The Book

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

December 4, 2014

Contents

Chapter 0

Mathematical Components

the library was originally developed to support the formalization of the oothm, a result that requires a wide panel of math theories.

0.1 challenges

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 (centuries long) 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. (Another way to explain this could be to imagine how to explain some math to a kid that did receive no education, or trying to imagine that kid inventing new math... the cultural backround is important to be up to speed. Maybe the aim of mathcomp is to implement that background at the quality of today's math).

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

Lost in the ...

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...).

0.2 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):

0.3 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.

0.4 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).

0.5 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).

0.6 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)

0.7 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. 0.3), 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

Learning Coq *how to read Math (step 1)*

The rules of the Calculus of Inductive Constructions are expressed on the syntax on terms and are implemented by the kernel of Coq. Such software component performs *type checking*: given a term and type it checks if such term has the given type. Terms are very verbose since a lot of information has to be made explicit in order to make type checking decidable.

Luckily the user very rarely interacts directly with the kernel. Instead she almost always interact with the refiner, a software component that is able to accept open terms. Open terms are in general way smaller than regular terms because some information is left implicit. In particular one can omit any subterm by writing "_" in place of it. Each missing piece of information is either reconstructed automatically by the *type inference* algorithm, or provided interactive later by means of proof commands. In this chapter we focus on type inference.

I'm a bit uneasy about citations, here I think I want to add one[**?**]. They are good readings but a arbitrary and not easy to find. We should define a policy for citations.

Type inference is *ubiquitous*: whenever the user inputs a term (or a type) the system tries to infer a type (or a sort) for it. One can think of the work of the type inference algorithm as trying to give a meaning to the input of the user possibly completing and constraining it by inferring some information. If the algorithm succeeds the term is accepted, otherwise an error is given.

What is crucial to the Mathematical Components library is that the type inference algorithm is *programmable*: one can extend the basic algorithm with small declarative programs that have access to the library of already formalized facts. In this way one can make the type inference algorithm aware of the contents of the library and make Coq behave as a trained reader that is able to guess the intended meaning of a mathematical expressions from the context and thanks to his background knowledge.

Introducing the reader to the type inference algorithm and helping her to make good use of it is the ultimate goal of this chapter.

4.1 Type inference and Higher Order unification

Learns HO unif is hard
Requires
Provides terminology
Level 1

The type inference algorithm is quite similar to the type checking one: it recursively traverses a term checking that each subterm has a type compatible with the type expected by its context. During type checking types are compared taking computation into account. Terms that are compared as equal are said to be *convertible*. Termination of reduction and uniqueness of normal forms provide guidance for implementing the convertibility test, for which a complete and sound algorithm indeed exists. Unfortunately type inference works on open terms, and this turns convertibility into a much harder problem called *higher order unification*. The special placeholder "`_`", usually called *implicit argument*, may occur inside types and stands for one, and only one, term that is not explicitly given. Type inference does not check if two types are convertible, it checks if they unify. Unification is allowed to assigning values to implicit arguments in order to make the resulting terms convertible. For example unification is expected to solve the unification problem comparing `(list _)` with `(list nat)` by choosing the value `nat` for the placeholder. Thanks to this assignment the two types become trivially convertible.

Unfortunately it is not hard to come up with classes of examples where guessing appropriate values for implicit arguments is, in general, not possible. In fact such guessing has be shown to be as hard as proof search in presence of higher order constructs. For example to unify `(prime _)` with `true` one has to guess a prime number. Remember that `prime` is a boolean function that fed with a natural numbers returns either `true` or `false`. While `2` would be a perfectly valid solution, it is clear that it is not the only one. Notice that enumerating all possible values until one finds a valid one is not a general solution, since the good value may not exist. Just think at the problem `(prime (4 * _))` versus `true`. An even harder class of problems is the one of synthesizing programs. Take for example the unification problem `(_ 17)` versus `[:: 17]`. Is the function we are looking for the list constructor? Or maybe, is it a factorization algorithm?

Given that there is no silver bullet for higher order unification Coq makes a sensible design choice: provide an (almost) heuristic-free algorithm and let the user extend it via an extension language. We will refer to such language as the language of *Canonical Structures*. Despite being a very restrictive language, it sufficient to let one program a wide panel of very useful functionalities. The one described in this chapter is notation overloading.

The concrete syntax for implicit arguments, an underscore character, does not let one name the missing piece of information[1]. If an expression contains multiple occurrence of the placeholder "`_`" they are all considered as potentially different by the system, and hence hold (internally) unique names. For the sake of clarity we will take the freedom to use the alternative syntax $?_x$ for implicit arguments (where x is a unique name).

[1] This may change in Coq 8.5

4.2 Type inference at work

Learns type and term inference
Requires have, move, exact
Provides Arguments (setting implicit)
Level 1

Lets start with the simplest example: defining the polymorphic identity function and checking its application to `3`.

Polymorphic identity

```
1 Definition id (A : Type) (a : A) : A := a.
2 Check (id nat 3).
3 Check (id _ 3).
```

Response

```
id nat 3 : nat
id nat 3 : nat
```

In the expression `(id nat 3)` no subterm was omitted, and indeed Coq accepted the term and printed its type. In the third line even if the sub term `nat` was omitted, Coq accepted the term. Type inference found a value for the place holder for us by proceeding in the following way: it traversed the term recursively from left to right, ensuring that the type of each argument of the application had the type expected by the function. In particular `id` expects two arguments. The former argument is expected to have type `Type` and the user left such argument implicit (we name it $?_t$). Type inference imposes that $?_t$ has type `Type`, and this constraint is satisfiable. The algorithm continues checking the arguments. According to the definition of `id` the type of the second argument must be the value of the first argument. Hence type inference runs recursively on the argument `3` discovering it has type `nat` and imposes that it unifies with the value of the first argument (that is $?_t$). For this to be true $?_t$ has to be assigned the value `nat`. As a result the system prints the input term, where the place holder has been replaced by the value type inference assigned to it.

At the light of that we see that every time we apply the identity function to a term we can omit to specify its first argument, since Coq is able to infer it and complete the input term for us. This phenomenon is so frequent that one can ask the system to insert the right number of `_` for him. For more details see Section 4.5 or refer to the user manual. Here we only provide a simple example.

Setting implicit arguments

```
1 Arguments id {A} a.
2 Check (id 3).
3 Check (@id nat 3).
```

Response

```
id 3 : nat
id 3 : nat
```

The `Arguments` directive "documents" the constant `id`. In this case it just marks then argument that has to be marked as implicit by surrounding it with curly braces. The declaration of implicit arguments can be locally disabled by prefixing the name of the constant with `@`.

Another piece of information that if often left implicit is the type of abstracted variables.

Omitting type annotations

```
1  Check (fun x => @id nat x).
2
3  Lemma prime_gt1 p : prime p → 1 < p.
```

Response

```
fun x : nat => id x :
  nat → nat
```

In the first line the syntax (`fun x => ...`) is sugar for `(fun x : _ => ...)` where we leave the type of `x` open. Type inference fixes it to `nat` when it reaches the last argument of the identity function. It unifies the type of `x` with the value of the first argument given to `id` that in this case is `nat`. This last example is emblematic: most of the times the type of abstracted variables can be inferred by looking at how they are used. This is very common in lemma statements. For example the third line states a theorem on `p` without explicitly giving its type. Since the statement uses `p` as the argument of the `prime` predicate, it is automatically constrained to be of type `nat`.

The kind of information filled in by type inference can also be of another, more interesting, nature. So far all place holders were standing for types, but an `_` can also be used in place of a term.

Inferring a term

```
1  Lemma example q : prime q → 0 < q.
2  Proof.
3  move=> pr_q.
4  have q_gt1 := prime_gt1 _ pr_q.
5  exact: ltnW q_gt1.
6  Qed.
```

Goal after line 3

```
q : nat
pr_q : prime q
==================
0 < q
```

The proof begins by giving the name `pr_q` to the assumption `prime q`. Then it builds a proof term by hand using the lemma stated in the previous example and names it `q_gt1`. In the expression `(prime_gt1 _ pr_q)` the place holder, that we name $?_p$, stands for a natural number. When type inference reaches $?_p$ it fixes its type to `nat`. What is more interesting is what happens when type inference reaches the `pr_q` term. Such term has its type fixed by the context: `(prime q)`. The type of the second argument expected by `prime_gt1` is `(prime` $?_p$`)` (i.e. the type of `prime_gt1` were we substitute $?_p$ for `p`. Unifying `(prime` $?_p$`)` with `(prime q)` is possible by assigning `q` to $?_p$. Hence the proof term just constructed is well typed, its type is `(1 < q)` and the place holder has been set to be `q`. As we did for the identity function we can declare the `p` argument of `prime_gt1` as implicit. . Declaring implicit arguments of lemmas is be tricky and requires one to think ahead how the lemma will be used. Section 4.5 is dedicated to it.

maybe also tell why one does not need two underscores in the last line

So far type inference and in particular unification has been used in its simplest form, and indeed a first order unification algorithm incapable of computing or synthesizing functions would have sufficed. In the next section we introduce the encoding of the relations that will be at the base of the declarative, logic, programs we will use to extend the unification in the higher order case.

As of today there is not precise documentation of the type inference and unification algorithms implemented in Coq. For a technical presentation of

a type inference algorithm close enough to the one of Coq we suggest the interested reader to consult [**?**]. The reader interested in a technical presentation of a simplified version of the unification algorithm implemented in Coq can read [**?**, **?**].

4.3 Records as first class relations

Learns records as relations, canonical base instances
Requires
Provides Canonical
Level 1

In computes science a record is a very common data structure. It is a compound data type, a container with named fields. Records are represented faithfully in the Calculus of Inductive Constructions as inductive data types with just one constructor holding all the data. The peculiarity of the records we are going to use is that they are dependently typed: the type of each field is allowed to depend on the values of the fields that precedes it.

Coq provides syntactic sugar for declaring record types:

```
Record eqType : Type := Pack {
  sort : Type;
  eq_op : sort → sort → bool
}.
```

The sentence above declares a new inductive type called `eqType` with one constructor named `Pack` with two arguments. The first one is named `sort` and holds a type; the second and last one is called `eq_op` and holds a comparison function on terms of type `sort`. What this special syntax does is declaring at once the following inductive type plus a named projection for each record field:

```
Inductive eqType : Type :=
  Pack sort of sort → sort → bool.
Definition sort (c : eqType) : Type :=
  let: Pack t _ := c in t.
Definition eq_op (c : eqType) : sort c → sort c → bool :=
  let: Pack _ f := c in f.
```

Note that the first projection is used in order to define the type of the second projection.

We think of the `eqType` data type as a relation linking a data type with a comparison function on that data type. Before putting the `eqType` relation to good use we declare an inhabitant of such type, that we call an *instance*, and we examine a crucial property of the two projections just defined.

We relate the following comparison function with the `nat` data type:

Maybe this function has been shown already

```
Fixpoint eqn m n {struct m} :=
 match m, n with
  | 0, 0  true
  | m'.+1, n'.+1  eqn m' n'
  | _, _  false
  end.
Definition nat_eqType : eqType := Pack nat eqn.
```

Projections, when applied to a record instance like `nat_eqType` compute and extract the desired component.

Computation of projections	Response
`Eval simpl in sort nat_eqType.`	`= nat`
`Eval simpl in eq_op nat_eqType.`	`= eqn`

Maybe simpl is already explained?

Given that `(sort nat_eqType)` and `nat` are convertible, we can use the two terms interchangeably. The same holds for `(eq_op nat_eqType)` and `eqn`. Thanks to these facts Coq can type check the following term:

```
Check (eq_op nat_eqType 3 4).
```

```
eq_op nat_eqType 3 4 : bool
```

This term is well typed, but checking it is not as simple as one may expect. The `eq_op` function is applied to three arguments. The first one is `nat_eqType` . The following ones are expected of to be of type `(sort nat_eqType)` but `3` and `4` are of type `nat`. Recall that unification takes computation into account exactly as the convertibility relation. In this case the unification algorithm unfolds the definition of `nat_eqType` obtaining `(sort (Pack nat eqn))` and reduces the projection extracting `nat`. The obtained term literally matches the type of the last two arguments given to `eq_op`.

Now, why this complication? Why should one prefer `(eq_op nat_eqType 3 4)` to `(eqn 3 4)`? The answer is *overloading*. It is recurrent in mathematics and computer science to reuse a symbol, a notation, in two different contexts. A typical example coming from the mathematical practice is to use the same infix symbol $*$ to denote any ring multiplication. A typical computer science example is the use of the same infix `==` symbol to denote the comparison over any data type. Of course the underlying operation one intends to use depends on the values it is applied to, or better their type[2]. Using records as relations will let us model these practices. In the rest of this chapter we will focus on the overloading of the `==` symbol.

To model overloading we first have to define another comparison function:

[2] Actually the meaning of a symbol in math is even deeper: by writing $a * b$ one expects the reader to figure out from the context which ring we are talking about, recall its theory, and use this knowledge to eventually justify the steps that will follow in a proof. This very same approach let us also model this practice, but we discuss it only in the next chapter

```
Definition eqb (a b : bool) := if a then b else (not b).
Definition bool_eqType : eqType := Pack bool eqb.
```

I need the reader to know something about Notation

Now we can define a notation that applies to any occurrence of `eq_op`.

Overloaded notation

```
Notation "x == y" := (eq_op _ x y).
Check (eq_op bool_eqType true false).
Check (eq_op nat_eqType 3 4).
```

Response

```
true == false : bool
3 == 4 : bool
```

The notation is used by Coq for both printing and parsing.

As a printing rule, the place holder stands for a wild card: the notation will be used no matter the value of the first argument of `eq_op`. As a result both occurrences of `eq_op` are printed with the infix `==` symbol. Of course the two operations are different, they are specific to the type of the arguments and the typing we saw above ensures the arguments match the type of the function.

When interpreted as a parsing rule, the place holder is interpreted as a itself: type inference is expected to find a value for it. Unfortunately such notation does not work as a parsing rule yet.

Error

```
Check (3 == 4).

```

Response

```
Error: complete this
with the real error
```

If we unravel the notation the input term is really `(eq_op _ 3 4)`. We name the placeholder $?_e$. If we replay the type inference steps seen before, the unification step is now failing. Instead of `(sort nat_eqType)` versus `nat`, now unification has to solve the problem `(sort` $?_e$`)` versus `nat`.

This problem falls in one of the problematic classes we presented in Section 4.1. Coq gives up, leaving to the user the task of extending the algorithm with a program able to solve unification problems of the form `(sort` $?_e$`)` versus `T` for any `T`. Given the current context it seems reasonable to use write an extension that picks `nat_eqType` when `T` is `nat` and `bool_eqType` when `T` is `bool`. In the language of Canonical Structures such program is expressed as follows.

Declaring Canonical Structures

```
Canonical nat_eqType.
Canonical bool_eqType.
```

The keyword `Canonical` was chosen to stress that the program is deterministic: each type `T` will be related to (at most) one *canonical* comparison function.

Testing CS Inference

```
1 Check (3 == 4).
2 Check (true == false).
3 Eval compute in (3 == 4).
```

Response

```
3 == 4 : bool
true == false : bool
= false
```

More precisely the canonical solutions registers are indexed in a table like the following one

Canonical Structures Index

projection	value	solution	rec calls
sort	nat	nat_eqType	
sort	bool	bool_eqType	

When a unification problem has the shape `(projection ?`$_{solution}$`)` versus `value` such table is looked up to find the corresponding solution for `?`$_{solution}$.

That makes this approach scale well to a large library is that once a relation like `eqType` has been defined and a notation associated to it one can, a posteriori, hook a new new type in. For example a user defining new data type with a comparison function can immediately take advantage of the boilerplate based on `eqType` by simply declaring a canonical structure. To recognize use extensible stuff we use the `Structure` keyword instead of `Record`: record is just for aggregating data type, while structures are for interfaces one can hook a new data type in.

TODO: explain why first class

4.4 Synthesizing a new comparison function

Learns derived instances
Requires Canonical
Provides RecCanonical
Level 1

So far we have used the `==` symbol terms whose type is atomic, like `nat` or `bool` . If we try for example to use it on terms of a compound type like `(nat * bool)` we encounter the following error.

Error

```
1 Check ((3,true) == (false,4)).
2
```

Response

```
Error: complete this
with the real error
```

This is not completely unexpected: we taught Coq how to compare booleans and natural number, not how to compare pairs. The obvious way to compare pairs is to compare them pairwise. Let's write a comparison function for pairs.

Do we have Sections here

Recursive canonical structure

```
1 Definition prod_cmp eqA eqB x y :=
2   eq_op eqA x.1 y.1 && eq_op eqB x.2 y.2.
3 Definition prod_eqType (eqA eqB : eqType) : eqType :=
4   Pack (sort eqA * sort eqB) (cmp_pair eqA eqB)
5 Canonical prod_eqType.
```

Note that the instance `prod_eqType` has two parameter of the same nature. Declaring such instance as canonical generates a program with two recursive calls able to tackle the following unification problem: `(sort ?`$_e$`)` versus `(T1 * T2 )`. Whenever such problem is faced it is reduced to the two smaller problems `(sort ?`$_{e1}$`)` versus `T1` and `(sort ?`$_{e2}$`)` versus `T2`. If both are solved, and hence we have a value v_1 for $?_{e1}$ and v_2 for $?_{e2}$, then the original problem is solved too by picking `(prod_eqType` v_1 v_2`)` for `?`$_e$.

Going back to our example:

Example

```
1  Check (3,true) == (false,4).
2
3
```

Response

```
eq_op
 (prod_eqType nat_eqType bool_eqType)
 (3,true) (false,4) : bool
```

The canonical structure index gets updated to

Canonical Structures Index

projection	value	solution	rec calls
`sort`	`nat`	`nat_eqType`	
`sort`	`bool`	`bool_eqType`	
`sort`	`T1 * T2`	`prod_eqType pA pB`	`pA` = rec(`sort`,`T1`), `pB` = rec(`sort`,`T2`)

Make other examples? Other overloaded stuff: maybe and example of how to hook up to `\in` .

Say we can express functions by recursion over the syntax of types, but also terms when they are injected into a type.

no idea if that output can be produce by Coq

4.5 Declaring implicit arguments

here we describe how to choose which arguments are implicit, that one has to think ahead how a lemma is used and hence which data type inference will have at hand. Also that the order of quantifiers is relevant. Maybe compare with eapply.

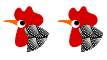
Learns Declaring overloaded notations
Requires Canonical
Provides stating lemmas
Level 2

4.6 Declaring overloaded notations

I guess nosimpl (or Arguments) goes here, plus a little discussion about scopes, the effect `%R` when using a bigop lemma.

Learns Declaring overloaded notations
Requires Canonical
Provides Notation
Level 2

4.7 Triggering type inference



Learns Implement `[foo of nat]`
Requires Canonical
Provides Phantom
Level 3

One does a minimal presentation of phantoms here, so pave the way to the 2 stars section in next chapter where one defines the smart constructor of an algebraic structure.

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)

gotcha: if you see GRing.toto then you broke an abstraction barrier

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