The Book

Assia, Enrico, . . . (you are welcome)

November 28, 2014

Contents

Chapter 0

The essence of math comp

This introduction explains the motivation of the book, which is to present the methodologies we propose to build libraries of formalized mathematics *that scale and can be reused.* Ideally, the content of the section is organized as a drawn-out comparison with the way mathematics on paper/blackboards are developed and written. Then the message is that in order to fulfill the technical needs, computer science techniques can be used and have worked well.

For instance, trying to formalize mathematical results from scratch in an empty context, without libraries and only with the tools proposed by the logic and the proof assistant is like trying to learn/invent mathematics without the help of the way mathematical concepts and notations have been shaped, structured, and denoted during the course of their study. In particular, this might lead the user to let the proof assistant impose a certain, usually more pedestrian proof, which we want to prevent as much as possible.

There are several important issues that have to be solved in order to come up with a library that has a chance to be reusable. The most pervasive one is the line drawn between what is trivial and what is logged in the proofs (for humans). Some are obvious (reflexively of relations, ...) but in general this should be analyzed by ... learning the maths and reading the literature in the domain of interest. This appreciation might also be depending on the knowledge the reader has: in 1st year texts you make dozen of proofs that certain sets are equipped with a structure of group, vector space or whatever, but very soon you're supposed to *see* that it is obviously the case. Emphasizing this point and providing a methodology to implement this in a systematic way might be one of the original points of this document.

One of the great achievements attributed to Bourbaki is the generalization of the so-called axiomatic method, that promotes the design and use of abstract mathematical structures that factor theory and notations. This not only make maths more readable, but also more understandable and you need to play with the concepts for some times to come up with the right abstractions. Incidentally, notations are not only overloading of symbols, but also carry inference of properties. We should be able to do this and find a way to capture this inference, which is fact a (Prolog like) program run by the trained reader. And in general we would like the user to provide as much, but no more information in his statement/proof as he would in a proof, avoiding redundant information (where redundant covers inferable).

Another kind of implicit is behind reasoning patterns that should also be modeled in a convenient way for the user. For instance, *mutatis mutandis*, wlog, etc. These rely on the fact that the reader is able both to infer the mathematical statement involved in a cut formula and to run rather easily the simple piece of proof that justify it. This is modeled by techniques based on formula generation by subterm selection. This also overcome an unwanted behaviors that accessing to a subterm depends on the depth at which it occurs in the formula (usual bureaucracy in naive formalization).

The techniques that make this possible are adapted from standard methods in computer science/programming languages. We stand on the shoulders of a foundational system based on a type theory, which promotes functions as the initial concept of the formalism (as opposed to sets). In particular computations plays a privileged role there, allowing to model a notion of “similar” up to (implicit) computations. Things that are equal up to computations can be substituted one by the other in a transparent way. This is central to the way things are modeled and at the heart of (small) scale reflection. Computer science engineering techniques are also useful to organize the infrastructure content that make possible to work with some comfort in upper layers of the library, in order to configured the content so that the machine can use it (order of arguments, implicit status, naming policy...).

0.1 challenges faced and tools adopted

(tools in a broad sense, the logic is a tool, coq is a tool, the plugin is a tool, the ssr style is a tool,...)

Challenges:

- large body (scale up), make proofs small and robust. We need to say that we do use ”deterministic automation”. Use Laurent’s data on de bruijn factor.

- model the use of math notations, their role in proofs, model proofs (also it is about reasoning, not just computations). Another way to say that is: model Bourbaki (rationalization of Math via structure/interfaces) but not the first book (set theory) that is replaced by CIC (link with section computational thinking).

We build on Coq and an extension. The main tools follow (in random order):

0.2 Computational Thinking

This section should motive the activity of formalizing mathematics with the **Coq** proof assistant, emphasizing its computing skills, and as opposed to other foundations like HOL. However the challenge is to keep mathematicians as the privileged target, while motivating the ssr approach to a CS oriented reader.

See `../coq/ch0.v`.

Aim of the chapter:

- should sound natural and easy to a CS person (but with the ssr twist)

- should sound different but well motivated to a Coq user (do show, maybe in the exercises, that leqn is 100 times better than ”Inductive le”). Try to reproduce the shock we had the first time we used Boolean predicates. It may help to compare, in the *advanced* section, the approach with the standard one, so that one sees two proof scripts in the same page.

0.3 logic programming ... type inference

to model proof search, give a meaning to notations, teach coq the work an informed reader does (contextualizing otherwise ambiguous notations, knowledge of interface/instance of algebraic structures).

relation with proof search: no ”blind” proof search (easy, ad-hoc, pervasive v.s. advanced, generalistic, potentially expensive and unstable).

0.4 automation in tactics

the main points:

- 1/3 is rewrite, term selection/search (one does not need to reach a sub formula as a goal in order to make progress for example, no monkey puzzle as in GG terminology).

- Create a formula without writing it: some advanced forms of forward reasoning tactics to deal with symmetries, generalizations (boils down to syntesize the cut formula out of the minimum possible user input, as in a text where one says ”similarly to that, we can also prove that”. (this is not very pervasive, dunno if it is worth putting it here in this chapter). Also elim does that. (technically also rewrite, but we may want to separate things)

This section is were one talks about the plugin, and some of the main design points of tactics: compositional (a language, not a list of commands), predictable (documented!!!), finally compact (symbols for uninteresting steps).

0.5 discipline

MAKE an howto out of that. Maybe one should also add a few notes on the style in scripts? like:

- you must be able to model (at least) 1 proof step in a sentence (line), e.g. ”rewrite preparegoal dostep ?cleanup.”
- uninteresting/recurrent lemmas/steps should be small (short names, easy to gray out)
- lemma statements are designed, not just written, having in mind their use (forward, backward, implicit argumets, arguments order) and the class of trivial hypotheses since an extra hyp that is proable triviality (via //, hint resolve, cnaonical) is for free. E.g. ”x
is a toto”, ”0 ¡= n”, ...
- also not every possible lemma, but a few that combine well
- proofs/definitions are reworked many times, why (understand recurrent proof schemas, compact, factor, make more stable/robust) and what is needed (like meaningful names, clear structure)

0.6 trivial=implicit (for a trained mathematician)

The idea is to try to identify what is trivial (mathematically speaking) and be sure you can model it as such:

- (level basic) make explicit the trivialities of each theory (what one expects to be proved by //).
- (level advanced) when you do new stuff, you must decide what is trivial/implicitly proved.
- (hard) which technique to make Coq prove it automatically (hint resolve, canon, comput... in the type)

This may also be another way present the whole chapter

mantra: (basic) if you see toto=false you should perform the case analysys via fooP

Part I

The art of formalizing

Chapter 1

Logics

From calculability to proofs, hence the CC, and the fact that reasoning principles without a computational content become axioms.

This is a non technical chapter and message should be:

- instantiation of a universal statement is application (also the pair)
- Excluded middle is not available by default (choice?)
- Conversion as a pervasive indistinguishably, what inside (beta, definition unfolding,...)
- Dependent types: eq, sigma (which example?)

One options is: avoid relating type theory and other logics. We say: we have a formal game where the basic elements are programs/functions that come with types to avoid confusion. full stop. (no relation with proof theory, set theory). maybe mention that roots are in calculability (hence the choice to pick functions as primitive and not sets). This is lucky because (computable) functions are today executable by a computer. Still not all concepts are ”computable” hence some principles are problematic: EM,.... we mainly stay in the lucky fragment (again no propaganda on intuitionistic logic, constructive math; just a mention).

Chapter 2

Programming

Presentation of inductive data structures, recursive programs on these data. Bool, Boolean connectives, Boolean reflection (cf ch. 0.2), views. Examples of equality tests (==, with a forward reference to explain the magic if needed), operations on sequences, nats, exercises on prime, div.

Chapter 3

Proofs

Where one learns to do proofs. Boolean reflection in practice, views, discussion on the definition of leq, proofs on things defined in the previous chapter, associated tactics, exercises on prime, div, binomial, etc.

spec? A new vernacular to declare specs without typing coinductive and by writing explicitly the equations.

Chapter 4

Type inference/Genericity

put here a minimalistic presentation of records.

Implicit arguments (Is it possible to survive without implicit argument up to this point? We should probably use them plus forward references to here), canonical structures. Combining with notations. Now the real ==.

Chapter 5

Mixing data and proofs

Boolean sigma types, records, coercions, UIP. Examples: ordinals, tuples (not their use which requires CS). This chapter might come after some presentation of basic canonical structures, i.e. reorganize the content between this chapter and the next.

example of tuples here? better ordinals?

GG: makes many points here (not fully understood by Enrico):

- ordinals/tuples are easy to use but hard to build
- it is a tradeoff, but is not clear if we can give hints on when a specific datatype like ordinals is better that unpackaged stuff.

UIP is advanced, the basic user should just be told that putting bools inside a record is just fine.

- record (flat)

Chapter 6

Hierarchy

Packaging records, the bigop hierarchy. Scaling with packed classes and mixins, to the ssralg hierarchy. Presentation of the content of ssralg in terms of structures and of the theory? Should the latter be a separate chapter.

Maybe a plugin for a new vernacular to script the creation/declaration of structures/instances so that the level basic can touch the argument easily.

Explain what the abstraction barrier is (like unfolding a GRing projection)

$\boxed{\text{gotcha: if you see GRing.toto then you broke an abstraction barrier}}$

$\boxed{\text{mantra: if you have a proof in mind, don't let the system drive you to another, less clean and abstract, proof.}}$

Declaring an instance is hard... we need to document the multuiple processes for each structure in each hierarchy and possibily make a program out of it.

Chapter 7

Larger scale reflection (out of place)

Four colours, decomposition in primes, example in peterfalvi. In particular example where a case analysis does not follow the constructors of an existing inductive type: then craft an ad hoc one to hint the proof. This topic is probably at a wrong place.

Part II

Mathematics in Mathematical Components

7.1 STYLE of these chapters

These chapters hopefully in the following style:

- first math mode to review the (not so standard) definitions and
- then touch some real proof script to show why/how the definition make it possible to model the proofs (as they are in math)
- use the (few) examples to illustrate a cool proof strategy if any (not necessarily typical of the math subject, but that happens to be there, like in OOTHM paper: circular leq, symmetries, ad hoc decision procedure, ...)
- try hard to show how CIC helps (which feature: HO, computation, dependent types ...), so that at the end we can sum up and make a synthesis of all that.

Maybe it is simpler to do it in 2 steps: 1) in this second part one identifies where CIC or the SSR style plays a crucial role 2) then we advertise these use cases in the first part to motivate the techniques, the complexity of the logic...

Chapter 8

Numbers

What are the numbers available in the library? How to use them, casts between types... Non trivial point in the formalisation: axiomatization, order is defined from norm, partial orders (in particular for complex numbers).

Chapter 9

Polynomials, Linear algebra

2-stage presentation: interface plus explicit. Expansion of Georges'ITP paper. Here is one of the main application of the choice operator (complement a base).

Chapter 10

Quotients

Chapter 11

Finite group theory

Data-structures, how to craft the set of variants of a same theorem to make the formalization handy. Permutations. Presentations?

Chapter 12

Representation theory, Character theory

As an example of application, in particular of the linear algebra theory.

Chapter 13

Galois Theory

Chapter 14

Real closed fields

Chapter 15

Algebraic closure

Part III

Conclusion and perspectives

Let's be brave: This part looks back to the techniques, methodologies and achieved formalized libraries described in the book and summarizes the role played by each on them, in the spirit of the introduction to the ssr manual. May be also discusses the possible extensions (like CoqEAL) or adaptation to the future evolutions of the system and formalism (Cubical Coq, HITs...).

Part IV

Annexes

Chapter 16

What is done where?

Chapter 17

How tos

suggestions:

- Get more information when you do not understand the error message
- Search in the library
- Canonical structures: define a new instance
- Canonical structures: add a new structure
- Give a relevant name to the lemma I just stated
- Forbid unwanted expansions
- Choose a notation (what not to do...)
- Compute “for real” (Natbin, Coqeal)
- MathComp script style

Chapter 18

Naming conventions

Chapter 19

Index of notations