.inria.fr/INRIA/coq/V8.0/Coq-8.0-Win.zip.

The Coq proof assistant, version 8.1, for Windows. (2007, September 3). Retrieved September 14, 2007, from http://coq.inria.fr/V8.1/files/coq-8.1-win.exe.

The Coq standard library. (n.d.). Retrieved April 6, 2007, from http://coq.inria.fr/library-eng.html.

The Coq users' contributions. (2006, December 15). Retrieved February 27, 2007, from ftp://ftp.inria.fr/INRIA/coq/V8.0/contrib-8.0.tar.gz.

Cruz-Filipe, Luís, Geuvers, Herman, and Wiedijk, Freek. C-CoRN, the Constructive Coq Repository at Nijmegen. (n.d.). Retrieved January 16, 2007, from http://www.cs.math.ist.utl.pt/s84.www/cs/lcf/pubs/report3.pdf.

Cruz-Filipe, Luís. A Constructive Formalization of the Fundamental Theorem of Calculus. Retrieved December 25, 2006, from http://www.cs.ru.nl/~lcf/pubs/paper2.ps.

Cruz-Filipe, Luís. (2004, June 15). Constructive Real Analysis: a Type-Theoretical Formalization and Applications. (Doctoral dissertation, University of Nijmegen, 2004). Retrieved December 28, 2006, from http://www.cs.ru.nl/~lcf/pubs/phd.pdf.

Cruz-Filipe, Luís. Formalizing Real Calculus in Coq. (n.d.) Retrieved April 3, 2007, from http://slc.math.ist.utl.pt/lcf/pubs/report1.pdf.

Denney, Ewen. The Synthesis of a Java Card Tokenisation Algorithm. (2001, November). Retrieved July 2, 2007, from http://www.inf.ed.ac.uk/publications/online/0143.pdf.

Desmettre, Olivier. A formalization of real analysis in Coq. (2003, February 27). Retrieved June 23, 2007, from http://pauillac.inria.fr/~desmettr/publications/Reals.ps.

ExistsFromPropToSet. (2006, June 12). In *Cocorico!, the Coq Wiki*. Retrieved June 8, 2007, from http://cocorico.cs.ru.nl/coqwiki/ExistsFromPropToSet.

FAQ about Coq. (n.d.). Retrieved June 10, 2007, from http://coq.inria.fr/V8.1/faq.html.

Felty, Amy. Coq Session 1. (n.d.). Retrieved December 12, 2006, from http://www.site.uottawa.ca/~afelty/csi5110/CoqSession1.txt.

Felty, Amy. Coq Session 2. (n.d.). Retrieved December 12, 2006, from http://www.site.uottawa.ca/~afelty/csi5110/CoqSession2.txt.

Filliâtre, Jean-Christophe. Design of a proof assistant: Coq version 7. (2000, October). Retrieved November 17, 2007, from http://72.14.253.104/search?q=cache:oQO1KLG1pIEJ:www.lri.fr/~filliatr/ftp/publis/coqv7.ps.gz+%22Design+of+a+proof+assistant:+Coq+version+7%22&hl=en&ct=clnk&cd=3&gl=us.

Geuvers, Herman, Pollack, Randy, Wiedijk, Freek, and Zwanenburg, Jan. The algebraic hierarchy of the FTA project. (2001, June). *Proceedings of Calculemus 2001*, 13-27. Retrieved February 3, 2007, from http://www.calculemus.net/meetings/siena01/Papers/overall.ps.

Geuvers, Herman, Pollack, Randy, Wiedijk, Freek, and Zwanenburg, Jan. A Constructive Algebraic Hierarchy in Coq. (2002, October). Retrieved December 28, 2006, from http://www.cs.ru.nl/~freek/pubs/alghier1.pdf.

Geuvers, Herman, and Niqui, Milad. Constructive Reals in Coq: Axioms and Categoricity.

(2000). Retrieved December 25, 2006, from http://www.cs.ru.nl/~milad/publications/TYPES00.ps.

Geuvers, Herman, Wiedijk, Freek, and Zwanenburg, Jan. A Constructive Proof of the Fundamental Theorem of Algebra without using the Rationals. (2001). Retrieved November 17, 2007, from http://citeseer.ist.psu.edu/rd/27441992%2C466997%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/24588/http:zSzzSzwww.cs.kun.nlzSz%7EfreekzSznoteszSzkneser.pdf/geuvers01constructive.pdf.

Giménez, Eduardo. Co-Inductive Types in Coq: An Experiment with the Alternating Bit Protocol. (1995, June). Retrieved December 27, 2006, from ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1995/RR1995-38.ps.Z.

Gonthier, Georges. A computer-checked proof of the Four Colour Theorem. (n.d.). Retrieved March 31, 2007, from http://research.microsoft.com/~gonthier/4colproof.pdf.

Gonthier, Georges. Notations of the Four Colour Theorem proof. (n.d.). Retrieved March 31, 2007, from http://research.microsoft.com/~gonthier/4colnotations.pdf.

Group action. (2007, March 12). In *Wikipedia, The Free Encyclopedia*. Retrieved March 12, 2007, from http://en.wikipedia.org/wiki/Group_action.

Gupta, Maya R. A Measure Theory Tutorial (Measure Theory for Dummies). (2006, May). Retrieved June 24, 2007, from https://www.ee.washington.edu/techsite/papers/documents/UWEETR-2006-0008.pdf.

Hales, Thomas C. Introduction to the Flyspeck Project. (2005). Retrieved October 31, 2007, from http://drops.dagstuhl.de/opus/volltexte/2006/432/pdf/05021.HalesThomas.Paper.432.pdf.

Han, Seokhyun. Personal site. (n.d.). Retrieved June 23, 2007, from http://www.cs.rhul.ac.uk/~seokhyun/.

Har-Peled, Sariel. Lecture 8. (2004, February 18). Retrieved June 23, 2007, from http://valis.cs.uiuc.edu/~sariel/teach/2003/b_273/notes/08_prob_III.pdf.

Herbelin, Hugo. (2006, February 14). On the form of the axiom of description [Message-ID: 200602140912.KAA30896@pauillac.inria.fr]. Message posted to the Coq-club electronic mailing list, archived at http://pauillac.inria.fr/pipermail/coq-club/. Retrieved June 11, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2006.txt.

Huet, G., Kahn, G., and Paulin-Mohring, C. The Coq Proof Assistant: A Tutorial. (2004, April 27). Retrieved July 3, 2007, from http://flint.cs.yale.edu/cs428/coq/pdf/Tutorial.pdf.

Hurd, Joe. HOL Theorem Prover Case Study: Verifying Probablistic Algorithms. (2002). Retrieved June 23, 2007, from http://www.cl.cam.ac.uk/~jeh1004/research/talks/holprob-short.pdf.

Imura, Hiroshi, Kimura, Morishige, and Shidama, Yasunari. The Differentiable Functions on Normed Linear Spaces. (2004, May 24). Retrieved February 10, 2007, from http://www.cs.ualberta.ca/~piotr/Mizar/mirror/http/fm/2004-12/pdf12-3/ndiff_1.pdf.

Inclusion-exclusion principle. (2007, May 7). In *Wikipedia, The Free Encyclopedia*. Retrieved May 7, 2007, from http://en.wikipedia.org/wiki/Inclusion-exclusion_principle.

Index of MML Identifiers. (n.d.). Retrieved February 6, 2007, from http://mizar.org/JFM/mmlident.html.

Introduction to Group Theory. (n.d.). Retrieved March 12, 2007, from http://members.tripod.com/~dogschool/.

Invited talks. (n.d.). Retrieved June 23, 2007, from http://www.cs.ru.nl/lc2006/invited.html.

Karrmann, Stefan. (2005, October 27). Early versus late (non-)informative terms. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2005/002133.html.

Koprowski, A. (2007, March 21). Setoid for arbitrary relation? Message posted to http://www.nabble.com/Setoid-for-arbitrary-relation--t3441453.html. Retrieved April 3, 2007.

Kouba, Duane. The method of integration by parts. (2000, April 23). Retrieved February 22, 2007, from http://www.math.ucdavis.edu/~kouba/CalcTwoDIRECTORY/intbypartsdirectory/IntByParts.html.

Lebesgue integration. (2007, July 5). In *Wikipedia, The Free Encyclopedia*. Retrieved July 5, 2007, from http://en.wikipedia.org/wiki/Lebesgue_integration.

Lecture 1 – the first Mizar article. (2004, May 10). Retrieved October 5, 2006, from http://ysserve.int-univ.com/Lecture/MizarLecture/lecture1.pdf.

Letouzey, Pierre. (2005, February 21). Manipulating proof terms [Message-ID: Pine.LNX.4.44.0502211639260.30942-100000@pc8-142]. Message posted to the Coq-club electronic mailing list, archived at http://pauillac.inria.fr/pipermail/coq-club/. Retrieved June 11, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2005.txt.

Letouzey, Pierre. A New Extraction for Coq. (2003, January 22). Retrieved November 17, 2007, from http://www.pps.jussieu.fr/~letouzey/download/extraction2002.ps.gz.

Luo, Zhaohui. Annual site reports 2001 for the TYPES project. (2002, October 8). Retrieved June 23, 2007, from http://www.dur.ac.uk/TYPES/report-2001.ps.

MacQueen, David. Using Dependent Type to Express Modular Structure. (1985, October 30). Retrieved January 2, 2007, from http://www.cs.cmu.edu/~rwh/courses/modules/papers/

macqueen86/paper.pdf.

Mamane, Lionel Elie. (2007, January 2). sin_seq. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://mailman.science.ru.nl/pipermail/c-corn/2007-January/000051.html.

Maor, Eli. (1998). Trigonometric delights. New Jersey: Princeton University Press. Retrieved January 8, 2007, from http://press.princeton.edu/books/maor/chapter_10.pdf.

Marzocchi, M., Brand, H., and Edgar, G. A. (1997, October 27-30). Re: Question: Lebesgue Measurable but not Borel. Message posted to news://sci.math. Retrieved June 25, 2007, from http://www.math.niu.edu/~rusin/known-math/97/measure.

Matuszewski, Roman and Rudnicki, Piotr. Mizar: the first 30 years. (n.d.). Retrieved November 5, 2007, from http://www.cs.ualberta.ca/~piotr/Mizar/History/04MMA/M30.pdf.

McCarty, M., Leibel, S., Davis, D., Edgar, G. A., and Boden, J. (1999, May 7-8). Re: I am losing my math ability. Message posted to news://sci.math. Retrieved June 24, 2007, from http://www.math.niu.edu/~rusin/known-math/99/lebesgue.

The Mizar Mathematical Library, version 4.60.938. (2006, March 8). Retrieved January 31, 2007, from ftp://mizar.uwb.edu.pl/pub/system/i386-win32/mizar-7.6.02_4.60.938-i386-win32.exe.

The Mizar system, version 7.6.02, for Windows. (2006, March 8). Retrieved January 31, 2007, from ftp://mizar.uwb.edu.pl/pub/system/i386-win32/mizar-7.6.02_4.60.938-i386-win32.exe.

Mizar Verifier Basic Package (ver 7.8.05 mml 4.87.985). (n.d.). Retrieved September 29, 2006, from http://www.wakasato.org/mizar/s7.8.05m4.87.985/verifier1/cai-start.cgi.

MML Search. (2005, December 30). Retrieved September 29, 2006, from http://www.wakasato.org/mizar/s7.8.05m4.87.985/mml_search.php.

Monoid. (2007, May 16). In *Wikipedia, The Free Encyclopedia*. Retrieved May 16, 2007, from http://en.wikipedia.org/wiki/Monoid.

The most important facts in MML. (n.d.). Retrieved February 6, 2007, from http://merak.pb.bialystok.pl/mmlquery/fillin.php?filledfilename=mml-facts.mqt&argument=number+102.

mulhern@gmail.com. (2006, June 16). Equality modulo proofs. Message posted to the Coq-club electronic mailing list. Retrieved July 3, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2006/002403.html.

Muzalewski, Michał. An Outline of PC Mizar. (1999, August 13). Retrieved March 30, 2007, from http://www.cs.ru.nl/~freek/mizar/mizarmanual.ps.gz.

Nakamura, Yatsuka, Watanabe, Toshihiko, Tanaka, Yasushi, and Kawamoto, Pauline. Mizar Lecture Notes. (n.d.). Retrieved March 20, 2007, from http://markun.cs.shinshu-u.ac.jp/kiso/projects/proofchecker/mizar/Mizar4/printout/mizar4en_prn.doc.

Naumowicz, Adam. (2006, November 23). Re: Re: a term representing the extension of a type. Message posted to http://www.nabble.com. Retrieved March 29, 2007, from http://www.nabble.com/Re:-Re:-a-term-representing-the-extension-of-a-type-p7506530.html.

Naumowicz, Adam. (2007, February 17). Re: referring to unlabeled theorems. Message posted to http://www.nabble.com. Retrieved April 25, 2007, from http://www.nabble.com/Re:-referring-to-unlabeled-theorems-p9022084.html.

Naumowicz, Adam. (2007, February 22). Re: unused loci. Message posted to http://www.nabble.com. Retrieved March 28, 2007, from http://www.nabble.com/Re:-unused-loci-p9095762.html.

O'Connor, Russell. (2007, February 18). le_or_lt for CReals. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://mailman.science.ru.nl/pipermail/c-corn/2007-February/000059.html.

O'Connor, Russell. (2007, January 2). sin_seq. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://mailman.science.ru.nl/pipermail/c-corn/2007-January/000052.html.

O'Connor, Russell. (2007, January 19). sin x < x. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://mailman.science.ru.nl/pipermail/c-corn/2007-January/000056.html.

O'Connor, Russell. (2006, December 6). Using a Prop to bound recursion [Message-ID: Pine.LNX.4.64.0612060242110.25142@erdos.theorem.ca]. Message posted to the Coq-club electronic mailing list, archived at http://pauillac.inria.fr/pipermail/coq-club/. Retrieved June 8, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2006.txt.

Oliver, M., Taylor, B., Grambsch, P., and Renfro, D. L. (1999, March 4). Re: Borel-sets. Message posted to news://sci.math. Retrieved June 25, 2007, from http://www.math.niu.edu/~rusin/known-math/97/measure.

Paulin, Christine. (1997, January 14). Re: Problems with exist [Message-Id: 199701140842.JAA11521@aquavit.ens-lyon.fr]. Message posted to the Coq-club electronic mailing list, archived at http://www.iist.unu.edu/~alumni/software/other/inria/www/coq/mailing-lists/coqclub/. Retrieved July 3, 2007, from http://www.iist.unu.edu/~alumni/software/other/inria/www/coq/mailing-lists/coqclub/0168.html.

Paulin-Mohring, Christine. A library for reasoning on randomized algorithms in Coq. (2007, May 30). Retrieved June 23, 2007, from http://www.lri.fr/~paulin/ALEA/library.pdf.

Plume. (n.d.). Retrieved June 23, 2007, from http://www.ens-lyon.fr/LIP/PLUME/.

Pollack, Robert. Dependently Typed Records in Type Theory. (2002, February 5). Retrieved January 2, 2007, from http://homepages.inf.ed.ac.uk/rpollack/export/recordsFAC.ps.gz.

Pottier, Loïc. Basic notions of algebra. (1999, March). Retrieved May 9, 2007, from http://coq.inria.fr/contribs/algebra.tar.gz.

Probability Tutorial. (n.d.). Retrieved June 24, 2007, from http://tutors4you.com/probabilitytutorial.htm.

Proofs from THE BOOK. (2007, October 19). In *Wikipedia, The Free Encyclopedia*. Retrieved October 29, 2007, from http://en.wikipedia.org/wiki/Proofs_from_THE_BOOK.

Raamsdonk, Femke van. Inductive types. (n.d.). Retrieved December 11, 2006, from http://www.cs.vu.nl/~femke/courses/lv/notes/week04.pdf.

Raamsdonk, Femke van. Logical verification 06-07 practical work week 1. (n.d.). Retrieved December 11, 2006, from http://www.cs.vu.nl/~femke/courses/lv/prac/pw01_answers.v.

Raamsdonk, Femke van. Logical verification 06-07 practical work week 2. (n.d.). Retrieved December 11, 2006, from http://www.cs.vu.nl/~femke/courses/lv/prac/pw02_answers.v.

Raamsdonk, Femke van. Logical verification 06-07 practical work week 3. (n.d.). Retrieved December 11, 2006, from http://www.cs.vu.nl/~femke/courses/lv/prac/pw03_answers.v.

Raamsdonk, Femke van. Logical verification 06-07 practical work week 4. (n.d.). Retrieved December 11, 2006, from http://www.cs.vu.nl/~femke/courses/lv/prac/pw04_answers.v.

Renfro, D. L. and Rubin, H. (1999, August 27). Re: nonmeasurable sets and non_Borel sets in R. Message posted to news://sci.math. Retrieved June 25, 2007, from http://www.math.niu.edu/~rusin/known-math/97/measure.

Richter, Stefan. Formalizing Integration Theory, with an Application to Probabilistic Algorithms. (2005, October 14). Retrieved June 25, 2007, from http://afp.sourceforge.net/browser_info/current/HOL/HOL-Complex/Integration/outline.pdf.

Rideau, Laurence, and Théry, Laurent. Formalising Sylow's Theorems in Coq. (2006, November 22). Retrieved March 31, 2007, from http://hal.inria.fr/docs/00/11/56/32/PDF/RT-0327.pdf.

Riemann integral. (2007, July 4). In *Wikipedia, The Free Encyclopedia*. Retrieved July 4, 2007, from http://en.wikipedia.org/wiki/Riemann_integral.

Rudnicki, Piotr. A Mizar demo. (1997, March 13). Retrieved September 29, 2006, from http://www.cs.ualberta.ca/~piotr/Mizar/Dagstuhl97/.

Rusin, Dave. Measure and integration. (2000, January 24). In *The Mathematical Atlas*. Retrieved June 24, 2007, from http://www.math.niu.edu/~rusin/known-math/index/28-XX.html.

Setoid. (2007, March 22). In *Wikipedia, The Free Encyclopedia*. Retrieved April 6, 2007, from http://en.wikipedia.org/wiki/Setoid.

Shmyrev, Nickolay V. (2007, January 19). sin x < x. Message posted to the Coq-club electronic mailing list. Retrieved November 5, 2007, from http://mailman.science.ru.nl/pipermail/c-corn/2007-January/000055.html.

Simpson, Carlos. Set-theoretical mathematics in Coq. (2004, February 20). Retrieved April 3, 2007, from http://arxiv.org/pdf/math.LO/0402336.pdf.

Smith, Geoff. The Inclusion Exclusion Counting Principle. (1998, February). Retrieved May 7, 2007, from http://people.bath.ac.uk/masgcs/book1/amplifications/inc_exc.pdf.

Sozeau, Matthieu. Subset coercions in Coq. (2006, April). Retrieved July 3, 2007, from http://www.lri.fr/~sozeau/research/russell/article.pdf.

Stein, Jasper. Linear algebra. (2003, September 19). Retrieved May 12, 2007, from http://coq.inria.fr/contribs/LinAlg.tar.gz.

Stump, Aaron. sni_sec. (2005). Retrieved July 13, 2007, from http://cl.cse.wustl.edu/classes/cse545/lec/sni.v.

Urban, Josef. (2002, July 25). Error Number: 129. In *Mizar TWiki*. Retrieved March 30, 2007, from http://wiki.mizar.org/cgi-bin/twiki/view/Mizar/ErrorNo129.

van Oosten, Jaap. Realizability: An Historical Essay. (2000, December 29). Retrieved November 17, 2007, from http://www.math.uu.nl/people/jvoosten/realizability/history.ps.gz.

Viltersten, K., gowan4@hotmail.com, Santos, J. C., and Jules. (2006, January 20). What is a Borel set? Message posted to news://sci.math. Retrieved June 29, 2007, from http://groups.google.com/group/sci.math/browse_thread/thread/d3160c9928ec1b42/77352d6bbf0bfe00?lnk=st&q=borel+set+real&rnum=6#77352d6bbf0bfe00.

Virk, Rahbar. The Orbit-Stabilizer Theorem. (n.d.). Retrieved March 12, 2007, from http://www.math.wisc.edu/~virk/notes/pdf/orphans/orbit-stabilizer_thm.pdf.

Wenzel, Markus, and Wiedijk, Freek. A Comparison of the Mathematical Proof Languages Mizar and Isar. (2002). Retrieved February 6, 2007, from http://www4.in.tum.de/~wenzelm/papers/romantic.pdf.

Werner, Benjamin. Sets in Types, Types in Sets. (2007, March 6). Retrieved June 8, 2007, from http://www.lix.polytechnique.fr/Labo/Benjamin.Werner//publis/tacs97.pdf.

Werner, Benjamin, Paulin, Christine, dowek@pomerol.inria.fr, Despeyroux, Joelle, and Kahn, Gilles. (1993, August 31). Various answers to Gilles Kahn's question about Sets in Coq. Messages posted to the Coq-club electronic mailing list. Retrieved July 3, 2007, from http://www.iist.unu.edu/~alumni/software/other/inria/www/coq/mailing-lists/coqclub/0018.html.

Weisstein, Eric W. (2002, June 24). Fundamental Theorems of Calculus. From *MathWorld*--A Wolfram Web Resource. Retrieved November 1, 2007, from http://mathworld.wolfram.com/FundamentalTheoremsofCalculus.html.

Weisstein, Eric W. (2003, September 2). Inclusion-Exclusion Principle. From *MathWorld*--A Wolfram Web Resource. Retrieved May 7, 2007, from http://mathworld.wolfram.com/Inclusion-ExclusionPrinciple.html.

Wiedijk, Freek. (2007, May 7). Formalizing 100 Theorems. Retrieved May 7, 2007, from http://www.cs.ru.nl/~freek/100/.

Wiedijk, Freek. Mizar: An Impression. (1999, September 27). Retrieved March 20, 2007, from http://www.cs.ru.nl/~freek/mizar/mizarintro.ps.gz.

Wiedijk, Freek (Ed.). The Seventeen Provers of the World. (2006a, March). New York: Springer-Verlag. Retrieved February 6, 2007, from http://www.cs.ru.nl/~freek/comparison/comparison.pdf.

Wiedijk, Freek. Writing a Mizar article in nine easy steps. (2006b). Retrieved March 14, 2006, from http://www.cs.ru.nl/~freek/mizar/mizman.pdf.

Wilde, Ivan F. Measure, integration & probability. (2006, February 2). Retrieved July 1, 2007, from http://www.mth.kcl.ac.uk/~iwilde/notes/mip/mip.pdf.

Zanella, Santiago. (2007, March 21). Re: Setoid for arbitrary relation? Message posted to http://www.nabble.com. Retrieved April 3, 2007, from http://www.nabble.com/Re:-Setoid-for-arbitrary-relation--p9668747.html.

Zumkeller, Roland. (2006, November 23). Newbie questions [Message-ID: d02dcb040611231401r47ed3f46j2d19b565356f9e5f@mail.gmail.com]. Message posted to the Coq-club electronic mailing list, archived at http://pauillac.inria.fr/pipermail/coq-club/. Retrieved June 11, 2007, from http://pauillac.inria.fr/pipermail/coq-club/2006.txt.