The Book

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

February 27, 2015

Contents

Chapter -1

Conventions

Words:

used	unused
implicit argument, place holder	existential/meta variable
language of canonical structures	hints
declarative program	logic program
type inference	pretyping, elaboration

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

Teaching 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. To keep type checking simple and decidable the syntax of terms makes all information explicit. As a consequence the terms written in such verbose syntax are pretty large.

Luckily the user very rarely interacts directly with the kernel. Instead she almost always interacts 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 can be 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 interactively 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[3]. 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 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 compare 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 fact 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 find an assignment that makes the type `(list _)` convertible to `(list nat)`. By picking the value `nat` for the placeholder the two types become syntactically equal and hence 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 number returns either `true` or `false`. While assigning `2` to the implicit argument would be a perfectly valid solution, it is clear that it is not the only one. Enumerating all possible values until one finds a valid one is not a good strategy either, 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 refer to such language as the language of *Canonical Structures*. Despite being a very restrictive language, it sufficient to program a wide panel of 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

[1] This may change in Coq 8.5

sake of clarity we take the freedom to use the alternative syntax $?_x$ for implicit arguments (where x is a unique name).

4.2 Type inference by examples

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

Lets start with the simplest example one could imagine: 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` takes two arguments. The former argument is expected to have type `Type` and the user left such argument implicit (we name it $?_A$). Type inference imposes that $?_A$ has type `Type`, and this constraint is satisfiable. The algorithm continues checking the remaining argument. 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 $?_A$). For this to be true $?_A$ 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 observe 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 4.6 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 considered 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 the `@` symbol.

Another piece of information that if often left implicit is the type of abstracted or quantified 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 the user is also allowed to put `_` 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. Choosing a good declaration of implicit arguments for lemmas is tricky and requires one to think ahead how the lemma is used. Section 4.6 is dedicated to that.

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 is at the base of the declarative programs we

write to extend unification in the higher order case. As of today there is no precise, published, 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 [1]. The reader interested in a technical presentation of a simplified version of the unification algorithm implemented in Coq can read [4, **?**].

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 type dependency between the two fields requires the first projection to be used in order to define the type of the second projection.

We think of the `eqType` record 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, equal up to computation, we can use the two terms interchangeably. The same holds for `(eq_op nat_eqType)` and `eqn`. Thanks to this fact 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` and its type, `eqType`, is trivially equal to the one expected by `eq_op`. The following two arguments are hence 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 lets us model these practices. Note that, thanks to its higher order nature, the term `eq_op` can always be the head symbol denoting a comparison. This makes it possible to recognize, hence print, comparisons in a uniform way as well as to input them. On he contrary, in the simpler expression `(eqn 3 4)` the name of the head symbol

[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 follow in a proof. This very same approach let us also model this practice, but we discuss it only in the next chapter

is very specific to the type of the objects we are comparing. Also note that polymorphism, in the sense of the ML programming language, is not what we are looking for, since it would impose the comparison function to behave uniformly on every type. What we are looking for is closer to the ad-hoc polymorphism of the Haskell programming language or the notion of subtyping provided by object oriented languages.

In the rest of this chapter we focus on the overloading of the == symbol and we start by defining another comparison function, this time for the bool data type.

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

Now the idea is to define a notation that applies to any occurrence of the eq_op head constant and use such notation for both printing and parsing.

I need the reader to know something about Notation

Overloaded notation

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

Response

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

As a printing rule, the place holder stands for a wild card: the notation is used no matter the value of the first argument of eq_op. As a result both occurrences of eq_op, line 2 and 3, are printed using the infix == syntax. Of course the two operations are different, they are specific to the type of the arguments and the typing discipline ensures the arguments match the type of the comparison function packaged in the record.

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

Error

```
1  Check (3 == 4).
2
```

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 place holder $?_e$. 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: the system has to synthesize a comparison function (or better a record instance containing a comparison function).

Coq gives up, leaving to the user the task of extending the unification algorithm with a program that is able to solve unification problems of the form (sort $?_e$) versus T for any T. Given the current context it seems reasonable to 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

```
1 Canonical nat_eqType.
2 Canonical bool_eqType.
```

The keyword `Canonical` was chosen to stress that the program is deterministic: each type `T` is 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
```

The mechanics of the small program we wrote using the `Canonical` keyword can be explained using the global table of canonical solutions. Whenever a record instance is declared as canonical Coq adds to such table an entry for each field of the record type.

Canonical Structures Index

projection	value	solution
`sort`	`nat`	`nat_eqType`
`sort`	`bool`	`bool_eqType`

Whenever a unification problem with the following shape is encountered, the table of canonical solution is consulted.

`(projection ?`$_S$`)` versus `value`

The table is looked up using as keys the projection name and the value. The corresponding solution is assigned to the implicit argument `?`$_S$.

In the table we reported only the relevant entries. Entries corresponding to the `eq_op` projection play no role and in the Mathematical Components library the name of such projections is usually omitted to signal that fact.

What makes this this approach interesting for a large library is that record types can play the role of interfaces. Once a record type has been defined and some functionality associated to it, like a notation, one can easily hook a new concept up by defining a corresponding record instance and declaring it canonical. One gets immediately all the functionalities tied to such interface work on the new concept. For example a user defining new data type with a comparison function can immediately take advantage of the overloaded `==` notation by packing the type and the comparison function in an `eqType` instance.

This pattern is so widespread and important that the Mathematical Components consistently uses the synonym keyword `Structure` in place of `Record` in order to make record types playing the role of interfaces easily recognizable.

Records are first class values in the Calculus of Inductive Constructions. As we have seen projections are no special, they are simple functions that pattern match on an inductive data type to access the record fields. Being first class

citizens means that one can write a term that combines the fields of two records and builds a new record. Thanks to this fact the language of Canonical Structures is able to forge new record instances by combining the existing ones via a set of user definable combinators. This is the subject of the next section.

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 whose type was built using a type constructor like the Cartesian product we encounter an error.

Error

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

```

Response

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

The term `(3,true)` has type `(nat * bool)` and, so far, we only taught Coq how to compare booleans and natural numbers, not how to compare pairs. Intuitively the way to compare pairs is to compare their components *using the appropriate comparison function*. Let's write a comparison function for pairs.

Do we have the Sections mechanism here?

Comparing pairs

```
Definition prod_cmp eqA eqB x y :=
  eq_op eqA x.1 y.1 && eq_op eqB x.2 y.2.
```

What is interesting about this comparison function is that the pairs `x` and `y` are not allowed to have an arbitrary, product, type here. The typing constraints imposed by the two `eq_op` occurrences forces the type of `x` and `y` to be `(sort eqA * sort eqB)`. This means that the records `eqA` and `eqB` hold a sensible comparison function for, respectively, terms of type `(sort eqA)` and `(sort eqB)`.

It is now sufficient to pack together the Cartesian product type constructor and this comparison function in an `eqType` instance to extend the canonical structures inference machinery with a new combinator.

Recursive canonical structure

```
Definition prod_eqType (eqA eqB : eqType) : eqType :=
  Pack (sort eqA * sort eqB) (cmp_pair eqA eqB)
Canonical prod_eqType.
```

The global table of canonical solutions is extended as follows.

Canonical Structures Index

projection	value	solution	combines solutions for
`sort`	`nat`	`nat_eqType`	
`sort`	`bool`	`bool_eqType`	
`sort`	`T1 * T2`	`prod_eqType pA pB`	`pA` ⟵ `(sort,T1)`, `pB` ⟵ `(sort,T2)`

The third column is empty for base instances while it contains the recursive calls for instance combinators. With the updated table when the unification problem

`(sort ?`$_e$`)` versus `(T1 * T2)`

is encountered a solution for `?`$_e$ is found by proceeding in the following way. Two new unification problems are generated: `(sort ?`$_{eqA}$`)` versus `T1` and `(sort ?`$_{eqB}$`)` versus `T2`. If both are successful and `v1` is the solution for `?`$_{eqA}$ and `v2` for `?`$_{eqB}$, the solution for `?`$_e$ is `(prod_eqType v1 v2)`.

After the table of canonical solutions has been extended our example is accepted.

no idea if that output can be produce by Coq

Example

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

Response

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

Make other examples? Other overloaded stuff: maybe and example of how to hook up to infix in ? or locked? or whatever? In any case a table with "all" the interfaces should probably be part of the book

In the running example of this chapter we use the canonical structures language to express structurally recursive programs on the syntax of types. The Calculus of Inductive Constructions allows arbitrary terms to occur inside types. As a consequence the language of canonical structures can express also structurally recursive programs on the syntax of terms. This capability is used, for example, in the next chapter to related Monoid laws to function symbols to model the syntax and theory of iterated, "big", operators.

4.5 Other aspects of type inference

Learns Coercions
Requires
Provides
Level 1

Even if the main way to extend the type inference algorithm is via Canonical Structures, another mechanism is available and used all over the library, even if it plays a minor role. The language of Canonical Structures lets one program how the value of an implicit argument can be synthesized, but can hardly be used to explain Coq how to "fix" an ill-typed term written by the user.

When a typing error arises, it always involves three objects: a term `t`, its type `ity` and the type expected by its context `ety`. Of course, for this situation to be an error, the two types `ity` and `ety` do not compare as equal. The simplest way one has to explain Coq how to fix `t`, is to provide a functional term `c` of type `(ity → ety)` that is inserted around `t`. In other words, whenever the user writes `t` in a context that expects a term of type `ety`, the system instead of raising an errors replaces `t` by `(c t)`.

A function automatically inserted by Coq to prevent a type error is called *coercion*. The most pervasive coercion in the Mathematical Components library is `is_true` that lets one write statements using boolean predicates..

I guess in a way or another is true has already been introduced

Coercion `is_true`

```
1  Lemma example : prime 27.
2  Proof.
3  Set Printing Coercions.
4  by [].
5  Qed.
```

Goal after line 3

```
==================
is_true (prime 27)
```

The statement of the example is processed by type inference, it is enforced to be a type, but `(prime 27)` is actually a term of type `bool`. Early in the library the function `is_true` is declared as a coercion from `bool` to `Prop` and hence is it inserted by Coq automatically.

```
1  Definition is_true b := b = true.
2  Coercion is_true : bool >→  Sortclass. (* Prop *)
```

Another coercion that is widely used injects booleans into naturals. Two examples follow.

```
1  Fixpoint count (a : pred T) (s : seq T) :=
2    if s is x :: s' then a x + count s' else 0.
3  Lemma count_uniq_mem s x : uniq s → count_mem x s = (x \in s).
```

In line number 2 the term `(a x)` is a boolean. The `nat_of_bool` function is automatically inserted to turn `true` into 1 and `false` into `0`. Similarly, in the last line the membership test is turned into a number, that is shown to be equivalent to the count of any element in a list that is duplicate free.

Another example of a coercion that is related to the running example of the current chapter is `sort`. Typically the projection of a record type extracting the data type is declared as a coercion letting one state generic theorems like in the following example.

```
1  Lemma (e : eqType) : ∀ x y : e, x == y → ...
```

Here the type of `x` and `y` is `(sort e)` and not `e` as the user initially wrote. Indeed `e` is a term (of type `eqType`) while the `forall` quantification expects a type after the colon. The `sort` function mapping an `eqType` into a `Type` is inserted automatically.

Coercions are composed transitively.

```
1  Check ∀ b : bool, (b + 3)%Z.
```

For the convenience of the reader we list here the most widely used coercions. there are also a bunch on Funclass not listed and elimT surely deserves some explanation.

Coercions

coercion	source	target
Posz	nat	int
nat_of_bool	bool	nat
elimT	reflect	Funclass
isSome	option	bool
is_true	bool	Sortclass

Another device that is used to help type inference is the `Implicit Types` directive. This directive lets one attach a default type to variable names.

Example of `Implicit Types`

```
1 Implicit Types m n : nat.
2 Check ∀ m n, n == m.
```

In the example above the statement we `Check` does not contain enough information alone to be well types. The overloaded `==` notation needs the terms to which it is applied to have a type for which a `Canonical Structure` is declared. Even if we did not annotate `n` and `m` with a type, the directive on the first line does it for us.

The reader already familiar with the concept of coercion may find the presentation of this chapter nonstandard. Indeed coercions are usually presented as a device to model subtyping in a theory that, like Calculus of Inductive Constructions, does not feature subtyping. As we will see in Chapter **??** the role played by coercions is in the modelling of the hierarchy of algrabraic structure is minor. Indeed what is hard is not to forget some fields of a structure to obtain a simpler one. What is hard is to reconstruct the missing fields of a structure or compare two structures finding the minimum super structure. These tasks are mainly implemented using canonical structures.

4.6 Declaring implicit arguments

Learns Declaring implicit arguments
Requires Canonical
Provides stating lemmas
Level 2

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 has at hand. Also that the order of quantifiers is relevant.

- lemmas: fwd/backward reasoning
- equations, look at the concl too, free vars are abstracted
- compare with eapply style

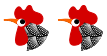
4.7 Declaring overloaded notations

Notations must be stable.

Learns Declaring overloaded notations
Requires Canonical
Provides Notation
Level 2

- Arguments nosimpl
- + that becomes * with a CS inference
- scopes the effect `%R` when using a bigop lemma.

4.8 Triggering type inference



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.

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

- lifting a term into a type
- notations are untyped

4.9 Discussion about type inference

The points I want to make are:

Learns difference with type classes, limitations
Requires RecCanonical,Coqercions
Provides
Level 2

- during type inference, HO infers (morally) the minimum info necessary to make things well typed (if we were 1st order that would really be true). hence we KNOW when CS inference is triggered EXACTLY. this is a big difference w.r.t. type classes. It is like a Prolog program where goals are reordered randomly
- I also want to talk about the overlapping instances problem, that is "easy" with TC, hard with CS and envisage an extension.
- limitation of coercions, from both the usability perspective and the expressive power they offer

Chapter 5

Mixing data and proofs

Organizing knowledge

Organizing, structuring, knowledge is a delicate task. A successful path followed in mathematics is algebraization. The focus shifts towards the relations among classes of similar objects and general theories that apply to all the objects of a given class. An experienced mathematician can use such structured knowledge by recognizing that his object of study belongs to a know class and hence a string of generic results holds for it.

Given how popular this approach is in modern mathematics we can't really avoid modelling it in the Mathematical Components library. It is well known that modelling structured knowledge, like algebra, inside a proof assistant is hard, and in our case an extra problem arises: we need such modelling to scale up to a large library. This extra difficulty turns out to actually provide good directions for finding the right techniques. Indeed, computer science too tackled the task of organizing large libraries of computer code, and did it for decades. One of the outcomes is object oriented programming (OOP). It is hard to give a comprehensive presentations of all the key principles behind OOP, but some of them match pretty well our needs. OOP advocates the use of interfaces (typically called classes) to organize a complex system into smaller parts, it enables one to declare relations between the interfaces to reuse and specialize them (inheritance). Finally OOP language provides some magic glue to make all that work together, enabling one to mix and match data and code belonging to different but related interfaces, as long as it "makes sense". What makes sense really means depends too much on the programming language details to be explained in general, and if we draw a parallel with mathematics this pretty much corresponds to the silent check an expert eye does when reading a text mixing different theories. When we look at functional programming languages, like the one provided in Calculus of Inductive Constructions, the aspect of this magic glue that is the most relevant here is *sub-type* polymorphism.

Its intuitive meaning can be explained with an example. Take the two related notions (types) of a simple point `P` on the plane and the more refined concept of coloured point `CP`. To maximize code reuse we want the function `move`, that shifts a point, to also work on points that happen to have a color. If we assign the type `P` $\rightarrow$ `P` to such function, then we can't use it on a coloured point `c`, because `(move c)` would be ill-typed. We could try to assign to `move` the polymorphic type $\alpha \rightarrow \alpha$ (for all α), but that would fail too. The function `move` really needs to know something about its input, like its coordinates, in order to perform. The polymorphic type requires the implementation of `move` to bee too generic, and in practice we can only inhabit the type $\alpha \rightarrow \alpha$ with the identity function,[1] that does not exactly move the point as one expects. Sub-type polymorphism lets one assign a polymorphic type to `move` where the type variable α is constrained to be "at least" a point, a constraint satisfied by all coloured points.

The record data types introduced in the previous chapter played the role of expressing a relation between a type and a function (the comparison one). Being records dependently typed, their fields can also carry proofs testifying that some property holds on the value of the other fields. This in turns makes it possible to use records to model precisely the concept of interface. Programmable type inference can then serve as the glue that links an object with all the interfaces it satisfies.

In this chapter we focus on the tricky construction of *dependent pairs packaging data and proofs*. To take full advantage of this construction while doing proofs we implement the glue typical of OOP languages by *programming type inference*. Inheritance among interfaces will be only sketched here: a complete treatment of it requires a full chapter, the next one.

5.1 The many ways to define a "sub-type"

Learns panel of options
Requires dependent product
Provides sigma with bool predicate
Level 1

First of all, the Calculus of Inductive Constructions has no built-in notion of sub-typing and what we propose "implements" the mechanics of sub-typing on top of records and programmable type inference. We begin by studying a simple example of sub-type and reach our proposed encoding by first showing some alternatives and discussing their drawbacks.

We pick the sub-type of sequences of a given length as our case study. The objective is to ease the reasoning on sequences when manipulated by operations that alters the length in a known way, while retaining all the theory we already have on general sequences.

An approach to describe such data type that is possible in type theories with indexed inductive families like the Calculus of Inductive Constructions is to craft a new ad-hoc type, the type of vectors.

[1] there is even a theory about that, called parametricity

Vectors

```
Inductive vect T : nat → Type :=
| vnil : vect T 0
| vcons n of T & vect T n : vect T n.+1.
```

In this construction the type carries the extra information we want. For example `(vect nat 7)` describes a sequence of natural numbers of length 7. One can define operations on the `vect` data type that signal, in their type, the effect they have on the length of the vectors they manipulate. For example one can read in the type of the function that reverses a vector that it preserves the length (`n` here).

Vector reversal

```
Fixpoint vrev T n (v : vect T n) : vect T n := ...
```

This means that there will be no need to convince Coq that `vrev` preserves the length of the input vector each time it is used: we morally prove it once an for all when we define the operation.

Vector assisted proof

```
Definition vlength T n (v : vect T n) := n.
Lemma simple T n v : vlength T n v = vlength T n (vrev T n v).
Proof. by []. Qed.
```

The drawback of this approach is that the type `(vect T n)` and `(seq T)` are different and we can't directly apply to a vector an operation that is defined on sequences. For example we already have a reversal function for sequences called `rev`, but writing `(rev vnil)` results in a type error. And when new operations have to be defined, new proofs have to be done. For example, even if we already proved that `rev` is an involution, we have to prove the same property from scratch for `vrev`.

Of course one can relate sequences and vectors by means of morphisms. Even if the Mathematical Components library provides adequate infrastructure for dealing with morphisms, its use is generally avoided when possible. GIVE A BETTER INSIGHT.

The conclusion is to drop the idea of defining a completely new data type, and try to reuse the one of sequences. For example by packing a sequence with a proof that witnesses that its length is exactly `n`. Such construction, a dependent pair packing data and proofs, is tricky to get right in the Calculus of Inductive Constructions. We start with the following "naive" record type `(lseq T n)`:

Simple dependent pair

```
Inductive has_len (T : Type) : seq T → nat → Prop :=
| base : has_len T [::] 0
| step n l a of has_len T l n : has_len T [:: a & l] n.+1.

Record lseq T n := Lseq { lval : seq T; llen : has_len T val n }.
```

The advantage of this approach over the former is that the `lval` projection can be used to get a `seq` out of an `lseq` directly. And using the `Coercion` mechanism the system would even insert it automatically. Hence writing `(rev l)` for `(l : lseq T n)` would be accepted by the system.

Note that so far records only contained types and terms, typically a function (like the comparison used to implement the overload `==` syntax). Here the record contains *a term and a proof*. Proofs are first class citizens in all meanings, and this may lead to a somewhat surprising behavior of the type theory. When we defined the function to compare pairs we had no doubts: both components of the pair play a role and must be taken into account. When comparing two elements of `lseq` we would like the second component not to matter. In other words we want this lemma to be provable:

Injectivity of lval

```
Lemma lval_inj T n (l1 l2 : lseq T n) : lval l1 = lval l2 → l1 = l2.
```

Intuitively it says that two sequence of length `n` are equal if they are equal as sequences, it does not matter *how one proved* that they have length `n`.

Unfortunately this fact is in general not true in Calculus of Inductive Constructions: all record components are relevant to equality. Luckily, for some inductive relations like `has_len`, one can show that their inhabitants (that are proofs) are canonical. In other words that two proofs of the same statement are equal. Unfortunately such proof, when possible, is not only tricky, but it also depends on the shape of the inductive relation. Hence every time one wants to form a sub-type expressing a new property, he has to prove such lemma again for the new predicate.

To the rescue comes the result of Hedberg [2] that proves such property for a wide class of predicates. In particular he shows that any type with decidable identity has unique identity proofs. If we pick the concrete example of `bool`, then all proofs that `(b = true)` for a fixed `b` are the same. As a direct application of this result we have that, as long as we can express the property defining the sub-type as a boolean predicate, we can reuse the same injectivity lemma.

Tuple sub-type of seq

```
Structure tuple_of n T := Tuple { tval :> seq T; _ : size tval == n }.
Notation "n .-tuple T" := (tuple_of n T) (at level 2).
```

Recall the hidden `is_true` coercion. If we unfold its definition we clearly see that the predicate `(size tval == n)` is an equality on `bool`.

In the Mathematical Components library, where *all predicates that can* be expressed as a boolean function *are expressed as a boolean function*, forming sub-types is extremely easy.

5.2 Working with sub-types

Learns CS = proof search
Requires records + CS
Provides working with tuples
Level 1

For the sake of this section, we will use this, equivalent, definition of the tuple type. Note that we use `=` in place of `==`.

Tuple sub-type of seq

```
Structure tuple_of n T := Tuple { tval :> seq T; _ : size tval = n }.
```

Now we focus on the use of the tuple type. The key property of this type is that it tells us the length of its elements when seen as sequences:

```
Lemma size_tuple {T n} (t : n.-tuple T) : size t = n.
Proof. by case: t. Qed.
```

In other words each inhabitant of the tuple type brings with it, in the form of a proof, its length. As test bench for the notion of tuple we pick this simple example: a tuple is processed using functions defined on sequences, namely `rev` and `map`.

```
Example seq_on_tuple n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x ←rev t]) = size t.
```

There are two ways to prove that lemma. The first one is to ignore the fact that `t` is tuple, consider it as a regular sequence, and use only the theory of sequences.

```
Proof. by rewrite map_rev revK size_map. Qed.
```

Mapping a function over the reverse of a list, it equivalent to first map the function over the list and then reverse the result (`map_rev`). Then, reversing twice a list is a no-op, since `rev` is an involution (`revK`). Finally, mapping a function over a list does not change its size (`size_map`). The sequence of rewritings make the left hand side of the conjecture identical to the right hand side, and we can conclude.

This simple example shows that the theory of sequences is usable on terms of type tuple. Still we didn't take any advantage of the fact that `t` is tuple.

The second way to prove this theorem is to rely on the rich type of `t` to actually compute the length of the sequence.

```
Example just_tuple_attempt n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x ← rev t]) = size t.
Proof. rewrite size_tuple.
```

The rewriting replaces the right hand side with `n` as expected, but we can't go any further: the lemma is not replacing the right hand side with `n`, even if we are working with a tuple `t`. Why is that? In the left hand side `t` is processed using functions on sequences. The type of `rev` for example is `(∀ T, seq T → seq T)`. The coercion `tval` from `tuple_of` to `seq` make the expression `(rev (tval t))` well typed, but the output is of type `(seq nat)`. If we go back to our example of the `move` function for points in the plane, it is like assigning to it the type `(α → P)` for α being at least a point. We would like the function to return a data as rich as the one it takes in input.

What happens if try to unify the left hand side of the `size_tuple` equation with the redex `(rev t)` (and we print coercions)?

Toolkit not belonging here

```
(* toolkit *)
Notation "X (*...*)" := (let x := X in let y := _ in x)
  (at level 100, format "X (*...*)").
Notation "[LHS 'of' equation ]" :=
  (let LHS := _ in
   let _infer_LHS := equation : LHS = _ in LHS)
  (at level 4).
Notation "[unify X 'with' Y ]" :=
  (let unification := erefl _ : X = Y in
   True).
```

Simulate rewrite size_tuple

```
Check ∀ T n (t : n.-tuple T),
 let LHS := [LHS of size_tuple _] in
 let RDX := size (rev t) in
 [unify LHS with RDX]
```

Response

```
Error:
In environment
T : Type
n : nat
t : n .-tuple T
LHS := size (tval ?94 ?92 ?96) (*...*) : nat
RDX := size (rev (tval n T t)) : nat
The term "erefl ?95" has type "?95 = ?95" while
it is expected to have type "LHS = RDX".
```

Unifying `(size (tval ?n ?T ?t))` with `(size (rev (tval n T t)))` is hard. Both term's head symbol is `size`, but then the projection `tval` applied to unification variables has to unified with `(rev ...)`, and both terms are in normal form.

Such problem is nontrivial because to solve it one has to infer a record for $?_t$ that contains a proof: a tuple whose `tval` field is `rev t` (and whose other field contains a proof that such sequence has length $?_n$).

This unification problem falls in the class handled by canonical Canonical Structures: a record projection against a value. We can declare Canonical Structures teaching Coq the effect of list operations over the length of their input.

```
Lemma rev_tupleP n A (t : n.-tuple A) : size (rev t) = n.
Proof. by rewrite size_rev size_tuple. Qed.
Canonical rev_tuple n A (t : n.-tuple A) := Tuple (rev_tupleP t).

Lemma map_tupleP n A B (f: A → B) (t : n.-tuple A) : size (map f t) = n.
Proof. by rewrite size_map size_tuple. Qed.
Canonical map_tuple n A B f (t : n.-tuple A) : n.-tuple B :=
  Tuple (map_tupleP f t).
```

Even if it is not needed for the lemma we took as our test bench, we add another example where the length is not preserved.

```
Lemma cons_tupleP n A (t : n.-tuple A) x : size (x :: t) = n.+1.
Proof. by rewrite /= size_tuple. Qed.
Canonical cons_tuple n A x (t : n.-tuple A) : n.+1 .-tuple A :=
  Tuple (cons_tupleP t x).
```

The global table of canonical solutions is extended as follows.

Canonical Structures Index

projection	value	solution	combines solutions for
`tval N A`	`rev A S`	`rev_tuple N A T`	`T ← (tval N A, S)`
`tval N B`	`map A B F S`	`map_tuple N A B F T`	`T ← (tval N A, S)`
`tval N.+1 A`	`X :: S`	`cons_tuple N A T X`	`T ← (tval N A, S)`

Thanks to the now extended capability of type inference we can prove our lemma by just reasoning about tuples.

```
Example just_tuple n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x ←rev t]) = size t.
Proof. by rewrite !size_tuple. Qed.
```

The iterated rewriting acts now twice replacing both the left hand and the right hand side with `n`. It is worth observing that the size of this proof (two rewrite steps) does not depend on the complexity of the formula involved, while the one using only the theory of lists requires one step per list-manipulating function. What depends on the size of the formula is the number of canonical structure resolution steps type inference performs. Another advantage of this last approach is that, unlike in the first, one is not required to know the names of the lemmas: it the new concept of tuple that takes care of the size related reasoning.

Going back to the example of (coloured) points, once a coloured point is

defined as a record embedding a simple point

```
1  Record CP := Coloured { coords_of :> P; colour_of : color }
```

the way we implement sub-type polymorphism between `CP` and `P` is by assigning the typed `(P → P)` to `move` but:

- thanks to the automatically inserted *forgetful coercion* `coords_of` we can apply `move` to a coloured point `c` obtaining `(move (coords_of c))`
- thanks to *programmable type inference* such expression can be automatically injected back in the type of coloured points by using the color of `c` and obtain `(Coloured (move (coords_of c)) (colour_of c))`

That is pretty much what ones expects `move` to do when applied to a coloured point: act on its coordinates and leave its colour untouched.

We have seen how to implement the glue that links a type (sequences) with a sub-type (tuples), but only in a very simple case. A tuple is trivially a sequence (by forgetting something), hence the theory of sequences "trivially" applies to tuples.

We now revise the definition of `eqType`, turning it into a real interface bringing with it a specific theory. Later we show that since `seq` is an instance of that interfaces and since `tuple` is a sub-type of `seq`, then the theory of `eqType` also applies to tuples.

5.3 Types with a decidable equality

Learns interface to a theory
Requires records + CS
Provides eqType
Level 1

The definition of `eqType` actually used in the Mathematical Components library is more complex than the one we saw in Section **??**. Indeed there is little point in associating to a type a comparison function if that function is not "correct".

eqType

```
Definition rel T := T → T → bool.

Module Equality.

Definition axiom T (e : rel T) := ∀ x y, reflect (x = y) (e x y).

Record class_of T := Mixin {op : rel T; _ : axiom op}.

Structure eqType : Type := Pack {
  sort : Type;
  class : class_of sort;
}.

End Equality.

Definition eq_op T := Equality.op (Equality.class T).
Notation "x == y" := (@eq_op _ x y).
```

The reason why the operation and the property are packed in a record called `class_of` will be described in the next chapter. Intuitively one could think the construction as flat: the relation now links a type, a comparison function and a witness that this comparison function actually reflects equality. Reflecting equality means that the comparison function returns `true` if and only if the two inputs are provably equal, and `false` when they are provably different.

Mantra

Whenever we write or read `==`, the intended meaning is a decidable comparison compatible with `=`

Given the new definition of `eqType`, when we write `(a == b)` type inference does not only infer a function to compare `a` with `b` but also a proof that such function is correct.

The simplest application is the `eqP` *generic lemma* that can be used in conjunction with any formula involving the `==` notation.

The eqP lemma

```
Lemma eqP (T : eqType) : Equality.axiom (eq_op T).
Proof. by case: T => ty [op prop]; exact: prop. Qed.
```

The proof is just unpacking the input `T`. Since type inference can synthesize an arbitrary complex value for `T` by composing canonical structures instances, this lemma give us a access to the proof part of such inferred value.

Use of eqP

```
Lemma test (x y :nat) (a b : bool) : (a,x) == (b,y) → fst (a,x) == b.
Proof. by move/eqP=> def_ax; rewrite def_ax. Qed.
```

The `(a,x) == (b,y)` assumption is reflected to `(a,x) = (b,y)` by using the `eqP` view specified by the user. Here we write `==` to have all the benefits of a computable function (simplification, reasoning by cases), but when we need the underlying logical property of substitutivity we access it via the view `eqP`.

In such sense, `==` has now a precise meaning. In general overloaded notation come with a meaning and an associated theory. Note that the proof language silently adjusted the view using `elimT`.

The eqtype view

```
Check elimT.
```

Response

```
elimT : ∀ (P : Prop) (b : bool),
          reflect P b → b → P
```

It is important to remark that `eqP` works for any instance of the `eqType` structure.

Now `eqType` can be seen as an interface to access a theory of results. Indeed we can finally state (and prove) the Hedberg theorem in its full generality.

Hedberg

```
Theorem eq_irrelevance (T : eqType) (x y : T) : ∀ e1 e2 : x = y, e1 = e2.
```

Now that the `eqType` structure comes with a well specified comparison function we can use it to program new functions, and reason about them. For example by imposing that that the type variable of a container types, like `seq`, is an `eqType` we can code the following function that tests membership.

Functions

```
Fixpoint mem_seq (t : eqType) (s : seq T) x :=
  if s is y :: s' then (y == x) && mem_seq T s' x else false.
```

Indeed we are allowed to write `(x == y)` because `T` is not any type, but a type with a decidable equality.

We spare the reader the definition of eqType for seq. MAYBE IN THE EXERCISES.

If the elements of a sequence are in an eqType, we can obtain the overload the `\in` notation on sequences. In turn we can then define new predicates like being duplicate free.

```
Fixpoint uniq s :=
  if s is x :: s' then (x \notin s') && uniq s' else true.
Fixpoint undup s :=
  if s is x :: s' then
    if x \in s' then undup s' else x :: undup s'
  else [::].
```

And show properties of these functions

Functions

```
Lemma undup_uniq s : uniq (undup s).
```

Note that the Mathematical Components library iterates the construction of `eqType` on the container itself: if `T` is `eqType` then also `seq T` is.

```
Let s1 := [:: 1; 2 ].
Let s2 := [:: 3; 5; 7].
Let ss := [:: s1 ; s2 ].
Check ss == [::].
Check uniq ss.
Check undup_uniq ss.
```

Indeed we can compare sequences of sequences, and apply to them the theory of sequences. Unfortunately this does not work with tuples (yet).

```
Fail Check ∀ (t : 3.-tuple nat), [:: t] == [::].
Fail Check fun t : 3.-tuple nat => uniq [:: t; t].
Fail Check fun t : 3.-tuple nat => undup_uniq [:: t; t].
```

Tuples are trivially sequences. Here we need to transport the property of being an eqType of sequences on tuples.

5.4 Building the `eqType` for `n.-tuples T`

Learns it is a schematic process
Requires proof language
Provides an example of sub-type of eqType
Level 1

We have that `(seq T)` is an `eqType` and we want to transport `==` on tuples. First we need to defined a comparison function for tuples.

Comparison of tuples

```
Definition tcmp n (T : eqType) (t1 t2 : tuple_of T n) :=
  tval t1 == tval t2.
```

Here we reuse the one on sequences, and we ignore the proof part of tuples.

We need now to prove

```
1  Lemma eqtupleP n (T : eqType) : Equality.axiom (tcmp n T).
2  Proof.
3  move=> x y; apply: (iffP eqP); last first.
4    by move→ .
5  case: x; case: y => s1 p1 s2 p2 /= E.
6  rewrite E in p2 *.
7  by rewrite (eq_irrelevance p1 p2).
8  Qed.
```

The first direction is trivial

Response line 4

```
T : eqType
n : nat
x : tuple_of T n
y : tuple_of T n
==================
x = y → tval x = tval y
```

For the other one the crucial step is the use of `eq_irrelevance`.

Response line 6

```
T : eqType
n : nat
s1 : seq T
p1 : size s1 == n
s2 : seq T
p2 : size s2 == n
E : s2 = s1
==================
Tuple T n s2 p2 = Tuple T n s1 p1
```

We can then declare

```
1  Canonical tuple_eqType n T :=
2    Equality.Pack (Equality.Mixin (tcmp n T) (eqtupleP n T)).
```

Simple test

```
1  Check ∀ (t : 3.-tuple nat), [:: t] == [::].
2  Check fun t : 3.-tuple nat => uniq [:: t; t].
3  Check fun t : 3.-tuple nat => undup_uniq [:: t; t].
```

We did it by hand, but all that is schematic, any sigma type can be dealt with in the very same way. When one has a base type T and a sub-type ST defined as a boolean sigma type, then one can build all the canonical instances by just knowing the name of the projection going from ST to T.

The lirbary is equipped with all the canonical structures that possibly apply. Only the reader willing to extend the library with new concepts is interested by what follows.

5.5 The toolkit to declare a new sub-type

As simple as such

```
Canonical tuple_subType := Eval hnf in [subType for tval].
Definition tuple_eqMixin := Eval hnf in [eqMixin of n.-tuple T by <:].
Canonical tuple_eqType := Eval hnf in EqType (n.-tuple T) tuple_eqMixin.
```

Learns Declare a new sub-type and derive eqType
Requires
Provides usage of `[ subType for ...]`
Level 2

maybe we can put here the nice trick for encoding a bizarre structure into a three and get the eqType out of it. In such case we need to rename the section into: declaring a new eqType (not only for subtypes).

5.6 The `subType` infrastructure

```
Structure subType : Type := SubType {
  sub_sort :> Type;
  val : sub_sort → T;
  Sub : ∀ x, P x → sub_sort;
  (* elim rule for the record *)
  _ : ∀ K (_ : ∀ x Px, K (@Sub x Px)) u, K u;
  _ : ∀ x Px, val (@Sub x Px) = x
}.

Notation "[ 'subType' 'for' v ]" := (SubType _ v _ inlined_sub_rect
    vrefl_rect)
 (at level 0, only parsing) : form_scope.
```

Learns How sub-type works
Requires CS
Provides super CS skills
Level 3

5.7 Guided tour of widespread eqtypes and their sub-types

Examples: nat, 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.

Learns contents
Requires
Provides ord, ...
Level 1

5.8 Sub-types in HoTT

Learns curiosity
Requires
Provides
Level 2

Recent advances in HoTT identify the class of propositions that have a canonical proof, mere propositions, and such class is closed under forall quantification, while the one we use is not (only bounded forall) but the idea is pretty much the same. Mere propositions can be easily used to form a subtype.

Also Cyril had other arguments on the fact that one can ask less (be more general) and recover all required properties later on, but I've to ask him again cause I've forgotten the example.

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

Bibliography

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[2] Michael Hedberg. A coherence theorem for martin-löf's type theory. *J. Funct. Program.*, 8(4):413–436, July 1998.

[3] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, page pages, 1992.

[4] Matthieu Sozeau and Beta Ziliani. Towards a better behaved unification algorithm for coq. In *Proceedings of The 28th International Workshop on Unification*, 2014.