An Introduction to MathComp-Analysis

カラテ Coq

Reynald Affeldt

January 27, 2023

Contents

Abstract

This document is a memo written for a class of about ten hours held at the Graduate School of Mathematics at Nagoya University from [2022-12-19] to [2022-12-23]. The intent is to provide the necessary background about the Coq proof assistant and the MathComp library (for students who already had an exposition to these pieces of software) to be able to understand and get started with the MathComp-Analysis library. This document is meant to be self-contained. Since Coq and MathComp are already explained elsewhere, the parts about them are rather cursory, relying on pointers to the appropriate literature such as the Coq reference manual [The Coq Development Team, 2022], the original SSReflect manual [Gonthier et al., 2016], and the Mathematical Components book [Mahboubi and Tassi, 2021]. And for Japanese readers: [Affeldt, 2017], [Hagiwara and Affeldt, 2018].

Revision history:

- First version: [2022-12-23]
- Second version: [2023-01-27] (typos)

Chapter 1

Overview of Coq and MathComp

Goal of this chapter: This chapter is an overview about proof assistants based on dependent type theory and serves as an introduction to the topic of this document which is more specifically about the MathComp-Analysis *library.*

1.1 A Bit of History

The creation of proof assistants is the result of research on the foundations of mathematics that has been happening since the last century.

It started with the discovery of contradictions in early set theory, contradictions such as this one explained by Russell in 1901: $a \in a$ is equivalent to $a \notin a$ if one defines $a \stackrel{\text{def}}{=} \{x \mid x \notin x\}$. Set theory has been quickly patched to avoid such contradictions.

The idea to use types to prevent contradictions provides an alternative to set theory to describe the foundations of mathematics. This alternative has been proposed by Russell in 1908:

> Whatever contains an apparent variable must not be a possible value of that variable. (Bertrand Russell, [Russell, 1908])

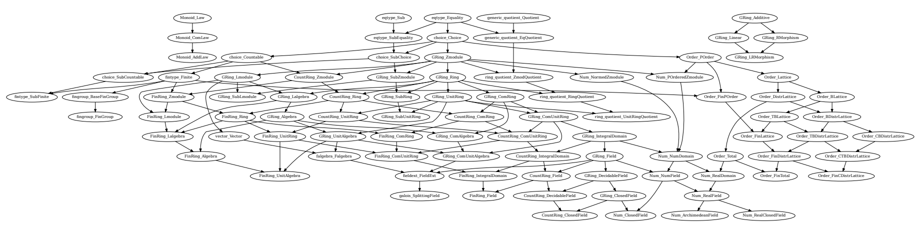
The theory of types has been developed in the *Principia Mathematica* written in 1910–1913 by Whitehead and Russell [Whitehead and Russell, 1927]. In the 1930s, Curry showed a correspondence between propositional logic and combinators. In 1940, Church used the λ-calculus to propose the simple theory of types [Church, 1940]. It was becoming clear that types could be used to perform proof checking and that there was a relation with algorithms. This ultimately led to the *Curry-Howard correspondence* in 1969 [Howard, 1980].

Proof checking using types was soon implemented using a computer by de Bruijn in 1967–1968 as the proof assistant AUTOMATH. Research on type theory and its implementation continued in the 1970s with Martin-Löf and Milner in particular and led to the implementation of LCF among others. As for the Coq proof assistant (Coq website), its implementation started in France in 1984 and is still thriving.

It is worth noting that in parallel efforts were also undertaken to put more structure in mathematics. In particular, starting in 1934–1935, Bourbaki has been using set theory for that purpose. On this occasion, Bourbaki put an emphasis on the notion of mathematical structure. However:

> Theory of Sets was meant to provide a formally rigorous basis for the whole of the treatise, and the concept of structure represented the ultimate stage of this undertaking. The result, however, was different: Theory of Sets appears as an ad-hoc piece of mathematics imposed upon Bourbaki by his own declared positions about mathematics, rather than as a rich and fruitful source of ideas and mathematical tools. (Leo Corry, [Corry, 1992])

Retrospectively, it seems that the notion of mathematical structure could not really be used systematically for the lack of a mechanical tool. In some sense, the Mathematical Components project (Mathematical Components website) that started at the Inria-Microsoft Research Joint Center in 2005 can be seen as an attempt at fulfilling this goal. By the way, here is what the hierarchy of mathematical structures in core MathComp looked like in August 2022:



With that many structures, it is no wonder that one cannot manage on paper. See Fig. 3.1, page 41 for a bit more readable hierarchy.

1.2 What are Proof Assistants Good for?

Today formal verification has emerged as a mean to guarantee the correctness of software and mathematics and it is often carried out using proof assistants.

The most obvious application of formal verification is to prevent bugs in computer programs. Formal verification is indeed recommended to provide the highest level of assurance in the international standard Common Criteria for Information Technology Security Evaluation for computer security (ISO/IEC 15408).

Formal verification is also useful to verify large mathematical proofs. The proof of the Kepler conjecture by Hales is a famous example. Though his proof was accepted as a theorem in 1998 by the *Annals of Mathematics*, referees said that they were only "99% certain" of the correctness because of a substantial use of computer programs in the course of the proof. Subsequently, Hales started the Flyspeck project in 2003 to formally verify his proof using the Isabelle/HOL and the HOL Light proof assistants. It took eleven years. Other famous mathematicians have recognized the need for formal verification:

> A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail. (Vladimir Voevodsky [Voevodsky, 2014])

The Coq proof assistant is a piece of software used to verify computer programs and mathematics. Coq has been awarded the ACM SIGPLAN Programming Languages Sofware Award and the ACM Software System Award in 2013. Xavier Leroy at Collège de France and his colleagues have been awarded the ACM Software System Award in 2021 for their verification of a C compiler in Coq. Though there are other proof assistants around (Mizar, Isabelle/HOL, PVS, Lean, etc.), none has received so much academic recognition so far. Coq has also been used to formalize mathematics, for example the Four Color Theorem in 2004 [Gonthier, 2008]

```
Theorem four_color (m : map R) : simple_map m -> map_colorable 4 m.
```

the Odd Order Theorem in 2013 [Gonthier et al., 2013]

```
Theorem Feit_Thompson (gT : finGroupType) (G : {group gT}) :
  odd #|G| -> solvable G.
```

the Abel–Ruffini theorem in 2021 [Bernard et al., 2021], etc. These examples are about algebra, the goal of this class is rather about analysis.

You can find online a list of research papers using MathComp. More generally, research papers on proof assistants can be found in the proceedings of the International Conference on Interactive Theorem Proving (ITP) or the ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP). Yet, the usage of proof assistants has been spreading to other conferences in computer science, in particular programming languages.

The author of this memo has been using Coq with colleagues to perform a few experiments, e.g.:

- Formalization of information theory (e.g., Shannon's source and channel coding theorems [Affeldt et al., 2014]), formalization of error-correcting codes [Affeldt et al., 2020b]
- 3D geometry for robotics [Affeldt and Cohen, 2017]
- Formalization of analysis [Affeldt et al., 2018, Affeldt et al., 2020a], measure and integration theory [Affeldt and Cohen, 2022]
- Verification of probabilistic programs [Affeldt et al., 2021, Affeldt et al., 2023]

In particular, the formalization of analysis gave rise to an extension of the MathComp library called MathComp-Analysis. It is available online as open source software (MathComp-Analysis) and this will be the main topic of this class.

1.3 Short Presentation of Coq

It is maybe better to follow this introduction using a proper installation of the Coq proof assistant, see Appendix B. Do not worry too much about details, we will come back to them in the next two chapters.

The Coq proof assistant is essentially a programming environment in which one can write functions. Let us define the addition of natural numbers (encoded in unary) using the Gallina language of the Coq proof assistant:

```
From mathcomp Require Import ssreflect ssrfun ssrbool eqtype.
From mathcomp Require Import ssrnat.

Fixpoint add n m :=
  if n is n'.+1 then (add n' m).+1
  else m.
```

When writing Coq code using the theory `X`, it is better to go to the file `X.v`, copy-paste the header (`Require Import`s, etc.), and add `Require Import X`. This should explain the first two lines above.

We can compute the result of, say, $2 + 3$:

```
Compute add 2 3.
(* = 5 : nat *)
```

What will turn out to be very important is the *type* of expressions. Each expression has a type. The type of 0, 1, 2, etc. is `nat`, the type of natural numbers. The type of `add` is:

```
About add.
(* add : nat -> nat -> nat *)
```

In other words, `add` is a function that takes two natural numbers and returns a natural number. More precisely, `add` is a function that given a natural number returns a function that takes a natural number and returns a natural number, so that `add 2` actually makes sense as an expression:

```
Check add 2.
(* add 2 : nat -> nat *)
```

In the simple case of the addition of natural numbers, the type information should not be surprising since it is what one can find in most typed programming languages.

Instead of natural numbers, let us consider the data structure of lists (notation: `[:: a; b; c]`) and their concatenation as implemented by the function `cat` (which is coming from the file MathComp `seq.v`):

```
From mathcomp Require Import seq.

Compute cat [:: 1; 2; 3] [:: 4].
(* = [:: 1; 2; 3; 4] : seq nat *)
```

```
About cat.
(* cat : forall {T : Type}, seq T -> seq T -> seq T *)
```

We can observe that, even though we computed the concatenation of lists of natural numbers, the `cat` function is not restricted to numerical types in particular. It can handle any type: the type of `cat` is parameterized by a type `T` that is generic. `Type` is a type provided by the GALLINA language to mean "any type". We say that `cat` is *polymorphic*.

We actually brought up the example of lists to explain another, more important feature of the COQ proof assistant: *dependent types*, i.e, types that depends on functions' inputs. The basic example used to illustrate dependent types is the type of "fixed-size lists"; you can think also of the type of vectors in algebra. Let us ignore for now how it is implemented and only look at the construct `[tuple of xyz]` that turns a list `xyz` into a fixed-size list (a `tuple` in MATHCOMP parlance).

```
From mathcomp Require Import tuple.

Check [tuple of cat [:: 1; 2; 3] [:: 4]].
(* [tuple of [:: 1; 2; 3] ++ [:: 4]] : (3 + 1).-tuple nat *)
```

Thanks to the type of fixed-size lists, the type system of COQ displays the size of the list in its type of the form `n.-tuple nat`. This is already a form of program verification in the sense that one can use the type system to verify the output of the `cat` function:

```
Check cat [:: 1; 2; 3] [:: 4] : 4.-tuple _.
Fail Check cat [:: 1; 2; 3] [:: 4] : 3.-tuple _.
(* The command has indeed failed with message:
   The term "[:: 1; 2; 3] ++ [:: 4]" has type "seq nat"
   while it is expected to have type "3.-tuple ?T". *)
```

The fact that the above command succeeds is a proof that the result has the right size.

The property of the `cat` function that we just used can be expressed in general terms:

```
Check (fun n m (x : n.-tuple nat) (y : m.-tuple nat) => [tuple of cat x y]).
(* : forall n m : nat, n.-tuple nat -> m.-tuple nat -> (n + m).-tuple nat *)
```

The `forall` expression indicates that the return type is depending on inputs. This notation suggests that types can be used to represent lemmas.

Indeed, let us work out some proof now. The following expression is a valid type (`[::]` is a notation for the empty list):

```
Check forall l : list nat, cat [::] l = l.
(* forall l : seq nat, [::] ++ l = l : Prop *)
```

Like `Type`, `Prop` is a Coq type. `Type` and `Prop` are (essentially) the only types provided by default by Coq. `=` is a binary predicate for equality. We will come back to them later.

The following type is also valid:

```
Check forall l : list nat, cat l l = l.
```

The statement is valid, that does not mean that it is true.

We can use Coq to discriminate between propositions that hold and propositions that do not hold. It is simply by providing a term that has the appropriate type. For example:

```
Check (fun l => erefl) : forall l : list nat, cat [::] l = l.
```

So, `fun l => erefl` (whatever it is) is a proof that the statement

```
forall l : list nat, cat [::] l = l
```

is true. In contrast, it is not a proof of `forall l : list nat, cat l l = l`:

```
Fail Check (fun l => erefl) : forall l : list nat, cat l l = l.
```

In Coq, we can regard types as lemmas and terms as proofs. This is the basic idea of the Curry-Howard correspondence we mentioned in Sect. 1.1.

Let us go back to natural numbers. `fun n => erefl` is also a proof that `add 0 n = n`:

```
Check (fun n => erefl) : forall n, add 0 n = n.
```

But it is not a proof for `add n 0 = n`:

```
Fail Check (fun l => erefl) : forall n, add n 0 = n.
```

What could be a proof then? Here is one:

```
Check (nat_ind (fun n => add n 0 = n) (erefl 0)
  (fun n (ih : add n 0 = n) =>
    eq_trans (f_equal (fun f => f (add n 0)) (erefl S)) (f_equal S ih)))
  : forall n, add n 0 = n.
```

Whatever that term is, it should be clear that it is not going to be practical to require a user to write such programs to serve as proofs.

In the Coq proof assistant (actually in most proof assistants), proofs are written incrementally using *tactics*. Tactics are provided to the proof assistant in the form of *scripts*. Here is a script (a one-liner) corresponding to the proof above:

```
Lemma addn0 n : add n 0 = n.
Proof. by elim: n => //= n ->. Qed.

About addn0.
(* addn0 : forall n : nat, add n 0 = n *)
```

We will explain the tactics in the next chapter but you can already guess that, even though it is definitely shorter than providing a term, it is still going to be a bit technical.

Let us conclude with a proof that the addition of natural numbers is commutative. It requires one more intermediate lemma:

```
Lemma addnS m n : add m n.+1 = (add m n).+1.
Proof. by elim: m => //= m ->. Qed.

Lemma addC m n : add m n = add n m.
Proof. by elim: m n => [n|m ih n]; rewrite ?addn0 //= ih -addnS. Qed.
```

This small example shows that to prove the commutativity of the addition of natural numbers `addC`, we needed already two lemmas: `addn0` and `addnS`. When developing a theory of lemmas, which lemma should be proved? How should they be organized in files? How should they be named and documented? There are going to be a lot of them. In fact, when developing a formal theory, many problems are not so much about the proof, because after all its main idea is already known, but rather about problems akin to software engineering.

1.4 The Rest of this Document

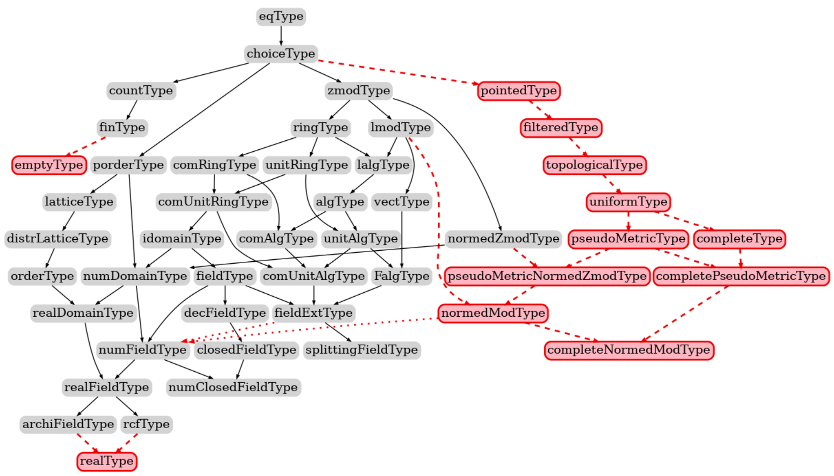
Hopefully, we are going to end the week by looking at the proof of Fubini's theorem. In other words, we are going to produce a proof term whose type will be something like:

```
Context d1 d2
  (T1 : measurableType d1) (T2 : measurableType d2) (R : realType).
Variables (m1 : {measure set T1 -> \bar R})
          (m2 : {measure set T2 -> \bar R}).
Hypotheses (sm1 : sigma_finite setT m1) (sm2 : sigma_finite setT m2).
Variable f : T1 * T2 -> \bar R.
Hypothesis mf : measurable_fun setT f.
Let m := product_measure1 m1 sm2.
Hypothesis imf : m.-integrable setT f.

Theorem Fubini :
  \int[m1]_x \int[m2]_y f (x, y) = \int[m2]_y \int[m1]_x f (x, y).
```

For this purpose, we will explain

- the commands and tactics of the Coq proof assistant that we will use,
- the mathematical structures provided by MathComp (in gray in the figure below), the mathematical structures provided by MathComp-Analysis (in red), as well as new mathematical structures for measure theory: semiring of sets, ring of sets, algebra of sets, σ-algebras, and



- the libraries of definitions, lemmas, notations, etc. that we have used or newly developed.

Chapter 2

Introduction to Coq using SSReflect

Goal of this chapter: This chapter aims at a technical introduction to the Coq *proof assistant using its* SSReflect *extension. We try to favor the information that is the most useful to use the* MathComp *libraries and refer to the* Coq *documentation [The Coq Development Team, 2022] otherwise.*

There exist many introductions to the Coq proof assistant, most notably the standard textbook [Bertot and Castéran, 2004] (whose French translation is available online [Bertot and Castéran, 2015]). Regarding SSReflect tactics, there is [Gonthier et al., 2016, Mahboubi and Tassi, 2021] as well as cheat sheets made available by the developers of SSReflect that might be worth printing.

2.1 The Languages of the Coq Proof Assistant

2.1.1 Gallina: the Language of Proofs

Gallina is an extensible language with which we will write formal statements and proofs. It is an implementation of a variant of the typed λ-calculus suitable for constructive reasoning. The core GALLINA terms are (approximate syntax):

`t`	:=	`Prop` | `Set` | `Type`	sorts
	|	`x, A`	variables
	|	`_`	triggers type inference
	|	`forall x : A, B`	dependent product
	|	`A -> B`	non-dependent product
	|	`fun x => t`	function abstraction
	|	`let x := t1 in t2`	local definition
	|	`t1 t2`	function application
	|	`c`	constant
	|	`match t1 with` *pattern* `=> t2 end`	patter-matching
	|	`fix f x : A := t`	anonymous fixpoint

`Type` is actually one identifier hiding a hierarchy (in the sense of subtyping) of types ($\texttt{Type}_1$, $\texttt{Type}_2$, etc.). We can see the actualy indices by using `Set Printing Universes` but that is rarely needed.

`Type` is *predicative*, so that, for example, one can write

```
(forall A : Type, A) : Type
```

but then if the first `Type` is $\texttt{Type}_1$, then the second `Type` should be, say, $\texttt{Type}_2$. See [Affeldt, 2017, slide 58] for a type derivation.

`Prop` is *impredicative*, so that for example:

```
(forall A : Prop, A -> A) : Prop
```

See [Affeldt, 2017, slide 58] for a type derivation.

`Prop` is intuitively the type of propositions (and predicates). Strictly speaking it is different from the type of boolean numbers (see Sect. 2.4.1). We can however extend the system to blur this difference (see Sect. 3.4.1 and Sect. 4.1) but, in general, the user should be aware that `Prop` and `bool` are different in COQ.

`Set` can be understood as $\texttt{Type}_0$. There is a COQ option to make `Set` impredicative which is occasionally useful (see [Affeldt and Nowak, 2021] for example) but this has not been a concern so far with MATHCOMP and MATHCOMP-ANALYSIS.

There are two useful notations in MATHCOMP for functions. A function that ignores its first argument can be write `fun _ => xyz` or `fun=> xyz`. The notation `f ^~ x` is a replacement for `fun y => f y x`.

The rest of the syntax is similar to programming languages from the ML family.

The user can augment this syntax with new constants using definitions (`Definition`, see Sect. 2.1.2), inductives (`Inductive`, See Sect. 2.4), and notations (see Sect. 2.3.2).

2.1.2 Vernacular: the Language of Commands

The VERNACULAR is a language of commands to organize GALLINA terms. In this document, these commands will appear in dark magenta and will be explained along the way.

The `Definition` command binds a term to an identifier. The term is visible and can be used to perform computation. We say it is *transparent*. See `Definition` in the Coq manual.

The `Lemma` command binds a type to an identifier. There is a term that is built incrementally by the tactics but it is kept hidden, not available for computation. We say it is *opaque*. `Lemma` is essentially a special case of `Definition`. See `Lemma` in the Coq manual.

The language of tactics should also maybe be counted as a third language. It will however be introduced incrementally along this chapter.

2.2 Interactive Proof

In a proof assistant (and in Coq in particular), it is maybe better not to think of a proof as something static, that you read, like a book. It is maybe better to think of it as something dynamic, that you interactively execute, and maintain in the long run. Of course, it can also be read, but maybe rather to get the gist of the proof or to retrieve useful information such as the key lemmas or the main technique (proof by contradiction, generalization of the induction principle, etc.).

A proof starts with a statement. At first, the statement is not yet proved, so it appears as a *goal*. Then the user writes a script, and the script shall fulfill the goal upon execution. Under the hood, the script is actually building a monolithic term by incrementally opening and closing subgoals like branches on a tree.

What the interface displays is only the current state which is a *local context* and a subgoal. The goal itself appears as a stack of hypotheses ended by a conclusion. This distinction between local context, stack of hypotheses, and conclusion is important to use tactics efficiently (in particular, SSReflect's ones).

Let us use the following notations to describe a goal. In particular, $\mathfrak{T}$ corresponds to the top of the stack of hypotheses:

$\mathfrak{h}_1$: $\mathfrak{t}_1$
$\mathfrak{h}_2$: $\mathfrak{t}_2$
...
====================
$\mathfrak{T}$ -> $\mathfrak{T}_1$ -> $\mathfrak{T}_2$ -> ... -> $\mathfrak{C}$

Example: Basic Propositional Logic Proving a lemma means to provide a term whose type reads as the wanted statement. Let us consider the following example: for all propositions A, B, C, $(A \to B \to C) \to (A \to B) \to A \to C$.

This can be proved by providing directly a Gallina term:

```
Definition axiomS (A B C : Prop) : (A -> B -> C) -> (A -> B) -> A -> C :=
  fun f g a => f a (g a).
```

Obviously, formal proof using `Definition` and GALLINA will not scale, hence the use of tactics.

The `move` Tactic

Used in isolation, `move` does almost nothing.

`move` can be combined with the *tactical* `=>` (this is not the same arrow as in `fun=> ...`) to pop objects from the stack of hypotheses. From the point of view of the λ-calculus, using the `move=>` tactic corresponds to having a λ-abstraction to the proof term. `move` can also be combined with the tactical `:` to push objects to the stack of hypotheses.

- `move=> h` pops $\mathfrak{T}$ and adds `h`:$\mathfrak{T}$ to the local context.
- `move=> _` removes $\mathfrak{T}$. More precisely, it removes it after the complete tactic has been executed. This is why you sometimes see tactics like `move=> _ ->` that seemingly delete something and then rewrite it, in fact deletion is not performed right away.
- `move:` $\mathfrak{h}_1$ puts $\mathfrak{t}_1$ as $\mathfrak{T}$ (and removes $\mathfrak{h}_1$:$\mathfrak{t}_1$ from the local context, this is often what the user wants).
- `move: (`$\mathfrak{h}_1$`)` puts $\mathfrak{t}_1$ as $\mathfrak{T}$ *but does not remove* $\mathfrak{h}_1$ *from the local context.*
- `move=> /h` applies the lemma `h` to $\mathfrak{T}$.
- `move=> /[dup]` duplicates $\mathfrak{T}$.
- `move=> /[swap]` swaps $\mathfrak{T}$ and $\mathfrak{T}_1$.
- `move=> /[apply]` replaces $\mathfrak{T}$ and $\mathfrak{T}_1$ by ($\mathfrak{T}$ $\mathfrak{T}_1$).
- `move=> /(_ x)` replaces $\mathfrak{T}$ with $\mathfrak{T}$ `x`.
- `move=>`{$\mathfrak{h}_1$} removes $\mathfrak{h}_1$:$\mathfrak{t}_1$, this is called a *clear-switch*.

That is essentially it about `move`.

Exercise 2.2.1. Correct the spelling:

```
Lemma ISAAC I S A C : I -> A -> S -> C.
```

Exercise 2.2.2. Correct the spelling:

```
Lemma ISAAC I S A C : I -> (I -> A) -> S -> C.
```

The `apply` Tactic

From the view point of the λ-calculus, the `apply` tactic corresponds to function application. When used alone, `apply` applies $\mathfrak{T}$ to $\mathfrak{T}_1$`-> ... ->` $\mathfrak{C}$ (so that $\mathfrak{T}$ ought better be a function). If you want to apply the lemma `h` to the conclusion, you use `apply: h`. You can often omit the `:` and use `apply h` but this is not recommended in general (MathComp expects `apply:`, in some advanced cases we risk performance problems).

Here is an example of interactive proof:

```
Lemma axiomS (A B C : Prop) : (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
move=> f.
move=> g.
move=> a.
apply: f.
apply: a.
apply: g.
apply: a.
Qed.
```

`exact` is a variant of `apply` that must prove the current goal. This is a *terminating tactic*. Terminating tactic appear in red.

Exercise 2.2.3. Prove `forall P Q R : Prop, (P -> Q) -> ((Q -> R) -> (P -> R))`.

2.3 Discoverability of Definitions and Lemmas

In practice, one spends a lot of time searching for existing lemmas to build a proof. It is therefore important to quickly learn how to explore the existing definitions and lemmas.

2.3.1 Checking Existing Lemmas

The `Print` command prints the term corresponding to an identifier. This is the command that we want to use to see the body of a function given its name.

`Check t : T` checks that term `t` has type `T`. `Check t` prints the type of the term `t`. This is the command to use to see the statement of a lemma given its name.

`About t` provides information about the name `t`. It is more informative than `Check t`. In particular, it provides information about *implicit arguments* (see Sect. 2.10) that is often useful to understand how to apply the lemma to arguments.

2.3.2 Searching for Lemmas and Notations

Search is performed using the command `Search`.

Once naming conventions are understood, searching using substrings of the name of a lemma is very effective. Concretely, `Search "abc" "d".` returns the lemmas with names such as `*abc*d*` or `*d*abc`. See Sect. 3.3 for naming conventions in MathComp.

Searching can also be done using patterns to represent the shape of a lemma. It is therefore important to anticipate what can be the shape of the lemma searched for. Here again it is useful to know the naming conventions. Let us assume that the theory of natural numbers has been imported. What are the lemmas using the expression "$2 \times \_$"?

```
Search (2 * _).
```

does not give much information but it indicates the existence of a `.*2` notation that gives better results:

```
Search (_.*2).
```

Another example: How to find the right-distributivity of multiplication over addition of natural numbers?

```
Search (_ * (_ + _)).
```

does not give much information. The right way to look for it is by knowing the naming conventions (see Sect. 3.3):

```
Search "mul" "D".
(* mulnDr: right_distributive muln addn
   mulnDl: left_distributive muln addn *)
```

`Locate` can be used to search for the location of an identifier, i.e., the file in which it has been defined. After having discovered the file, you might want to look at its contents. See Table 2.1 for a few files from the Coq distribution worth looking at.

In general, when Coq displays symbols that are not characters, this is a notation. For example, the type of pairs is as follows:

```
Inductive prod (A B : Type) : Type := pair : A -> B -> A * B.
```

Here, `A * B` is a notation for `prod A B`.

`Locate` can also be used to look for the term behind a notation `Locate "xyz"` looks for a notation that has `xyz` as part of it. For example:

```
Locate "*".
(* ...
   Notation "x * y" := (prod x y) : type_scope
   ... *)
```

Another example: What is behind the `.*2` notation we saw just above?

```
Locate ".*2".
(* Notation "n .*2" := (double_rec n) : nat_rec_scope
   Notation "n .*2" := (double n) : nat_scope (default interpretation) *)
```

Coq libraries	
Init/Datatypes.v	nat, list, etc.
Init/Logic.v	eq_refl, True, etc.
Init/nat.v	nat, sub, etc.
Logic/Classical_Prop.v	axioms for classical reasoning, etc.
...	...
SSReflect libraries	
ssr/ssrfun.v	injective, etc.
ssr/ssrbool.v	notations for boolean numbers, etc.
ssrmatching/ssrmatching.v	LHS, RHS
...	...

Table 2.1: Some files of interest in the Coq source code (directory `coq/theories`)

You can observe that notations belong to *scopes* and in case of conflicting notations this is the latest opened scopes that win. So before starting a proof, one has to open the right scopes in the right order.

If you think that notations are making it harder to understand the current goal, you can disable the display of notations by using:

```
Unset Printing Notations.
```

The user can create new notations but designing a notation in a proof assistant is not an easy task: which ASCII/unicode symbols should we use? what are the right precedence levels? what will happend behind the scene? etc. In general, when one declares a new notation, one needs to choose a precedence level and a scope. It is often useful to check the existing notations and their precedence levels. This is can be done using `Print Grammar constr`. See also Syntax extensions and notation scopes in the Coq reference manual.

Summary: When you are facing an unknown notation or definition, you should locate it, check for its terms, and maybe also search for related lemmas. That is one way to discover formal libraries.

Exercise 2.3.1. What is the ++ notation and where is it defined?

2.4 Inductive Types

The user is not limited to the types provided by default by Gallina (Sect. 2.1.1). The command `Inductive` adds to Coq constants (a new type and *constructors* for objects of this type) as well as induction principles (automatically generated and proved). Use `Variant` when the induction principles are not needed.

2.4.1 Boolean Numbers

Definition of boolean numbers:

```
Inductive bool : Set :=
  true  : bool
| false : bool.
```

This introduces the type `bool` of boolean numbers with two constructors `true` and `false`.

We can do pattern matching with inductive types, e.g.:

```
Definition negb := fun b : bool => if b then false else true.
```

This is actually a notation for:

```
Definition negb := fun b : bool =>
  match b with
    true => false
  | false => true
  end.
```

Boolean connectives: `negb` (notation `~~`), `andb` (notation `&&`), `orb` (notation `||`), `implyb` (notation `==>`), etc.

2.4.2 Proof by Case Analysis

The existence of several ways to construct the same type calls for *case analysis*.

The `case` Tactic

The `case` tactic performs case analyis. Given `b : bool`, `case: b` (which can be understood as `move: b; case`, `;` is a tactical to chain tactics in sequence) creates two subgoals: in the first one, `b` has been replaced by `true`, in the second one, `b` has been replaced by `false`. Used alone, `case` applies to $\mathfrak{T}$.

It is possible to use the `=>` tactical to perform case analysis. `case` is actually equivalent to `move=> [... | ... | ... | ...]` where the number of ... is equal to the number of constructors (so no `|` when there is only one constructor). The tactic creates as many subgoals as there are constructors. This is an example of *intro-pattern*. [1]

Exercise 2.4.1. Prove `forall b, b || ~~ b`,
`forall b, ( ~~ ~~ b) ==> b`,
`forall a b, ((b ==> a) ==> b) ==> b`,
`forall a b, (~~ b ==> ~~ a) ==> (a ==> b)`.

2.4.3 Natural Numbers

Definition of natural numbers:

[1] More generally, the complete syntax for case analysis is: `case:` *d-item+* / *d-item** where *d-item* can be a term, an occurrence switch, or a clear switch (i.e., not an intro-pattern). It is advanced usage.

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
```

The constructors really are capital letters: `O` and `S`. Observe that the type of `S` is defined using `nat`: inductive types can be recursive.

Along with the definition `nat`, `O`, and `S`, Coq generates the `nat_ind` induction principle:

```
Check nat_ind.
(* nat_ind : forall P : nat -> Prop,
     P 0 ->
     (forall n : nat, P n -> P n.+1) ->
     forall n : nat, P n *)
```

This is a term whose type is the induction principle over natural numbers. Coq actually proves a standard induction principle for each defined inductive type. We will see in Sect. 2.4.5 how to use it.

In MathComp, `S n` appears as `n.+1` (`.+1` is a notation). There are also the `.+2`, the `.+3`, etc. notations. Strictly speaking, the notation `+.1` is not part of the Coq distribution since it comes from a file of MathComp: `ssrnat.v` (see Sect. 3.4.3). We are however anticipating on the next chapter because `ssrnat.v` is so much more practical than the theory of natural numbers from the Coq standard library.

Advanced Induction Principles Note that one can also defined mutually recursive inductive types in which case the induction principle needs to be defined by the user using the `Scheme` command. This will not be relevant in this document, see [The Coq Development Team, 2022]. As for strong induction, MathComp will provide a solution in the next chapter (Sect. 3.4.3).

2.4.4 Recursive Functions

The existence of recursive inductive types calls for recursive functions. They can be written with the `Fixpoint` command.

The functions that one can write in Coq need to be terminating. When the recursion is *structural* (i.e., it can be decided by a syntactic criterion), the system detects termination automatically. Otherwise, one needs to resort to Coq extensions like `Equations` [Sozeau, 2009] or more generally the `Program`/`Fix` approach (see, e.g., [Saito and Affeldt, 2022, Sect. 3.1] for details). This is occasionally useful but we will not need it in this document.

Here is the example of Fibonacci numbers:

```
From mathcomp Require Import ssreflect eqtype ssrbool ssrnat.

Fixpoint fib n :=
  if n is n'.+1 then
    if n' is n''.+1 then
```

```
      fib n' + fib n''
    else
      1
  else
    1.

Compute fib 5.
```

Note the `if ... is ... then ... else ...` notation to perform pattern-matching in only one branch.

Here is a variant of Fibonacci numbers borrowed from [Appel, 2022]:

```
From mathcomp Require Import ssreflect eqtype ssrbool ssrnat.

Fixpoint loop n a b :=
  if n is n'.+1 then
    loop n' (a + b) a
  else
    b.

Definition fastfib n := loop n 1 1.

Compute fastfib 5.
```

2.4.5 Proof by Induction

The `elim` Tactic

The `elim` tactic applies an induction principle to the goal. Its syntax is similar to the `case` tactic (Sect. 2.4.2). It looks at the type of its parameter to decide the induction principle to apply, so that `elim: n` performs a proof by induction on natural numbers when `n : nat`.

Exercise 2.4.2. Prove `forall n, n < 2 ^ n`.

Exercise 2.4.3. Prove `forall n, n ^ 2 < 3 ^ n`.

2.5 List Data Structures

2.5.1 Lists

Lists are an example of inductive type with a *parameter*:

```
Inductive list (A : Type) : Type :=
  nil : list A
| cons : A -> list A -> list A.
```

Observe that the parameter is shared by all constructors. We will come back to lists in Sect. 3.4.5

2.5.2 Vectors

Inductive types can also have *indices*: a "parameter" that changes for each constructor. Here is for example the type of vectors from the Coq standard library:

```
Inductive t A : nat -> Type :=
  |nil : t A 0
  |cons : forall (h:A) (n:nat), t A n -> t A (S n).
```

This type is notoriously difficult to use and it is really just an example.

Inductive types with indices can also be referred to as *type families*.

2.6 The Leibniz Equality and Rewriting

Until now, inductive types were defining data structures. We now look at inductive types defining propositions.

Equality is defined as an inductive type:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : x = x
```

This defines the predicate "being equal to `x`" and the only proof is `eq_refl`. Observe that the index is not used.

The corresponding inductive principle is (defined in `Logic.v`[Coq]):

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
  P x -> forall y : A, x = y -> P y
```

It says that given a proof of `x = y`, what holds for `x` also holds for `y`. This is called the Leibniz equality.

Exercise 2.6.1. Prove the symmetry of equality using `eq_ind` directly (i.e., no `rewrite`).

The `rewrite` Tactic

The `rewrite` tactic is by far the most used tactic [Gonthier and Tassi, 2012]. When `h` is an equality (see Sect. 2.6), The semantics of `rewrite h` is to rewrite the "first" occurrence of the left-hand side in the goal (even if it is not visible). `rewrite -h` rewrites the right-hand side.

We can be more precise about what we want to rewrite by using *occurrence switches*: `rewrite` {3}`h` performs rewriting of the 3rd occurrence of the left-hand side of `h`.

We can use patterns to be even more precise w.r.t. what we want to rewrite with the following syntax: `rewrite [`*pattern*`]h`. There is a number of predefined patterns: `LHS` for the left-hand side of a goal when it is an equality, `leLHS` for the left-hand side of a goal when it is an inequality (this is recent syntax), etc.

We can even use *contextual patterns* to indicate a precise `X` inside a pattern: `rewrite [X in pattern]h` or `rewrite [in X in pattern]h` where *pattern* contains an occurrence of `X`.

See [Gonthier et al., 2016, Sect. 8] or [Mahboubi and Tassi, 2021, Sect. 2.4.1] for more about contextual patterns.

The following idiom isolates some expression `X` in the goal and rewrites it into `t`, generating the equality `(*1*) = t` as a new subgoal:

```
rewrite [X in pattern](_ : _(*1*) = t)
```

`rewrite` can also be used with a number of *simplification operations* ("s-item", [Gonthier et al., 2016, Sect. 5.4]):

- `//`: tries to get rid of easy subgoals
- `/=`: tries to perform reduction in the subgoal
- `//=`: combines both

In fact, simplification operations can also be used with `move` and conversely `move` can also perform rewriting:

- `move=> ->` is equivalent to `rewrite` $\mathfrak{T}$
- `move=> {n}->` is equivalent to `rewrite {n}`$\mathfrak{T}$

This might seem confusing at first but in practice this feels very natural.

Last, `rewrite /identifier` unfold the definition of `identifier` when it is transparent. In practice, you often want to do it in the course of a sequence of `rewrite`s, that it is why it is provided as part of the `rewrite` tactic.

Exercise 2.6.2. Complete the following proof:

```
Lemma fastfibE n : fastfib n = fib n.
Proof.
suff : forall i, i <= n -> loop i (fib (n - i + 1)) (fib (n - i)) = fib n.
  by move/(_ _ (leqnn n)); rewrite subnn.
```

2.7 More Propositional Logic with Coq

Definition of falsity:

```
Inductive False : Prop :=
```

This is not a typo: there is no constructor at all. `~ A` is a notation for `A -> False`.

Exercise 2.7.1. Prove `forall P Q, (P -> Q) -> ( ~ Q -> ~ P)`.

Definition of truth:

```
Inductive True : Prop :=  I : True.
```

Disjunction:

```
Inductive or (A B : Prop) : Prop :=
  or_introl : A -> A \/ B
| or_intror : B -> A \/ B.
```

`\/` is a notation. When the goal is of the form `A \/ B`, the tactic `left` turns the goal into `A`, and `right` turns the goal into `B`. Note that there is also a version of `or` that resides in `Set`:

```
Inductive sumbool (A B : Prop) : Set :=
  left : A -> {A} + {B}
| right : B -> {A} + {B}.
```

We can perform case analysis on a proof of a disjunction with the `case` tactic or with the intro-pattern `[|]` (see Sect. 2.4.2).

Conjunction:

```
Inductive and (A B : Prop) : Prop := conj : A -> B -> A /\ B.
```

`/\` is a notation. Yes, this is really just the `Prop`-`Prop` version of the `Type`-`Type` definition of `prod` we saw page 22. When the goal is of the form `A /\ B`, it is customary to use the `split` tactic. It has the same effect as `apply: xyz` where `xyz` is the only constructor of the inductive type in question. Conversely, to perform case analysis on a proof of a conjunction with the case tactic or with the intro-pattern `[]` (no `|` since there is no additional subgoal to generate, there is only one contructor).

Exercise 2.7.2. Prove the commutativity of the conjunction without using tactics.

Exercise 2.7.3. Prove that $A \wedge B \rightarrow A \vee B$ without using tactics.

Exercise 2.7.4. Prove that $\bot \rightarrow A$ without using tactics.

2.8 Predicate Logic: the Existential Quantifier and Sigma-types

While the universal quantifier is built-in in the logic of Coq, the existential quantifier needs to be defined:

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  ex_intro : forall x : A, P x -> exists y, P y
```

Here, `A` and `P` are parameters. The constructor has two parameters: the witness `x` and a proof that `P` holds for `x`, i.e., an object of type `P x`. This is a *dependent pair*, in the sense that the second projection depends on the first one.

Note that the `P` predicate is `Prop`-valued and that an existential formula is `Prop`-valued. There are variants that are at least as useful:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  exist : forall x : A, P x -> {x : A | P x}
```

This is referred to as a *sigma-type*.

```
Inductive sigT (A : Type) (P : A -> Type) : Type :=
  existT : forall x : A, P x -> {x : A & P x}
```

Exercise 2.8.1. Prove:

```
Lemma ex_ex_ex (X : Type) (A B : X -> Prop) :
  (exists x, A x /\ B x) -> ((exists x, A x) /\ (exists x, B x)).
```

`exists2 x : A, P x & Q x` is equivalent to `exists x : A, P x /\ Q x` but is defined in such a way that its case analysis can be done in only one step.

2.9 Views

Equivalences are very important and are therefore given a special treatment. There are equivalences between two propositions in `Prop` (`P <-> Q`, which is a notation for `(P -> Q) /\ (Q -> P)`) and equivalence between a proposition in `Prop` and a proposition in `bool` (contributing to blurring the difference between `Prop` and `bool`). In the latter case, the equivalence is expressed using a `reflect` predicate:

```
Inductive reflect (P : Prop) : bool -> Set :=
  ReflectT : P -> reflect P true
| ReflectF : ~ P -> reflect P false.
```

What happens when one does a case analysis on a proof of `reflect P b`? Case analysis is happening w.r.t. `b`.

```
Lemma test (P : Prop) (b : bool) : reflect P b -> P = b.
Proof.
case.
```

```
P : Prop
b : bool
============================
P -> P = true

goal 2 is:
~ P -> P = false
```

Coq

Here is an example of `reflect` taken from `ssrbool.v`:

```
Variable b1 b2 : bool.
Lemma andP : reflect (b1 /\ b2) (b1 && b2)
```

One can apply the view to the goal by using `apply/view` or `exact/view` or to the top of the stack of hypotheses by using `move/view`. For example:

```
Lemma andbC' (a b : bool) : a && b -> b && a.
Proof.
```

```
a, b : bool
============================
a && b -> b && a
```

```
move=> /andP.
```

```
a, b : bool
============================
a /\ b -> b && a
```

This way, we have switched from a boolean view to a `Prop` view where we can use `case`, `move=>`, etc.

2.10 Implicit Arguments

A term can have several parameters and some of them can sometimes be inferred from others. It would be cumbersome to ask the user to provide systematically all the arguments in such cases. It is therefore possible to declare some arguments as *implicit*: the user does not need to provide them explicitly and Coq synthesizes them automatically.

Let us look again at the type of lists (Sect. 2.5.1):

```
About cons.
(* cons : forall {T : Type}, T -> seq T -> seq T *)
```

The parameter `T : Type` is indicated into curly brackets `{ ... }` to indicate that it is implicit. It means that Coq will try to infer the parameter `T : Type` given the other arguments. The reason why the following command succeeds:

```
Check cons 0 nil.
(* [:: 0] : seq nat *)
```

It is because we are in a context where Coq can figure out that `0` is actually a natural number. In case of doubt, one can disable this automatic inference using the `@` prefix:

```
Check @cons nat 0 nil.
```

or

```
Check @cons _ 0 nil.
```

Recall that the underscore `_` is a place holder that Coq tries to fill using type inference.

An implicit argument is *strict* when it is inferable from the type of some other arguments (this is the case of `T` in `cons`).

Implicit arguments may also be marked by square brackets `[ ... ]`. This means that the implicit argument is *non-maximally inserted*.

By default, the user has to distinguish itself between implicit and non-implicit arguments by using parentheses, curly or square brackets. The setting of implicit arguments can be decided globally so that Coq decides automatically which arguments are implicit. That is why most MathComp files start with the following commands:

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

`Set Implicit Arguments` sets automatically as implicit arguments that can be detected as such. Thanks to `Unset Strict Implicit`, even non-strict arguments are marked as implicit. `Unset Printing Implicit Defensive` is to simplify the display of arguments.

When in some case the setting of implicit arguments turns out to be wrong, this can be adjusted using the command `Arguments`, e.g.:

```
Arguments cons [T].
About cons.
(* cons : forall [T : Type], T -> seq T -> seq T *)
```

See the Coq reference manual on Implicit arguments for more details. In particular, for an example of an implicit argument that is not strict.

2.11 Script Management

IMPORTANT **Terminating tactics** When a tactic solves a subgoal, it is important for maintenance to mark it as such. `exact` is a terminating tactic in itself but `rewrite`, `apply` for example are not. In such cases, the `by` tactical should be used. This way, when a script breaks, it will break as soon as possible, usually at a point where it is easier to fix.

IMPORTANT **Indentation** Indentation is two spaces and it must be done such that the level of indentation at any time indicates the number of subgoal still to be solved. Bullets (`-`, `+`, `*`) are also available to structure scripts it à la org-mode[2].

Reorganizing subgoals The application of a tactic can generate several goals. It is often convenient to get rid of easy subgoals in priority. This is often true after a `rewrite` and this can sometimes by just a matter of using a simplification operation. More generally, it is customary to end the tactic with `; last first.` so as to bring the subgoal upfront (if there is only one, use `last 2 first` if there are two, etc.).

[2] `https://orgmode.org/`

Forward reasoning On paper, mathematical proofs are often performed by *forward reasoning*: intermediate facts are proved incrementally to reach a goal. *Backward reasoning* is then rather when we massage the goal we want to prove. The basic tactic to do backward reasoning is `apply` and since it is such a primitive tactic the user might be tempted to use backward reasoning more than necessary. The main tactic that implements forward reasoning is `have`.

```
have H : statement.
  (* prove statement here*)
(* continue with H : statement in the local context *)
```

The `have` tactic is also often used to apply a lemma directly:

```
have := lemma_instance.
```

Note the `:=` syntax. With this tactic, $\mathfrak{T}$ is now `lemma_instance` and the proof can go on by using `move`, etc. It is possible (and better) to put intro-patterns between `have` and `:=`.

The tactic `pose` can be used to introduce definitions locally inside a script (see [Gonthier et al., 2016, Sect. 4.1]). `set` also introduces a local definition but it does that by pattern-matching an expression in the goal or in the local context (see [Gonthier et al., 2016, Sect. 4.1]).

Factorize Arguments with `Section` and `Variable`

Theories (sets of lemmas that share common parameters) are better organized inside *sections* using the commands `Section`/`End`. Common parameters are introduced at the top of the section using `Variable`, `Hypothesis`, or `Context`. When used inside `Section`/`End`, `Let` acts like `Definition` inside the section but it is unfolded outside. See also [Mahboubi and Tassi, 2021, Sect. 1.4].

Chapter 3

Introduction to the MathComp Library

Goal of this chapter: In this chapter, we review the parts of the MathComp *library that are the most useful for* MathComp-Analysis. *It is mostly about algebra, not yet about analysis.*

3.1 Useful Notation Scopes in MathComp

Mathematics relies a lot on notations. It is therefore no surprise that MathComp relies a lot on Coq's notations (see Sect. 2.3.2).

Notations are organized in scopes. MathComp provides several notation scopes. For example, depending on the scope, `+` does not have the same meaning. `(_ + _)%N` is the addition of natural numbers, `(_ + _)%R` is the addition of rings, etc.

Notations need not be declared and defined in the same file. For example, notations about natural numbers need not be declared along the definition and the basic theory of natural numbers. The notation `\sum _(i <- r | P) F` belongs to the scope `nat_scope` but is declared along with iterated operations (Sect. 3.4.7). See Table 3.1 for examples of scopes.

3.2 Generic Definitions and Notations

MathComp introduces a number of generic definitions so that definitions are more uniform. For example, the property of right identity is captured by the generic definition `right_id e op`, meaning that for the binary operation `op`, `e` is the neutral element on the right, i.e., `op x e = x`. See Table 3.2.

The downside of such generic definitions is that it complicates search of lemmas for a given operator (as we saw in Sect. 2.3.2).

Scope	Delimiter	Meaning	Where declared
`type_scope`	`type`	product, etc.	Coq `Init/Notations.v`
`bool_scope`	`B`	boolean numbers	Coq `Init/Datatypes.v`
`nat_scope`	`N`	natural numbers	Coq `Init/Nat.v`
`fun_scope`	`FUN`	`\o`, `^~`, `+%R`, `-%R`, etc.	Coq `ssr/ssrfun.v`
`pair_scope`	`PAIR`	projections `.1`, `.2`	Coq `ssr/ssrfun.v`
`seq_scope`	`SEQ`	`[::]` , `[:: ...; ...]`	Coq `ssreflect/seq.v`
`order_scope`	`O`	ordered types	MathComp `ssreflect/order.v`
`big_scope`	`BIG`	iterated operations	MathComp `ssreflect/bigop.v`
`ring_scope`	`R`	ring	MathComp `algebra/ssralg.v`
`int_scope`	`Z`	integers	MathComp `algebra/ssrint.v`

Table 3.1: Examples of scopes used in MathComp

Notations about Functions The fact that the function `f` is (monotonically) non-decreasing is noted

```
{homo f : x y / x <= y >-> x <= y}
```

which means `forall x y, x <= y -> f x <= f y`.

If one uses `mono` instead of `homo`, one gets an equivalence instead of an implication:

```
{mono f : x y / y <= x >-> y <= x}
```

means `forall x y, (f x <= f y) = (x <= y)`.

The `mono` notation can be used with partially applied functions:

```
{mono +%R x : y z / y < z}
```

which means `forall y z, (x + y < x + z) = (y < z)`.

```
{morph f : x y / x + y}
```

means `forall a b, f (x + y) = f x + f y`. For example, here is an alternative statement for the right-distributivity of multiplication over addition of natural numbers:

```
Lemma mulnDr' n : {morph muln n : x y / x + y}.
Proof. exact: mulnDr. Qed.
```

Exercise 3.2.1. What does `Lemma opprD : {morph -%R: x y / x + y : V}` mean? (From MathComp `ssralg.v`.)

`injective f`	`forall x1 x2, f x1 = f x2 -> x1 = x2`
`cancel f g`	`g (f x) = x`
`involutive f`	`cancel f f`
`left_injective op`	`injective (op^~ x)`
`right_injective op`	`injective (op y)`
`left_id e op`	$\texttt{e}\ \boxed{\texttt{op}}\ \texttt{x = x}$
`right_id e op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{e = x}$
`left_zero z op`	$\texttt{z}\ \boxed{\texttt{op}}\ \texttt{x = z}$
`right_zero z op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{z = z}$
`self_inverse e op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{x = e}$
`idempotent op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{x = x}$
`commutative op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{y = y}\ \boxed{\texttt{op}}\ \texttt{x}$
`associative op`	$\texttt{x}\ \boxed{\texttt{op}}\ \texttt{(y}\ \boxed{\texttt{op}}\ \texttt{z) = (x}\ \boxed{\texttt{op}}\ \texttt{y)}\ \boxed{\texttt{op}}\ \texttt{z}$
`right_commutative op`	$\texttt{(x}\ \boxed{\texttt{op}}\ \underline{\texttt{y}}\texttt{)}\ \boxed{\texttt{op}}\ \underline{\texttt{z}}\ \texttt{= (x}\ \boxed{\texttt{op}}\ \underline{\texttt{z}}\texttt{)}\ \boxed{\texttt{op}}\ \underline{\texttt{y}}$
`left_commutative op`	$\underline{\texttt{x}}\ \boxed{\texttt{op}}\ \texttt{(}\underline{\texttt{y}}\ \boxed{\texttt{op}}\ \texttt{z) =}\ \underline{\texttt{y}}\ \boxed{\texttt{op}}\ \texttt{(}\underline{\texttt{x}}\ \boxed{\texttt{op}}\ \texttt{z)}$
`left_distributive op add`	`(x + y) * z = (x * z) + (y * z)`
`right_distributive op add`	`x * (y + z) = (x * y) + (x * z)`
`left_loop inv op`	`cancel (op x) (op (inv x))`

Table 3.2: A few generic definitions in MathComp

3.3 IMPORTANT Naming Conventions

The user needs to be able to find lemmas quickly and to type them quickly. The names of lemmas therefore need to be easy to remember, short, with a uniform and precise format. This also makes the lemmas easier to search for (in the sense of Sect. 2.3.2).

More generally, being picky about names is customary in programming (see for example the Hungarian notation).

MathComp libraries enforce a strict naming convention. There are a few rules to remember.

3.3.1 Properties of Operations

There is a couple of a long and a short identifier for most basic operations. In the name of lemmas, the long identifier is typically used as a prefix when it is the head symbol of the left-hand side. When two operators are involved, the main one is used as a prefix and the other one is referred to by its short identifier, e.g., `D` for the addition. See Table 3.3. Furthermore, there are also one-letter identifiers for standard types. For example, `n` corresponds to natural numbers. See Table 3.4. From these rules, we can guess the names of many lemmas. For example, the right-distributivity of multiplication over addition is `mulnDr` for the natural numbers and `mulrDr` for rings.

When a lemma looks like a rewriting rule (most of them do), the name of the lemma is often prefixed by a long identifier (corresponding to the head symbol

Long identifier	Short identifier	Meaning
`add`	`D`	addition
`sub`	`B`	subtraction
`opp`	`N`	opposite
`mul`	`M`	multiplication
`exp`	`X`	exponentiation (by a natural number)

Table 3.3: Naming Convention: Identifiers for operations

One-letter identifier	Meaning
`n`	natural numbers
`r`	elements of a ring
`f`	elements of a field
`e`	extended real numbers
`y`	∞
`Ny`	$-\infty$

Table 3.4: Naming Convention: Identifiers for positional notation

of the left-hand side) and it is followed by a pattern that corresponds to the shape of the left-hand side. This pattern uses one-letter identifiers. Constants are referred as such (e.g., `0` for the 0 of the addition). Standard types are referred to by their one-letter identifier from Table 3.4. This way, the name of the lemma gives a good idea of what the lemma does. For example, `n0` indicates that the lemma is of the form $n\boxed{\texttt{op}}0$. For illustration, `addn0` should correspond to `forall n, n + 0 = n` and indeed:

```
About addn0.
(* addn0 : right_id 0%N addn *)
(* Print right_id.
fun (S T : Type) (e : T) (op : S -> T -> S) => forall x : S, op x e = x
     : forall S T : Type, T -> (S -> T -> S) -> Prop *)
```

Let us call that this naming scheme the *positional notation*.

There is also a number of one-letter identifiers used as suffixes for properties of operations (see Table 3.5). From this table, we can guess that the associativity of `nat` is `mulnA`, its commutativity is `mulnC`, etc. In particular, the cancellation property is typically written with the generic `cancel` predicate and marked with `K`. The suffix `E` is for rewriting lemmas that "unfold" a term into its definition. See Table 3.5.

3.3.2 Properties of Relations

Table 3.6 does not provide all the information but it is supposed to lead you to valuable lemmas such as:

```
Lemma leq_pmul2l m n1 n2 : 0 < m -> (m * n1 <= m * n2) = (n1 <= n2).
```

Identifier	Property
`A`	associativity
`C`	commutativity
`C`	set complement
`D`	set theoretic difference
`K`	cancellation
`E`	equality

Table 3.5: Naming Convention: Suffixes for the properties of operations

`le`	$\leq$ for ordered types
`leq`	$\leq$ for `nat`
`lt`	$<$ for ordered types
`ltn`	$<$ for `nat`

Table 3.6: Naming Convention: Identifiers for relations

or

```
Lemma ltr_addl x y : (x < x + y) = (0 < y).
```

3.4 About Mathematical Structures

In mathematics, structures are organized as a hierarchy, in the sense that, e.g., a field is defined using a ring. In consequence, an element of a field is also an element of the underlying ring. It is an important issue in formal mathematics to get this inheritance right. A *mathematical structure* consists typically of:

1. a carrier (typically an object in `Set` or `Type`)
2. a set of operations (including constants, i.e., 0-ary operations)
3. the properties of the operations (one also says the "axioms" of the base theory, not to be confused with Coq `Axiom`s which are unproved lemmas)

In proof assistants, the elements of a mathematical structure are declared by an interface. In MathComp, an interface is essentially *record* (see Record types in the reference manual). A record is in fact an inductive type, and since the axioms depends on the operations, which depends on the carrier, it should be obvious that dependent types are here again put at work. An *instance* of an interface is an implementation of this interface. For example, `nat` (and its comparison function `eqn`) is an instance of `eqType` (a type with a decidable equality).

But how should we use record? There are several strategies depending on what we put in parameters. *Bundled*: everything as fields. *Semi-bundled*: only the carrier is a parameter. *Unbundled*: only the axioms are fields. In MathComp, mathematical structures are bundled. However, they are implemented by first providing a semi-bundled record (the "class") and then a bundled record using the class (the "structure"). This is the *packed classed* methodology. To make inference work in presence of inheritance, MathComp uses the mechanism of *canonical structures* of Coq in a clever way [Garillot et al., 2009].

Until very recently, the construction of hierarchies was done by hand and it was error-prone. Today, MathComp uses a dedicated tool called Hierarchy-Builder [Cohen et al., 2020]. Sect. 5.2 will provide an example of usage of Hierarchy-Builder.

The hierarchy of mathematical structures provided by MathComp is displayed in Fig. 3.1. You can also check for [Gonthier et al., 2016, page 65, version 16 not 17)] for an older but easier to read hierarchy.

The main thing to remember is that the definition of a mathematical structure is to be find in interfaces. In MathComp, they are referred to as *mixins*. It is a technical term that comes from object-oriented programming and it can be understood as an interface.

directory	files of interest
`mathcomp/ssreflect`	`ssrnat.v` (Sect. 3.4.3), `seq.v` (Sect. 3.4.5), `order.v` (Sect. 3.4.6), etc.
`mathcomp/algebra`	`ssralg.v` (Sect. 3.5.1), `ssrnum.v` (Sect. 3.5.2), `ssrint.v` (Sect. 3.5.2), `matrix.v` (not a topic in this document), etc.

Table 3.7: Some files of interest in MathComp, see also Table 2.1 for MathComp files distributed with Coq

3.4.1 `ssrbool.v`: Boolean Reasoning

`ssrbool.v`[Coq] is a file that comes with the Coq proof assistant. It contains definitions and lemmas about boolean numbers that we already discussed. See also Appendix A.

To blur the difference between `Prop` and `bool`, `ssrbool.v`[Coq] contains a *coercion* `is_true` that the user should be aware of:

```
(* Definition is_true b := b = true. defined in Init/Datatypes.v *)
Coercion is_true : bool >-> Sortclass.
```

The effect of that command is that instead of printing `is_true b`, i.e., `b = true` when `b` is a boolean number (actually, anything that has the type of a boolean number), Coq displays simply `b`. Because of the heavy use of boolean numbers with MathComp, this makes for clearer statements and goals, but it might surprise you from time to time, so it is good to keep it in mind. In case of doubt, use

```
Set Printing Coercions.
```

to see the coercions.

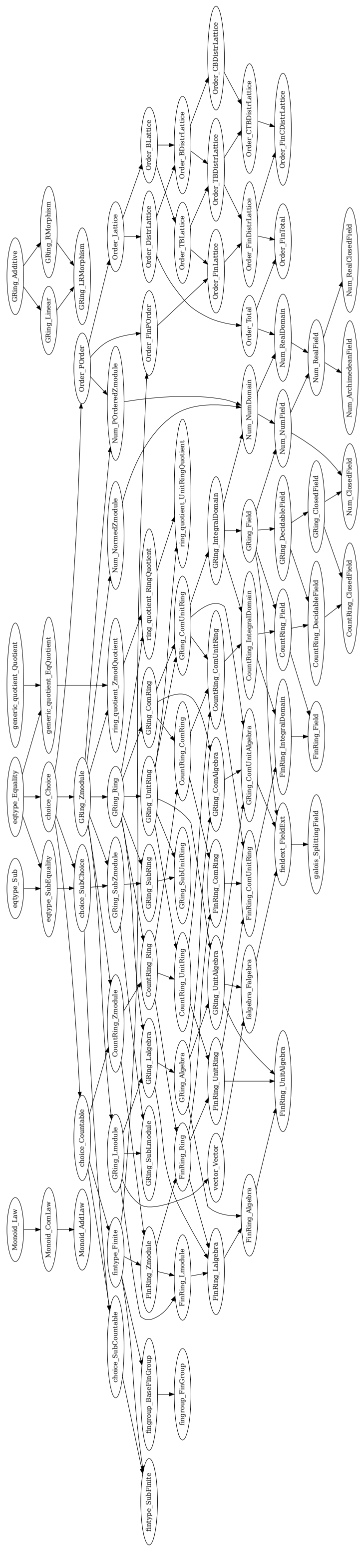


Figure 3.1: The hierarchy of mathematical structures of MathComp as of 2022-08-23 (Hierarchy-Builder version)

A boolean statement `b` can be rewritten using `rewrite` since it is actually `b = true` thanks to the `is_true` coercion.

`ssrbool.v` (Coq) also defines the type of boolean predicates

```
Definition pred T := T -> bool.
```

and a few boolean predicates such as the predicate that is always true:

```
Notation xpredT := (fun=> true).
```

Because of the pervasive use of `if ... then .... else ...`, there is couple of useful lemmas about branching. `ifT` rewrites `if b then t1 else t2` into `t1` and generates `b` as a subgoal. Similarly for `ifF`. To perform a case analysis on a goal that contains an expression, one can do:

```
case: ifPn.
```

`ifPn` is often more appropriate that using `ifP`.

The idiom to do a case analysis on an arbitrary boolean expressiob `b` is:

```
have [|] := boolP b.
```

The implementation of `boolP` is similar to the one of the `reflect` predicate of Sect. 2.9.

There is a family of contraposition lemmas that are very useful, such as

```
contra  : forall [c b : bool], (c -> b) -> ~~ b -> ~~ c.
contraTN: forall [c b : bool], (c -> ~~ b) -> b -> ~~ c
contraTT: forall [c b : bool], (~~ c -> ~~ b) -> b -> c
```

`contraNT` should be easy to guess. Search for `"contra"` to look for an appropriate contraposition lemma when in need.

3.4.2 `eqtype.v`: Decidable Equality

This file contains the most basic mathematical structure in MathComp. It introduces the type `eqType` of types with a decidable equality, i.e., types that can be related with the boolean equality `==`. To find the interface of `eqType`, look for `mixin` in `eqtype.v` (MathComp) (`eqtype.v` on github):

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

Structure mixin_of T := Mixin {op : rel T; _ : axiom op}.
```

So, an `eqType` is a type that has an equality relation `==` that satisfies `axiom` which is a `reflect` relation (Sect. 2.9). The mechanism that associates `op` with `==` is a bit technical (omitted).

Natural numbers and boolean numbers are declared to be `eqType`s by providing the `nat_eqType` and `bool_eqType` instances, so that one can check:

```
Check 0 == 1.
(* 0 == 1 : bool *)
Check true == false.
(* true == false : bool *)
```

This is actually the same as `@eq_op _ 0 1` and `@eq_op _ true false` but in the first case Coq fills the placeholder with `nat_eqType` and `bool_eqType` in the second case. `nat_eqType` is not exactly `nat` but that will not bother Coq. See [Mahboubi and Tassi, 2021, Section 6.3–6.4] for more details.

`eqVneq` is a useful lemma from MathComp `eqtype.v` to do case analysis in the course of proofs with the idiom:

```
have [->|ab] := eqVneq a b.
```

In the first subgoal, `a` is replaced by `b`, in the second subgoal, the local context now contains the hypothesis `ab : a != b`. As a side effect, occurrences of `a == b` and `b == a` are replaced by their truth values.

Case analysis with `eqVneq` is possible this way because it is specified using an inductive predicate with two constructors:

```
Variant eq_xor_neq (T : eqType) (x y : T) : bool -> bool -> Set :=
  | EqNotNeq of x = y : eq_xor_neq x y true true
  | NeqNotEq of x != y : eq_xor_neq x y false false.

Lemma eqVneq (T : eqType) (x y : T) : eq_xor_neq x y (y == x) (x == y).
```

This way of providing case analysis is very common in MathComp, this is actually similar to the `reflect` relation (Sect. 2.9).

3.4.3 `ssrnat.v`: Natural Numbers

MathComp `ssrnat.v` is a file that contains the theory of natural numbers. The proofs it contains are short (many one-liners) and it is a good place to learn and appreciate SSReflect tactics. See also Appendix A.

This file does not redefine totally the operations on natural numbers. For example, subtraction and multiplication are coming from Coq `Init/Nat.v`.

The comparison operations are boolean functions, i.e., they have type `bool`, not `Prop`. In the past, relations were given the type `Prop` in the standard library of Coq but this turned out to be inconvenient in practice. For example, large inequality is formalized as follows:

```
Definition leq m n := m - n == 0.
```

This is the notation `m <= n`. The notation `m < n` is really just `m.+1 <= n`. It might be surprising at first to define inequalities in such a convoluted way, but this enables more sharing of proofs and contributes to simplify the theory.

There is a number of useful lemmas that have been designed to make case analysis more efficient. E.g.:

```
Variant leq_xor_gtn m n : nat -> nat -> nat -> nat -> bool -> bool -> Set :=
  | LeqNotGtn of m <= n : leq_xor_gtn m n m m n n true false
  | GtnNotLeq of n < m  : leq_xor_gtn m n n n m m false true.

Lemma leqP m n : leq_xor_gtn m n (minn n m) (minn m n) (maxn n m) (maxn m n)
                              (m <= n) (n < m).
```

Case analysis on `leqP m n` will generate two subgoals. In the first one, `m <= n` is true and `n < m` is false. In the second one, `m <= n` is false and `n < m` is true. Occurrences of `m <= n` and `n < m` are replaced by their truth values. In addition, `maxn m n`, etc., are also replaced by their adequate values.

Exercise 3.4.1. Prove `leqP`.

Other interesting contents: the notation `.*2` we already saw, the theory of the predicate `odd`, etc.

Related files: Properties about division MathComp `div.v` (`m %/ d`, `m %% d`: quotient and remainder of euclidean division) and prime numbers MathComp `prime.v` (`prime p`: `p` is a prime number)

Strong Induction Recent versions of MathComp suggest to use the predicate `ubnP` to perform proofs by strong induction:

```
Check ubnP.
(* ubnP : forall m : nat, {n : nat | m < n} *)
```

`ubnP` is using the type `sig` from Sect. 2.8.

.../...

Example of proof by strong induction:

```
Fixpoint a n :=
  match n with
  | 0 => 2
  | 1 => 4
  | n'.+1 => a n' + (a n'.-1).*2
  end.

Lemma ha n : 2 ^ n <= a n.
Proof.
have [m nm] := ubnP n.
```

```
n, m : nat
nm : n < m
----------------------------------------
2 ^ n <= a n
```

```
elim: m => // m ih in n nm *.
(* this is the same as move: m n nm; elim=> // m ih n nm. *)
```

```
m : nat
ih : forall n : nat, n < m -> 2 ^ n <= a n
n : nat
nm : n < m.+1
----------------------------------------
2 ^ n <= a n
```

Exercise 3.4.2. Complete the proof of `forall n, 2 ^ n <= a n`.

3.4.4 `fintype.v`: Finite Types

MathComp
`fintype.v` contains a theory of finite types, i.e., types with a finite number of inhabitants. The example of interest for us is the type of so-called *ordinals* `'I_n` where `n` is a natural number (and `'I_` is a notation). It is the type of the natural numbers $\{0, \dots, n-1\}$.

Though conceptually simple, ordinals can be painful to manipulate and should therefore not be used without a good reason. They are however important for iterated operations (see Sect. 3.4.7). In particular, for the type `'I_n.+1` (note the `.+1`), `ord0` is 0 and `ord_max` is `n`. Ordinals are automatically turned into natural numbers if needed thanks to the coercions `nat_of_ord`. Turning a natural number into an ordinal requires a bit of help from the user:

```
From mathcomp Require Import fintype.

Fail Check 0 : 'I_3.
Check inord 0 : 'I_3.
About inord.
(* inord : forall {n' : nat}, nat -> 'I_n'.+1 *)
```

3.4.5 `seq.v`: Lists

This is recap of lists notations (in scope `seq_scope`) and of standard functions about lists:

- ```
  Variables (T1 T2 : Type) (f : T1 -> T2).
  Fixpoint map s := if s is x :: s' then f x :: map s' else [::].
  ```

  Notation: `[seq f i | i <- s]`

- ```
  Variable a : pred T.
  Fixpoint filter s :=
    if s is x :: s' then if a x then x :: filter s' else filter s' else [::].
  ```

  Notation: `[seq i <- s | a i]`

- ```
  Variables (T : Type) (R : Type) (f : T -> R -> R) (z0 : R).
  Fixpoint foldr s := if s is x :: s' then f x (foldr s') else z0.
  ```

  `foldr f z0 [:: a; b; c]` is `f a (f b (f c z0 ))`

- ```
  Fixpoint iota m n := if n is n'.+1 then m :: iota m.+1 n' else [::].
  ```

  So `iota m n` is the list `[:: m; m + 1; ...; m + n - 1]`.

Extended explanations about lists can be found in [Mahboubi and Tassi, 2021, Sect. 1.3.1].

3.4.6 `order.v`: Ordered Types

There are several ordered types, most notably `porderType` and `orderType`. See MathComp `order.v`, beware: this is a huge file (try looking at the right of Fig. 3.1). Theories are organized in modules, so that, for example, to enjoy the lemmas about totally ordered types, the development should start with `Import Order.TTheory`.

Transitivity lemmas are particularly useful. Given a goal of the form:

```
a < K
```

`rewrite (@le_lt_trans _ _ M)//` generates two goals:

```
a <= M
```

and

```
M < L
```

Similarly for `lt_le_trans`, `le_trans`, `lt_trans`. They are (of course) often used in practice.

Similarly to `leqP` (Sect. 3.4.3) for natural numbers, we can do case analysis with any ordered type using:

```
have [|] := leP a b.
```

Although there is a bridge to treat natural numbers as an ordered type, the former have been kept separated for historical reasons, hence the apparent duplication of lemmas.

3.4.7 IMPORTANT `bigop.v`: Iterated Operations

Iterated operations is the formalization of notations such as Σ_i, Π_i, $\cup_i$, etc. They were introduced in MathComp by [Bertot et al., 2008] and were instrumental to complete the formal proof of the Odd Order Theorem [Gonthier et al., 2013].

The first lines of Table 3.8 introduce the generic notation. The implementation `\big[op/idx]_(i <- s | P i) F i` of iterated operations is essentially a wrapper for a `foldr` (Sect. 3.4.5) of a function `F` that represents each term (of the sum, product, etc.) along a list `s`, filtered by a boolean predicate `P` (Sect. 3.4.1).

It is important to understand that in Table 3.8, in `i < n`, `i` is an ordinal (Sect. 3.4.4). In contrast, the enumeration `m <= i < n` is implemented by the `iota` function as `iota m (n - m)` (Sect. 3.4.5), so there `i` is a natural number.

From the user point of view, the lemmas listed in Appendix A are maybe among the most useful ones (e.g., `big1`, `eq_bigr`).

From the developer point of view, it is sometimes useful to use more technical lemmas, that reveal a bit about the formalization of iterated operations. For example, `big_mkcond` is sometimes useful to get rid (temporarily) of the filtering predicate by putting it into the function body:

```
Lemma big_mkcond I r (P : pred I) F :
  \big[*%M/1]_(i <- r | P i) F i =
     \big[*%M/1]_(i <- r) (if P i then F i else 1).
```

Similarly, `big_seq` "duplicates" the enumeration list as a predicate so that it can be available to prove properties of `F` if needed:

```
Lemma big_seq (I : eqType) (r : seq I) F :
  \big[op/idx]_(i <- r) F i = \big[op/idx]_(i <- r | i \in r) F i.
```

There is a number of generic lemmas that are useful for prove properties of iterated operations. For example, to prove a property for a bigop knowing it is true for each element, one can use `elim/big_ind : _ => //` (this is the generic syntax for case analysis that we mentioned in Sect. 2.4.2) where `big_ind` is:

```
big_ind : K idx ->
  (forall x y : R, K x -> K y -> K (op x y)) ->
  forall (I : Type) (r : seq I) (P : pred I) (F : I -> R),
  (forall i : I, P i -> K (F i)) -> K (\big[op/idx]_(i <- r | P i) F i)
```

Reminder: To do the following exercises, you should go to MathComp `bigop.v` and copy the header to a new file (and append `From mathcomp Require Import bigop.`)

Exercise 3.4.3. Prove `forall n, 2 * (\sum_(0 <= x < n.+1) x) = n * n.+1`. (`\sum_` is a notation for `\big[addn/O]_`)

Exercise 3.4.4. Prove

```
forall n : nat, \sum_(0 <= x < n.+1) (x + x) = 2 * \sum_(0 <= x < n.+1) x
```

using `big_ind2`.

Exercise 3.4.5. Prove $1 + 2 + \cdots + 2^n = 2^{n+1} - 1$

Exercise 3.4.6. Prove

```
forall n, (6 * \sum_(k < n.+1) k ^ 2) = n * n.+1 * (n.*2).+1.
```

Exercise 3.4.7. Prove

```
forall (x n : nat) :
  1 < x -> (x - 1) * (\sum_(k < n.+1) x ^ k) = x ^ n.+1 - 1
```

Iterated operations are generic. It suffices for the operation to meet some requirements to enjoy a particular lemma. For example, provided that the carrier with the operation `op` form a monoid (`bigop.v` MathComp)

```
Structure law := Law {
  operator : T -> T -> T;
  _ : associative operator;
  _ : left_id idm operator;
  _ : right_id idm operator
}.
```

the lemmas `big1`, `big_nat_recr` become available (see Appendix A). This is because `addn`, the addition of natural numbers, as been shown to form a monoid

```
Canonical addn_monoid := Law addnA add0n addn0.
```

that one can use these lemmas.

Searching lemmas about iterated operations is notoriously difficult. It is maybe better to look for the most generic notation

```
Search (\big[_/_]_(i <- _ | _) _).
```

or for lemmas with the substring `"big"`.

`\big[op/idx]_(i \in D) f i` (Table 3.8) assumes that `f` takes a finite number of values (note the `\in` notation). This is useful as a notation because it allows to write sums like `\sum_(x \in [set: R]) f x` when the support of `f` is finite, as we will do in Sect. 6.2.1 to define integration. The definition is bit technical, see file `fsbig.v` MathComp-Analysis.

The `under` Tactic With iterated operators the need to use `rewrite` below λ-abstractions became more pressing. The `under` tactic can be used for that purpose [Martin-Dorel and Tassi, 2019]. A common usage with iterated operators is `under eq_bigr do rewrite ...`, with series `under eq_eseries do rewrite ...`.

Finitely iterated operations:	
`\big[op/idx]_(i <- s | P i) f i`	$\boxed{\texttt{op}}_{i<\|s\|, i \in P} f(s_i)$
`\big[op/idx]_(i < n | P i) f i`	$\boxed{\texttt{op}}_{0 \le i<n, i \in P} f(i)$
`\big[op/idx]_(m <= i < n | P i) f i`	$\boxed{\texttt{op}}_{m \le i<n, i \in P} f(i)$
Application to numeric functions:	
`\sum_(i <- s | P i) f i`	$\sum_{i<\|s\|, i \in P} f(s_i)$
Iterated operations over finite supports:	
`\big[op/idx]_(i \in D) f i`	$\boxed{\texttt{op}}_{i \in D} f(i)$ if $f(i)$ has a finite number of values in D s.t. $f(i) \neq$ `idx` o.w. `idx`
Countably iterated sum of numeric functions (see Sect. 4.8):	
`\sum_(i <oo | P i) f i`	$\sum_{i=0, i \in P}^{\infty} f(i)$
`\sum_(m <= i <oo | P i) f i`	$\sum_{i=m, i \in P}^{\infty} f(i)$
Sum of extended real numbers over general sets (see Sect. 5.1):	
`\esum_(i in P) f i`	$\sum_{i \in P} f(i)$

Table 3.8: Summary of iterated operations. The symbol $\boxed{\texttt{op}}$ is the iterated operation corresponding to `op`.

3.4.8 About Finite Sets

MathComp comes with a theory of finite sets *over finite types* (`finset.v`[MathComp]). It is moderately useful[1] outside of MathComp, which has been designed originally to develop the theory of finite groups. See also Appendix A.

The `finmap` library [Cohen and Sakaguchi, 2015] provides an alternative where the carrier type only needs to be a `choiceType`, intuitively a type equipped with a form of the axiom of choice (which extends `eqType` in the hierarchy of mathematical structures in MathComp). This is less restrictive and useful in MathComp-Analysis (for example to define countable sums, see Sect. 5.1).

3.5 Mathematical Structures in `algebra`

3.5.1 `ssralg.v`: Algebraic Structures

Most algebraic mathematical structures can be found in `ssralg.v`[MathComp]. They can also be found in Fig. 3.1. Let us just mention the most important ones.

- `zmodType` for abelian groups. It provides one constant (`0`), one unary operation (`-%R`), one binary operation (`+%R`) (all in `ring_scope`, see Table 3.1), and the axioms of an abelian group (`addrA`, `addrC`, `add0r`, `addNr`). They are in the module `GRing` so to use them one often starts its development with `Import GRing.Theory`. (Otherwise, identifiers should be fully qualified.)

[1] except maybe for the formalization of finite distributions in [Infotheo, 2022]...

- `ringType`: rings, provides one constant (`1`), one binary operation (`*%R`), and the axioms of a ring (`mulrA`, `mul1r`, `mulr1`, `mulrDl`, `mulrDr`, `oner_neq0` for $1 \neq 0$, which means that the trivial ring is excluded).

- `comRingType`: commutative rings, adds `mulrC`

- `lmodType R`: left modules over `R` which have the following mixin:

```
Structure mixin_of (R : ringType) (V : zmodType) : Type := Mixin {
  scale : R -> V -> V;
  _ : forall a b v, scale a (scale b v) = scale (a * b) v;
  _ : left_id 1 scale;
  _ : right_distributive scale +%R;
  _ : forall v, {morph scale^~ v: a b / a + b}
}.
```

  The contents of this mixin should be entirely readable since we have explained in previous sections all the syntax. The notation for the scaling operation it `*:`. Properties are available as the lemmas `scalerA`, `scale1r`, `scalerDr`, `scalerDl`. Left modules will be used to define normed modules in MathComp-Analysis (Sect. 4.6.5).

- `idomainType`: integral domains, with the axiom

```
forall x y : R, x * y = 0 -> (x == 0) || (y == 0)
```

- `fieldType`: fields. Note that the units of a field (more generally of a `unitRingType`) are available though the predicate `unit` (which comes with the `unitRingType`). The properties of units are recovered via lemmas such as:

```
Variable F : fieldType.
Lemma unitfE x : (x \in unit) = (x != 0).
```

  Needless to say, the properties of units will be useful to deal with real numbers in MathComp-Analysis.

3.5.2 `ssrnum.v`: Numeric Types

`ssrnum.v` (MathComp) provide mathematical structures for numeric types. `numDomainType` combines integral domains, ordered types, and a notion of norm (notation `` `| ... | ``). This is the basic numeric type.

The combination of an abelian group with a notion of norm is `normedZmodType` and will also be used to define normed module in Sect. 4.6.5.

As of today, the formalization of these two mathematical structures is a bit technical. This is not where you want to start to read `ssrnum.v` (MathComp). This is also a huge file, organized with modules. To use the definitions and lemmas in `ssrnun.v` (MathComp), developments usually starts with

```
Import Num.Def Num.Theory.
```

`numDomainType` extends to `numFieldType` in a natural way, which extends to `realDomainType` (all elements are non-positive or non-negative), which extends to `realFieldType`, which extends to `archiFieldType`, etc. All these types will be used pervasively in the development of MathComp-Analysis.

Exercise 3.5.1. Show that $\sqrt{4+2\sqrt{3}} = 1 + \sqrt{3}$. Look at `sqrt` in MathComp `ssrnum.v`.

Integers The numeric type of relative integers is provided by the file MathComp `ssrint.v`. Injection of a natural number to an integer: `n%:Z`. Integers have their importance when dealing with real numbers in the next chapter because of the flooring and ceiling functions.

Summary About Numerical Types There are several numeric types in MathComp and going from one to another might sometimes feel painful. That actually seem to be a common defect in proof assistants based on type theory [Harrison, 2018]. Indeed, on paper, we take the following inclusions for granted:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \overline{\mathbb{R}}$$

Fig. 3.2 illustrates some ways to go between numeric types used in MathComp-Analysis.

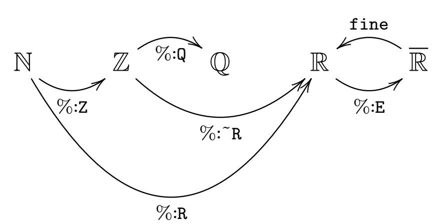


Figure 3.2: Some conversions between numeric types. Anticipating on Sect. 5.1

Chapter 4

Classical Reasoning using MathComp

Goal of this chapter: This chapter introduces the basics of the formalization of analysis in MathComp-Analysis*. It covers material that can be found in [Affeldt et al., 2018, Rouhling, 2019, Affeldt et al., 2020a].*

The MathComp-Analysis [Affeldt et al., 2017] library contains two directories: `classical` and `theories`. The contents of `classical` is generic, it essentially develops classical reasoning on top of MathComp. This is the purpose of the first part of this chapter. The rest of this chapter deals with the topic of convergence, which spans the files MathComp-Analysis `topology.v`, MathComp-Analysis `normedtype.v`, and MathComp-Analysis `sequences.v` from the directory `theories`.

4.1 Axioms Introduced by MathComp-Analysis

MathComp is constructive: it relies solely on Coq, it does not rely on classical reasoning. MathComp-Analysis starts by giving up on constructivism by adding a bunch of axioms, which are known to be compatible with the logic of Coq.

One of the motivation is to be able to do set-theoretic reasoning and to further blur the difference between `Prop` and `bool`.

4.1.1 Propositional Extensionality

The equivalence between two propositions implies their equality:

```
Axiom propositional_extensionality :
  forall P Q : Prop, P <-> Q -> P = Q.
```

Exercise 4.1.1. Prove `True = true`. Similarly, prove `true = True`.

4.1.2 Functional Extensionality

Pointwise equality of functions implies their equality. This is stated more generally for functions with dependent types `forall x : A, B x` (instead of the less general `A -> B` type, remember Sect. 2.1.1).

```
Axiom functional_extensionality_dep :
  forall (A : Type) (B : A -> Type) (f g : forall x : A, B x),
    (forall x : A, f x = g x) -> f = g.
```

Here is an alternative representation of functional extensionality:

```
Lemma eq_fun T rT (U V : T -> rT) :
  (forall x : T, U x = V x) -> (fun x => U x) = (fun x => V x).
```

It can be used together with the `under` tactic (Sect. 3.4.7) to do rewriting under λ-abstractions as in `under eq_fun do rewrite ...`.

4.1.3 Constructive Indefinite Description

The `Prop`-valued existential quantifier implies the `Type`-valued one (see Sect. 2.8).

```
Axiom constructive_indefinite_description :
  forall (A : Type) (P : A -> Prop),
    (exists x : A, P x) -> {x : A | P x}.
```

The existential on the left is in `Prop`, the one on the right is the one in `Type` (as we saw in Sect. 2.8).

4.1.4 Consequences of Classical Axioms

We can derive a version of the axiom of choice which is very useful:

```
Lemma choice X Y (P : X -> Y -> Prop) :
  (forall x, exists y, P x y) -> {f & forall x, P x (f x)}.
```

We can derive the law of the excluded middle:

```
Lemma pselect (P : Prop): {P} + {~P}.
```

Recall that the notation `{ ... } + { ... }` is for a `Set`-version of the disjunction (Sect. 2.7).

We can turn a proposition in `Prop` into a boolean number:

```
Definition asbool (P : Prop) :=
  if pselect P then true else false.
```

`` `[< P >] `` is a notation for `asbool P`.

Contraposition lemmas that are true classically become provable, e.g.:

```
Lemma contra_notP (Q P : Prop) : (~ Q -> P) -> ~ P -> Q.
```

can be found in MathComp-Analysis `boolp` whereas

```
Lemma contraPnot (P Q : Prop) : (Q -> ~ P) -> (P -> ~ Q).
```

was already in Coq `ssrbool.v` (Sect. 3.4.1)

4.2 Naive Set Theory

We now define a theory of sets which are not necessarily finite. This comes as an addition to Sect. 3.4.8 and the naming actually overlaps.

4.2.1 Basic Set-theoretic Operations

In MathComp-Analysis, a set is formalized as a (characteristic) function from some type `T` to `Prop` (see Sect. 2.1.1):

```
Definition set T := T -> Prop.
```

`set0` is the empty set (the function that returns `False`) and `setT` is the full set (the function that returns `True`, notation `[set: T]` where `T` is the support type). Any `Prop`-valued function `P` gives rise to a set using the notation `[set x | P]`. Notation scope is `classical_set_scope`, delimiter `classic`.

Given an element `x : T`, we can write `x \in A` for a set `A`, this is a `bool` expression. Of course, this is equivalent to `A x`, which is in `Prop`. Rewriting with `inE` turns `x \in A` expression into a function application `A x`. Similarly, the lemma `mem_set` allows to move from `A x` to `x \in A` (and back with the lemma `set_mem`). One can use either `x \in A` or `A x` to state that an element belongs to a set.

Basic operations on sets can be formalized using basic logic operators (see Sect. 2.7). Intersection is essentially conjunction:

```
Definition setI A B := [set x | A x /\ B x].
```

One can use the notation ``A `&` B`` instead of `setI A B` in the scope `classical_set_scope`.

Union is essentially disjunction (notation: ``A `|` B``):

```
Definition setU A B := [set x | A x \/ B x].
```

Complement (notation: ``~` A``):

```
Definition setC A := [set a | ~ A a].
```

Subset relation (notation: ``A `<=` B``):

```
Definition subset A B := forall t, A t -> B t.
```

Exercise 4.2.1. State and prove De Morgan's laws.

Exercise 4.2.2. Find De Morgan's laws in `classical_sets.v` (MathComp-Analysis).

Exercise 4.2.3. Given a type `T`, show that `set T` with inclusion is a *poset* (reflexivity and transitivity). Show that `set T` with containment is a poset.

4.2.2 More Set-theoretic Constructs

The preimage of the set `A` by the function `f` is noted `f @^-1` A` (yes, you can get used to this notation). This is a notation for `[set t | A (f t)]`.

The iterated operations of core MathComp are finite (Sect. 3.4.7). With classical sets, we can deal with countable iterated unions:

```
Definition bigcup T I (P : set I) (F : I -> set T) :=
  [set a | exists2 i, P i & F i a].
```

Notations are similar to the ones for finite sets, that is: `\bigcup_(i in P) F`, `\bigcup_(i : T) F`, `\bigcup_(i < n) F`, etc. And similarly for countable iterated intersections `\bigcap_(i in P) F`, etc.

In measure theory in particular, there is a pervasive use of families of pairwise disjoint sets. `trivIset D F` is a predicate stating that the family of sets `F` indexed by `D` is pairwise disjoint:

```
Definition trivIset T I (D : set I) (F : I -> set T) :=
  forall i j : I, D i -> D j -> F i `&` F j !=set0 -> i = j.
```

There is an operation to pick a particular element from an arbitrary set defined by comprehension: `xget x0 P` returns an element of the set `P` or `x0` is `P` is empty.

There is also a predicate `finite_set` defined in the file MathComp-Analysis `cardinality.v` that discriminates `set`s that are actually finite. It uses a relation that defines the cardinality of a set.

4.3 Supremum and Infimum

Using classical sets (Sect. 4.2) and ordered types (Sect. 3.4.6), we can develop a theory for supremums and infimums.

The set of upper bounds of a set `A` is formed by the elements `y` such that $\forall x \in A,\ y \geq x$:

```
Definition ubound A : set T := [set y | forall x, A x -> (x <= y)%O].
```

Similarly for the set of lower bounds:

```
Definition lbound A : set T := [set y | forall x, A x -> (y <= x)%O].
```

A *supremum* is an upper bound that is less than or equal to any other upper bound:

```
Definition supremums A := ubound A `&` lbound (ubound A).
```

We call supremum an element of `supremums`:

```
Definition supremum x0 A :=
  if A == set0 then x0 else xget x0 (supremums A).
```

(`xget` is defined in Sect. 4.2)

4.4 Mathematical Structures in MathComp-Analysis

In Sect. 3.4, we saw that MathComp introduces a number of mathematical structures for algebra. MathComp-Analysis extends MathComp with the mathematical structures that appear in red in Fig. 4.1. We do not dwell upon their formalization now; we will look at an example in more details with better tools later when dealing with measure theory (Sect. 5.3).

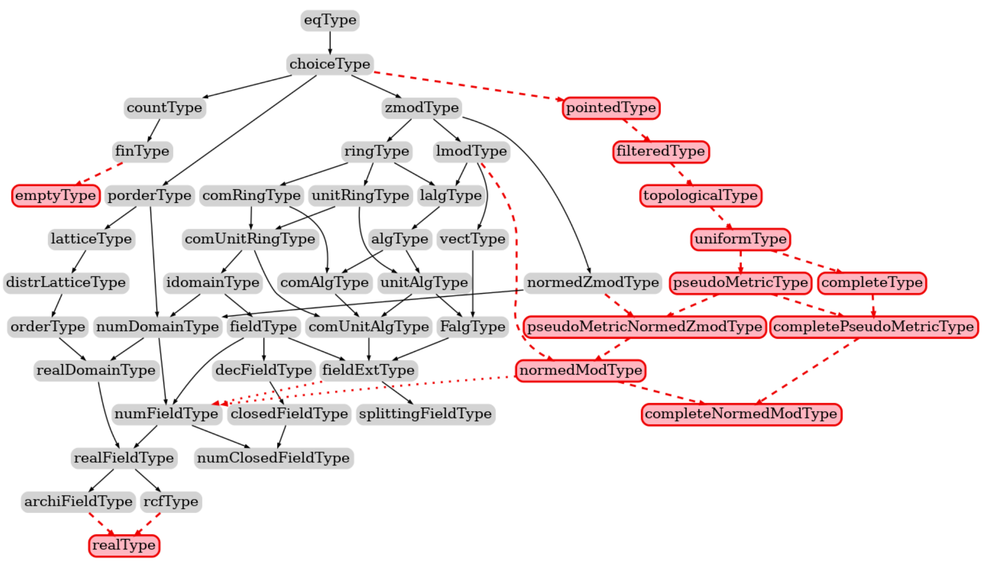


Figure 4.1: The hierarchy of mathematical structures of MathComp-Analysis as of 2022-11-16. The dotted lines do not mark inheritance but instances.

4.4.1 Pointed Types

Mathematical structures introduced by MathComp-Analysis are pointed. They necessarily have an object `point` that be used as a default value.

This is useful for example to define the restriction of a function to a set. `f \_ D` is the restriction of function `f` to `D`. Outside of `D`, it returns the `point` of the supporting `pointedType`.

`get` is a notation for `xget point` (see Sect. 4.2).

4.4.2 Real Numbers

Real numbers are defined in the file `reals.v`[MathComp-Analysis]. The type of real numbers extends a type `R : archiFieldType` (Sect. 3.5.2) with the following axioms:

1. for any non-empty set E with an upper bound, `supremum 0 E` is an upper bound (`supremum 0` is in fact noted `sup`)

2. for any non-empty set E with an upper bound, for any ε, there is an element $e \in E$ such that `sup E` $- \varepsilon < e$

Look for the mixin in MathComp-Analysis `reals.v`. See Sect. 4.6 for the other structures introduced by MathComp-Analysis.

To construct a real number from a natural number n: `n%:R`.

Exercise 4.4.1. Let d is a semimetric ($d(x,x) = 0$, $d(x,y) = d(y,x) \geq 0$, $d(x,z) \leq d(x,y) + d(y,z)$). Show that $\frac{d}{1+d}$ is a semimetric.

4.5 Convergence

4.5.1 Filters

Convergence in MathComp-Analysis is expressed using *filters*[1]. The notion of filter was introduced by Cartan:

> Ça n'avait pas de nom naturellement, cette notion que je venais de trouver, alors, pour se convaincre que ça marchait, on prenait des exemples, et puis au moment où l'instrument arrivait, on disait : "Boum !" Alors on a appelé ça les "boums" ! Évidemment ça ne pouvait pas rester longtemps les boums, et surtout s'il fallait publier le résultat. (Henri Cartan, [Broué, 2012])

The axioms of a filter F of type `set (set T)` are (see `Filter` in MathComp-Analysis `topology.v`):

1. The full set belongs to F (`filterT`).
2. F is closed by (binary) intersection (`filterI`).
3. F is closed by containment:

   ```
   filterS : forall P Q : set T, P `<=` Q -> F P -> F Q
   ```

The empty set can belong to a filter in MathComp-Analysis. This is the type of *proper filter* that excludes the empty set. See the `ProperFilter` in MathComp-Analysis `topology.v`. We thus recover the standard definition of filter [Wilansky, 2008, Sect. 3.2].

Given a family of sets `B : I -> set T` indexed by a set `D : set I`, the expression `filter_from D B` is the set of sets `P` that are supersets of `B i`'s:

```
Definition filter_from
    {I T : Type} (D : set I) (B : I -> set T) : set (set T) :=
  [set P | exists2 i, D i & B i `<=` P].
```

`filter_from setT B` forms a filter if `I` is not empty and if

```
forall i j, exists k, B k `<=` B i `&` B j
```

[1] Nothing to do with the `filter` function of MathComp `seq.v`.

This corresponds to the definition of a *filterbase* [Wilansky, 2008, Sect. 3.2] (see lemma `filter_fromT_filter`).

Example 4.5.1. The filter based on the sets $\{n \mid N \le n\}$ for all natural numbers N (i.e., the "$[N, \infty)$" for some N) is the "eventually filter" `\oo` to talk about the behavior of sequences when the index tends to infinity. It is defined as follows:

```
filter_from setT (fun N => [set n | (N <= n)%N])
```

See `eventually` and `eventually_filter` in MathComp-Analysis `topology.v`.

Example 4.5.2. The filter formed by the sets P such that there exists an M such that for all $M < x$, $x \in P$ is noted `+oo` (i.e., this filter contains all $]M; \infty)$ for some M). See `pinfty_nbhs`, declared to be a proper filter via `proper_pinfty_nbhs`, in MathComp-Analysis `normedtype.v`.

See MathComp-Analysis `normedtype.v` for the examples of filters corresponding to convergence to the right (`at_right`, notation `^'+`), to the left, etc.

4.5.2 Convergence using Filters

The notation for convergence in MathComp-Analysis is `F --> G` where `F` and `G` are filters. What is behind is just an inclusion, namely $G \subseteq F$.

We define the image of a filter F by a function f as the set of sets `P : set U` such that the preimage of `P` by `f` is in `F`:

```
Definition fmap {T U : Type} (f : T -> U) (F : set (set T)) :=
  [set P | F (f @^-1` P)].
```

The notation `E @[x --> F]` is the image of the filter `F` by the function `fun x => E`. `E x @[x --> F]` is the same as `E @ F`.

By combining the notation for convergence of filters and the notation for the image of a filter, we obtain a notation for convergence of functions (and of sequences): `f x @[x --> a] --> l` for

$$f(x) \underset{x \to a}{\to} l.$$

We can already define continuity: a function f is *continuous* iff $\forall a, f(x) \underset{x \to a}{\to} f(a)$. See the notation `continuous` in MathComp-Analysis `topology.v`.

4.5.3 Filtered Types

There is a notion of *filtered type* for types whose points can each be equipped with a filter. Given a point `x` in some filtered type, `nbhs x` is the set of sets associated with this point. See `Module Filtered` in MathComp-Analysis `topology.v`.

Limits

Given a filter `F`, `lim F` is defined as being a `l` such that `F --> l`. This definition uses the `get` operation seen in Sect. 4.2 (see `lim_in` in MathComp-Analysis `topology.v`). `lim F` is essentially a notation, so it should be `Search`ed as `(lim _)` rather than `lim`.

4.6 Other Structures in MathComp-Analysis

4.6.1 Topological Spaces

In MathComp-Analysis, the interface of a *topological space* is defined by the following mixin:

```
Record mixin_of (T : Type) (nbhs : T -> set (set T)) := Mixin {
  open : set (set T) ;
  ax1 : forall p : T, ProperFilter (nbhs p) ;
  ax2 : forall p : T, nbhs p =
    [set A : set T | exists B : set T, open B /\ B p /\ B `<=` A] ;
  ax3 : open = [set A : set T | A `<=` nbhs^~ A ]
}.
```

A topological space is therefore given by a set of sets that corresponds to the *open sets*. `nbhs` is meant to come from a filtered type (Sect. 4.5.3). Axiom `ax2` ensures that `nbhs` corresponds to the definition of *neighborhood*. Given a point `p`, a set in `nbhs p` is a set that contains an open that contains `p` (i.e., a neighborhood). Axiom `ax3` is an alternative characterization of open sets: an open set is a neighborhood of all of its points (`A p` implies `nbhs p A`). Axiom `ax1` adds that neighborhoods are proper filters. This gives rise to the type `topologicalType`.

The definition of topological space in MathComp-Analysis departs from the standard, textbook definition of topological space according to which a topological space is a set equipped with a set of subsets which is stable by union and by finite intersection, and that contains the empty set and the full set. The function `topologyOfOpenMixin` provides a way to construct the above mixin from the standard, textbook axioms.

Exercise 4.6.1. Find the lemmas in MathComp-Analysis `topology.v` corresponding to the standard, textbook definition of topological spaces.

It can be proved that if the support set is Hausdorff (i.e., $x \neq y$ can be separated by neighborhoods), see `hausdorff_space` in MathComp-Analysis `topology.v`), then the limit is unique, and we have:

```
Lemma cvg_lim {U : Type} {F} {FF : ProperFilter F} (f : U -> T) (l : T) :
  f @ F --> l -> lim (f @ F) = l.
```

4.6.2 Uniform Spaces

We mention the formalization of uniform space only for the sake of exhaustiveness because this is too technical.

Definition of neighborhoods using entourages:

```
Definition nbhs_ {T T'} (ent : set (set (T * T'))) (x : T) :=
  filter_from ent (fun A => to_set A x).
```

`to_set A x` is a notation for `[set y | A (x, y)]`.

Interface of *uniform spaces* (it extends topological spaces):

```
Record mixin_of (M : Type) (nbhs : M -> set (set M)) := Mixin {
  entourage : (M * M -> Prop) -> Prop ;
  ax1 : Filter entourage ;
  ax2 : forall A, entourage A -> [set xy | xy.1 = xy.2] `<=` A ;
  ax3 : forall A, entourage A -> entourage (A^-1)%classic ;
  ax4 : forall A, entourage A -> exists2 B, entourage B & B \; B `<=` A ;
  ax5 : nbhs = nbhs_ entourage
}.
```

Axiom `ax5` says that the notion of neighborhood using entourage is the same as the notion of neighborhoods using topological spaces.

4.6.3 Pseudometric Spaces

A *pseudometric space* extends a uniform space with a notion of ball (`ball x r` is a ball centered at `x` of radius `r`) and three intuitive axioms (reflexivity `ax1`, symmetry `ax2`, transitivity `ax3`):

```
Record mixin_of
    (R : numDomainType) (M : Type) (entourage : set (set (M * M))) :=
Mixin {
  ball : M -> R -> M -> Prop ;
  ax1 : forall x (e : R), 0 < e -> ball x e x ;
  ax2 : forall x y (e : R), ball x e y -> ball y e x ;
  ax3 : forall x y z e1 e2, ball x e1 y -> ball y e2 z ->
    ball x (e1 + e2) z;
  ax4 : entourage = entourage_ ball
}.
```

Axiom `ax4` states that the notion of entourage using balls is the same as the notion of entourage from uniform spaces:

```
Definition entourage_ {R : numDomainType} {T T'} (ball : T -> R -> set T') :=
  @filter_from R _ [set x | 0 < x] (fun e => [set xy | ball xy.1 e xy.2]).
```

In a pseudometric space, we can define the set of balls centered at `x` with a positive radius:

```
Definition nbhs_ball_ {R : numDomainType} {T T'} (ball : T -> R -> set T')
  (x : T) := @filter_from R _ [set e | e > 0] (ball x).
Definition nbhs_ball {R : numDomainType} {M : pseudoMetricType R} :=
 nbhs_ball_ (@ball R M).
```

`nbhs_ball` is provably equivalent to `nbhs`.

Given a norm, we can define balls the usual way:

```
Definition ball_
  (R : numDomainType) (V : zmodType) (norm : V -> R) (x : V) (e : R) :=
  [set y | norm (x - y) < e].
```

We use this definition of balls (`ball_`) with the definition of neighborhoods defined using balls (`nbhs_ball_`) to define a filtered type for `normedZmodType`'s (see Sect. 3.5.2) and by extension for `numDomainType`, `realDomainType`, `numFieldType`, `realFieldType`, `realType`, etc. so that for the latter mathematical structures, the notion of ball coincides the notion of ball defined using the norm.

4.6.4 Complete Spaces

We only mention for the sake of exhaustiveness that pseudometric spaces can furthermore be extended to complete spaces but we do not use them in this document. See $\overset{\text{MathComp-Analysis}}{\texttt{topology.v}}$.

4.6.5 Normed Modules

Normed modules extend pseudometric spaces with a scaling operation.

We first introduce an intermediate (a bit artificial) mathematical structure of `pseudoMetric_normedZmodType`:

```
Record mixin_of (T : normedZmodType R) (ent : set (set (T * T)))
    (m : PseudoMetric.mixin_of R ent) := Mixin {
  _ : PseudoMetric.ball m = ball_ (fun x => `| x |) }.
```

It combines a `normedZmodType` (Sect. 3.5.2) with a pseudometric space (Sect. 4.6.3).

Then we combine a `pseudoMetric_normedZmodType` with a left module (see Sect. 3.5.1) and the following axiom:

```
Record mixin_of (K : numDomainType)
  (V : pseudoMetricNormedZmodType K) (scale : K -> V -> V) := Mixin {
  _ : forall (l : K) (x : V), `| scale l x | = `| l | * `| x |;
}.
```

For example, the type of real numbers (Sect. 4.4.2) can be equipped with the structure of normed module.

4.7 near Notations and Tactics

As we saw in Sect. 4.5, MathComp-Analysis is using filters for convergence. The use of filters calls for the introduction of dedicated notations and tactics. In general one does not use filters directly to prove a statement of the form `f @ F --> y` but rather a combination of ε reasoning and filter reasoning through the `near` notations and tactics [Affeldt et al., 2018].

The notation `\forall t \near F, P` is a notation for a proposition `P` that holds when `t` "is near" `F`. For example, when `F` is the `\oo` or the `+oo` filter, this intuitively means that `t` tends towards ∞. Of course, if one can prove `forall x, P x`, one can also prove `\forall x \near F, P x` for any filter `F`:

```
Lemma nearW {T : Type} {F : set (set T)} (P : T -> Prop) :
  Filter F -> (forall x, P x) -> (\forall x \near F, P x).
```

Switching from a convergence statement of the form `f x @[x --> F] --> y` to a combination of ε and `near` notations is the matter of a family of lemmas such as:

```
Lemma cvgrPdist_lt {T} {F : set (set T)} {FF : Filter F} (f : T -> V) (y : V) :
  f @ F --> y <-> forall eps, 0 < eps -> \forall t \near F, `|y - f t| < eps.

Lemma cvgr_dist_lt {T} {F : set (set T)} {FF : Filter F} (f : T -> V) (y : V) :
  f @ F --> y -> forall eps, 0 < eps -> \forall t \near F, `|y - f t| < eps.

Lemma cvgrPdist_le {T} {F : set (set T)} {FF : Filter F} (f : T -> V) (y : V) :
  f @ F --> y <-> forall eps, 0 < eps -> \forall t \near F, `|y - f t| <= eps.
...
```

See $\overset{\text{MathComp-Analysis}}{\texttt{normedtype.v}}$.

Introducing a near variable is performed by the tactic `near=>`. Discharging a near variable is performed by the tactic `near:`. `near do rewrite ...` is a short cut for `near=> x; rewrite ...; near: x`. At the time of this writing, these are *not* a combination with the `=>` and `:` tacticals of Sect. 2.2.

Scripts using the `near` tactic need to be concluded with

```
Unshelve. all: end_near. Qed.
```

instead of `Qed`.

Example: .../...

```
Lemma opp_continuous : continuous (@GRing.opp V).
Proof.
move=> y.
```

```
y : V
----------------------------------------
- x @[x --> y] --> - y
```

```
apply/cvgrPdist_lt => e e0.
```

```
y : V
e : K
e0 : 0 < e
----------------------------------------
\forall t \near y, `|- y - - t| < e
```

This is a notation to say that the set `[set t | `|- y - - t| < e]` belongs to `nbhs`, the neighboring filter of `y`. The neighboring filter of `y` is indeed defined using the balls that are centered at `x`: `[set z | norm (y - z) < e]` for all `e` (Sect. 4.6.3).

```
near=> t.
```

```
y : V, e : K, e0 : 0 < e
t : V
_Hyp_ : t \is_near (nbhs y)
----------------------------------------
`|- y - - t| < e
```

```
rewrite -opprD normrN.
```

```
y : V, e : K, e0 : 0 < e
t : V
_Hyp_ : t \is_near (nbhs y)
----------------------------------------
`|y - t| < e
```

```
near: t.
```

```
y : V, e : K, e0 : 0 < e
----------------------------------------
\forall t \near nbhs y, `|y - t| < e
```

```
exact: cvgr_dist_lt.
Unshelve. all: by end_near. Qed.
```

Another example: .../...

```
Lemma cvgD f g a b : f @ F --> a -> g @ F --> b -> (f + g) @ F --> a + b.
Proof.
move=> fFa gFb; apply/cvgrPdist_lt => e e0.
near=> t.
```

```
fFa : f x @[x --> F] --> a
gFb : g x @[x --> F] --> b
e : K
e0 : 0 < e
t : T
_Hyp_ : t \is_near (nbhs F)
============================
`|a + b - (f + g) t| < e
```

```
rewrite opprD addrAC addrA -(addrA (a - _)) -(addrC b) (splitr e).
```

```
============================
`|a - f t + (b - g t)| < e / 2 + e / 2
```

```
rewrite (le_lt_trans (ler_norm_add _ _))// ltr_add//.
```

```
t : T
_Hyp_ : t \is_near (nbhs F)
============================
`|a - f t| < e / 2

goal 2 is:
`|b - g t| < e / 2
```

```
near: t.
```

```
e : K
e0 : 0 < e
============================
\forall x \near nbhs F, `|a - f x| < e / 2
```

```
apply: cvgr_dist_lt => //.
```

```
e : K
e0 : 0 < e
============================
0 < e / 2
```

```
by rewrite divr_gt0.
```

The proof is similar for the other goal.

.../...

Other lemmas to look at if time permits:

- In MathComp-Analysis `topology.v` : `closed_cvg`, etc.
- In MathComp-Analysis `normedtype.v` : `lime_le`, `cvgr_lt`, etc.

4.8 Sequences

Sequences are defined in the eponymous file MathComp-Analysis `sequences.v` . They are just functions with domain `nat`:

```
Definition sequence R := nat -> R.
```

where `R` is expected to be a numeric type. `R ^nat` is a notation for `sequence R`.

Because sequences are a special case of functions, the lemmas from MathComp-Analysis `topology.v` and MathComp-Analysis `normedtype.v` are readily available to deal with sequences.

Example: the squeeze Lemma .../...

```
Context {T : Type} {a : set (set T)} {Fa : Filter a} {R : realFieldType}.
Lemma squeeze_cvgr f g h : (\near a, f a <= g a <= h a) ->
  forall (l : R), f @ a --> l -> h @ a --> l -> g @ a --> l.
Proof.
move=> fgh l lfa lga.
```

```
fgh : \near a, f a <= g a <= h a
l : R
lfa : f x @[x --> a] --> l
lga : h x @[x --> a] --> l
============================
g x @[x --> a] --> l
```

```
apply/cvgrPdist_lt => e e_gt0; near=> x.
```

```
e : R
e_gt0 : 0 < e
x : T
_Hyp_ : x \is_near (nbhs a)
============================
`|l - g x| < e
```

```
have := near fgh x. (* near lemma here *)
```

```
x : T
_Hyp_ : x \is_near (nbhs a)
============================
(x \is_near (nbhs a) -> f x <= g x <= h x) -> `|l - g x| < e
```

```
move=> /(_ _)/andP[//|fg gh].
```

```
fg : f x <= g x
gh : g x <= h x
============================
`|l - g x| < e
```

```
rewrite distrC ltr_distl (lt_le_trans _ fg) ?(le_lt_trans gh)//=.
```

```
fg : f x <= g x
gh : g x <= h x
============================
h x < l + e

goal 2 is:
l - e < f x
```

```
by near: x; apply: (cvgr_lt l); rewrite // ltr_addl.
```

And the other goal is similar. .../...

Countable sums are defined in MathComp-Analysis `sequences.v`. This is just a combination of the iterated operations of MathComp (Sect. 3.4.7) and of the notion of limit from MathComp-Analysis `topology.v` (Sect. 4.5.3):

```
Notation "\big [ op / idx ]_ ( i <oo | P ) F" :=
  (lim (fun n => (\big[ op / idx ]_(i < n | P) F))) : big_scope.
```

In the development of measure theory (Chapter 5), we are going to use in particular countable sums of extended real numbers.

Chapter 5

Measure Theory with MathComp-Analysis

Goal of this chapter: We introduce the basics of measure theory with MathComp-Analysis *and illustrate the use of* Hierarchy-Builder *to build a hierarchy of mathematical structures for measure theory.*

5.1 Extended Real Numbers

At the time of this writing, the theory of extended real numbers spans two files: `constructive_ereal.v` (MathComp-Analysis) and `ereal.v` (MathComp-Analysis). The definition in itself is in `constructive_ereal.v` (MathComp-Analysis):

```
Variant extended (R : Type) := EFin of R | EPInf | ENInf.
```

The notation scope is `ereal_scope`, with delimiter `E`. The notation `r%:E` is for `EFin r`; it is to inject a real number into the extended real numbers. `+oo` is for `EPInf` and `-oo` is for `ENInf`. Regarding naming conventions, `y` means ∞, `Ny` means $-\infty$ (remember Table 3.4).

The extended real numbers do not have the best structure. They do not form a group because $\infty - \infty$ is undefined. How to deal with such exceptional cases is always a delicate matter. The choice of MathComp-Analysis is to define $\infty - \infty$ to be $-\infty$ so that the addition (`+%E`) is associative and that the set of extended real numbers forms a monoid (see Sect. 3.4.7):

```
Definition adde_subdef x y :=
  match x, y with
  | x%:E , y%:E  => (x + y)%:E
  | -oo, _       => -oo
  | _    , -oo => -oo
  | +oo, _       => +oo
```

```
  | _      , +oo => +oo
  end.
```

Exercise 5.1.1. Define the addition so that $\infty - \infty = 0$ and show that the addition is not associative.

We define the supremum of a set extended real numbers using `supremum`, like we did for real numbers in Sect. 4.4.2, except that we can now take the default value to be $-\infty$:

```
ereal_sup S := supremum -oo S
```

Sequences of Extended Real Numbers Sequences of extended real numbers are heavily used in measure theory.

The lemmas about sequences of extended real numbers should be reminiscent of the their real numbers counterparts. For example, compare this lemma for extended real numbers (where `+oo` refers to `+oo%E`)

```
Lemma cvgeyPge : f @ F --> +oo <-> forall A, \forall x \near F, A%:E <= f x.
```

with its counterpart for real numbers (where `+oo` refers to the filter seen in Sect. 4.5.1):

```
Lemma cvgryPge : f @ F --> +oo <-> forall A, \forall x \near F, A <= f x.
```

In Sect. 4.8, we explained that countable (generic) sums are formally defined as a combination of (finite) iterated operations and limits. We instantiate this definition with the addition of extended real numbers:

```
Notation "\sum_ ( m <= i <oo | P ) F" :=
  (\big[+%E/0%E]_(m <= i <oo | P%B) F%E) : ereal_scope.
```

and then go one developing the theory of series of extended real numbers with lemmas reminiscent of iterated operations such as:

```
Lemma eq_eseries (R : realFieldType) (f g : (\bar R)^nat) (P : pred nat) :
  (forall i, P i -> f i = g i) ->
  \sum_(i <oo | P i) f i = \sum_(i <oo | P i) g i.
```

You can `Search` for the `eseries` substring in MATHCOMP-ANALYSIS `sequences.v` to find out about generic lemmas about countable sums and for the `nneseries` substring for lemmas about non-negative terms.

The notations for countable sums that we introduced so far were defined as a combination of MATHCOMP (finite) iterated operations and limit. We introduce another, compatible definition expressed as the combination of finitely-supported sums (Sect. 3.4.7) and supremum: *sums over general sets*.

$$\sum_{i \in S} a_i \stackrel{\text{def}}{=} \sup \left\{ \sum_{i \in A} a_i \mid A \text{ finite subset of } S \right\}.$$

In COQ:

```
Definition esum S a := ereal_sup [set \sum_(x \in A) a x | A in fsets S].
Notation "\esum_ ( i 'in' P ) A" := (esum P (fun i => A)).
```

where `fsets S` is the set of finite sets (defined using classical sets—Sect. 4.2) included in `S`.

5.2 Building Hierarchies with Hierarchy-Builder

Hierarchy-Builder is a tool introduced in [Cohen et al., 2020] to facilitate the construction of hierarchies of mathematical structures. It provides new commands, the main ones being `HB.mixin`, `HB.structure`, `HB.factory`, and `HB.instance`. The next pages will provide examples of their usage.

Let us just explain in a generic way the most basic scenario. Here is the pattern to declare a structure `Struct` intended to sit at the bottom of a hierarchy. The interface of the structure goes into a mixin:

```
HB.mixin Record isStruct params carrier := {
  ... properties about the carrier ...
}
```

The structure per se (and its type) is declared like a sigma-type:

```
#[short(type=structType)]
HB.structure Definition Struct := {carrier of isStruct carrier}
```

Hierarchy-Builderis using Coq *attributes* (`#[...]`) to declare the type corresponding to a structure. See the Coq reference manual for more information about Coq attributes, although there are not much to know about them for our purpose.

Here is the pattern to declare a new structure `NewStruct` that extends an existing structure `Struct`; note the `of` syntax.

```
HB.mixin Record NewStruct_from_Struct params carrier
    of Struct params carrier := {
  ... more properties about the carrier ...
}
```

In the case of the extended structure, the sigma-type makes appear the dependency to the parent structure.

```
#[short(type=newStructType)]
HB.structure NewStruct params :=
  {carrier of NewStruct_from_Struct parames carrier
            & Struct params carrier}.
```

This process results in the creation of the types `structType` and `newStructType` such that elements of the latter are also understood to be elements of the former.

5.3 Formalization of σ-algebras

The type of a σ-algebra can be defined as the result of a hierarchy of mathematical structures comprising semiring of sets, ring of sets, and algebra of sets (Fig. 5.1). It is defined in this way in particular when we build measures by extension using Carathéodory's theorem.

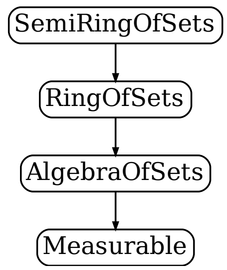


Figure 5.1: Hierarchy of measure theory structures

A *semiring of sets* is a set of sets called `measurable` that contains the empty set, that is closed under intersection (`setI_closed`) and that is "closed by semi difference" (`semi_setD_closed`):

```
HB.mixin Record isSemiRingOfSets (d : measure_display) T := {
  ptclass : Pointed.class_of T;
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableI : setI_closed measurable;
  semi_measurableD : semi_setD_closed measurable;
}.
```

Here the carrier is `T : Type`. Do not care too much about the `measure_display` parameter, this is a trick to get nice notations (check [Affeldt and Cohen, 2022, Sect. 3.4]). Do not care too much also about the field `ptclass`, this is to make inference work correctly with the current version of MathComp (this field should become useless with MathComp 2.0).

The definition of `setI_closed` is obvious:

```
Definition setI_closed := forall A B, G A -> G B -> G (A `&` B).
```

The definition of `semi_setD_closed` is more contrived:

```
Definition semi_setD_closed := forall A B, G A -> G B -> exists D,
  [/\ finite_set D,
      D `<=` G,
      A `\` B = \bigcup_(X in D) X &
      trivIset D id].
```

To paraphrase, it means that given two sets A and B belonging to G, there exists a set of sets D such that: (1) D is finite, (2) $D \subseteq G$, (3) $A \setminus B = \bigcup_{X \in D} X$, and (4) sets in D are pairwise disjoint.

The type of semirings of sets is `semiRingOfSetsType`, defined as a sigma-type (Sect. 2.8) with a carrier that satisfies the mixin `isSemiRingOfSets`:

```
#[short(type=semiRingOfSetsType)]
HB.structure Definition SemiRingOfSets d := {T of isSemiRingOfSets d T}.
```

A *ring of sets* is a semiring of sets that is closed by finite union (predicate `setU_closed`):

```
HB.mixin Record RingOfSets_from_semiRingOfSets d T
    of isSemiRingOfSets d T := {
  measurableU :
    setU_closed (@measurable d [the semiRingOfSetsType d of T]) }.
```

Observe that the mixin of ring of sets uses the `measurable` field of the mixin of semiring of sets. The notation `[the semiRingOfSetsType d of T]` is for forcing `T` to be recognized as a semiring of sets instead of a mere `Type`. From Coq version 8.16, we actually do not need to use this complicated syntax anymore but we are trying to keep the code compatible with earlier versions of Coq for a while.

```
#[short(type=ringOfSetsType)]
HB.structure Definition RingOfSets d :=
  {T of RingOfSets_from_semiRingOfSets d T & SemiRingOfSets d T}.
```

An *algebra of sets* is a ring of sets that contains the full set:

```
HB.mixin Record AlgebraOfSets_from_RingOfSets d T of RingOfSets d T := {
  measurableT : measurable [set: T]
}.
HB.structure Definition AlgebraOfSets d :=
  {T of AlgebraOfSets_from_RingOfSets d T & RingOfSets d T}.
```

A *σ-algebra* is an algebra of sets that is closed by countable union:

```
HB.mixin Record Measurable_from_algebraOfSets d T
    of AlgebraOfSets d T := {
  bigcupT_measurable : forall F : (set T)^nat,
    (forall i, measurable (F i)) -> measurable (\bigcup_i (F i))
}.
#[short(type=measurableType)]
HB.structure Definition Measurable d :=
  {T of Measurable_from_algebraOfSets d T & AlgebraOfSets d T}.
```

In the end, we thus arrive at a type `measurableType` that is now available to declare σ-algebras and develop their theory.

Example of a derived property: .../...

```
Lemma bigcap_measurable F P :
  (forall k, P k -> measurable (F k)) -> measurable (\bigcap_(i in P) F i).
Proof.
move=> PF; rewrite -[X in measurable X]setCK.
```

```
d : measure_display
T : measurableType d
F : (set T) ^nat
P : set nat
PF : forall k : nat, P k -> d.-measurable (F k)
============================
d.-measurable (~` ~` (\bigcap_(i in P) F i))
```

```
rewrite setC_bigcap.
```

```
============================
d.-measurable (~` (\bigcup_(i in P) ~` F i))
```

```
apply: measurableC.
```

```
============================
d.-measurable (\bigcup_(i in P) ~` F i)
```

```
apply: bigcup_measurable => k Pk.
```

```
PF : forall k : nat, P k -> d.-measurable (F k)
k : nat
Pk : P k
============================
d.-measurable (~` F k)
```

```
apply: measurableC.
```

```
PF : forall k : nat, P k -> d.-measurable (F k)
k : nat
Pk : P k
============================
d.-measurable (F k)
```

```
exact/PF.
Qed.
```

.../...

We have successful defined σ-algebras but we are in a situation similar to topological spaces in Sect. 4.6.1: the mixin of σ-algebra does not correspond to the standard, stand-alone definition of a σ-algebra as set of subsets that contains the full set, and that is closed under complementation and countable unions. For topological spaces, we saw that MathComp-Analysis `topology.v` provides a constructor `topologyOfOpenMixin` whose signature corresponds to the standard, stand-alone definition. The `HB.factory` command of Hierarchy-Builder is for dealing with such situations. A *factory* is very much like a mixin in the sense that it is an interface:

```
HB.factory Record isMeasurable (d : measure_display) T := {
  ptclass : Pointed.class_of T;
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableC : forall A, measurable A -> measurable (~` A) ;
  measurable_bigcup : forall F : (set T)^nat,
    (forall i, measurable (F i)) -> measurable (\bigcup_i (F i))
}.
```

The difference with a mixin is that the developer has to provide a proof that from the factory one can build the original mixin that defined the σ-algebra in the first place. This is performed by the `HB.builders` command of Hierarchy-Builder [Cohen et al., 2020]. From the user perspective, this is a simplification: the user can use either interface to create a σ-algebra.

5.4 Generated σ-algebra

The goal of this section is to show that we can define a concrete example of σ-algebra that inhabits the type `measurableType`.

A *generated σ-algebra* `<< G >>` is the smallest σ-algebra that contains some set of sets `G`.

We start by defining the notion of "smallest". We want that `smallest p G` defines the smallest set `M` such that ``G `<=` M`` and such that `M` verifies the property `p`. We can take the intersection of all such `M`'s:

```
Context {T} (C : set T -> Prop) (G : set T).
Definition smallest := \bigcap_(A in [set M | C M /\ G `<=` M]) A.
```

This is actually already defined in MathComp-Analysis `classical_sets.v`.

We now define a predicate to qualify σ-algebras:

```
Definition salgebra T (G : set (set T)) :=
  [/\ G set0, (forall A, G A -> G (~` A)) &
    (forall A : (set T)^nat, (forall n, G (A n)) -> G (\bigcup_k A k))].
```

This is not a type like `measurableType`, even though its contents are essentially the axioms of a σ-algebra.

Therefore, the smallest σ-algebra that contains a set of sets `G` is:

```
Notation "<< G >>" := (smallest (@salgebra _) G).
```

Can we, for any `G`, use `<< G >>` to instantiate the interface of σ-algebra from Sect. 5.3?

We start by proving that `<< G >>` is a σ-algebra in the sense of the predicate `salgebra`:

```
Variables (T : Type) (G : set (set T)).

Lemma salgebra0 : << G >> set0.
Proof. ... Qed.

Lemma salgebraC : forall A, << G >> A -> << G >> (~` A).
Proof. ... Qed.

Lemma salgebraU : forall A : (set T)^nat,
  (forall n, << G >> (A n)) -> << G >> (\bigcup_k A k).
Proof. ... Qed.
```

Exercise 5.4.1. Prove `salgebra0`, `salgebraC`, `salgebraU`. Do that in a new file that starts by loading the header of $\overset{\text{MathComp-Analysis}}{\texttt{measure.v}}$ and add

```
From mathcomp Require Import measure.
```

Since we have verified all the properties of a σ-algebra, we can instantiate the factory of Sect. 5.3. This factory has one parameter (a display), a carrier (a `Type`), (a technical thing about pointed types, do not mind, as explained earlier this is scheduled for disappearance soon,) a set of sets (the measurable sets), and three properties. Let us prepare one identifier `sdisplay` for the display, one identifier `salgType` to extract the carrier from a set of sets, and take `<< G >>` to be the set of sets. We can instantiate using the `HB.instance` command:

```
Variables (T : pointedType) (G : set (set T)).

(* don't ask *)
Canonical salgType_eqType := EqType (salgType G) (Equality.class T).
Canonical salgType_choiceType := ChoiceType (salgType G) (Choice.class T).
Canonical salgType_ptType := PointedType (salgType G) (Pointed.class T).
(* /dont' ask *)

HB.instance Definition _ := @isMeasurable.Build
  (sdisplay G)
  (salgType G)
  (Pointed.class T) (* don't ask *)
  << G >>
  (@salgebra0 _ G) (@salgebraC _ G) (@salgebraU _ G).
```

`isMeasurable.Build` is a constructor function that has been produced by Hierarchy-Builder upon definition of the `isMeasurable` interface, be it a mixin or a factory.

As a consequence of the above instantiation, we are now given for any set of sets `G`, the type `salgType G` of a generated σ-algebra:

```
Variable T : pointedType.
Variable G : set (set T).
Check << G >> : set (set T).
Check salgType G : measurableType _.
```

5.5 Formalization of Measures

In the same way that a σ-algebra is also an algebra of sets, and a ring of sets, and a semiring of sets, etc. a measure is also an additive measure. HIERARCHY-BUILDER can also deal with hierarchy of morphisms, where the carrier is not a `Type` but a function.

A function of type `set T -> \bar R` is *semiadditive* when for any sequence of measurable, pairwise-disjoint sets F we have

$$\forall n, \text{measurable}\left(\bigcup_{k<n} F_k\right) \to \mu\left(\bigcup_{k<n} F_k\right) = \sum_{k<n} \mu(F_k)$$

Formal paraphrase in MATHCOMP-ANALYSIS:

```
Definition semi_additive := forall F n,
 (forall k : nat, measurable (F k)) -> trivIset setT F ->
  measurable (\big[setU/set0]_(k < n) F k) -> (* *** *)
  mu (\big[setU/set0]_(i < n) F i) = \sum_(i < n) mu (F i).
```

Of course, the condition `(* *** *)` is trivially satisfied when `T` is a `measurabelType`.

A function of type `set T -> \bar R` is *semi-σ-additive* when, for any sequence of measurable, pairwise-disjoint sets F we have

$$\text{measurable}\left(\bigcup_{k} F_k\right) \to \sum_{k<n} \mu(F_k) \underset{n\to\infty}{\to} \mu\left(\bigcup_{k} F_k\right)$$

Formal paraphrase in MATHCOMP-ANALYSIS:

```
Definition semi_sigma_additive :=
  forall F, (forall i : nat, measurable (F i)) -> trivIset setT F ->
  measurable (\bigcup_n F n) ->
  (fun n => \sum_(0 <= i < n) mu (F i)) --> mu (\bigcup_n F n).
```

Hierarchy of Measures At the bottom of the hierarchy of measures, we put *additive measures*. They are non-negative functions that are *semiadditive*:

```
HB.mixin Record isAdditiveMeasure d
    (R : numFieldType) (T : semiRingOfSetsType d)
    (mu : set T -> \bar R) := {
  measure_ge0 : forall x, 0 <= mu x ;
  measure_semi_additive : semi_additive mu }.

HB.structure Definition AdditiveMeasure d
    (R : numFieldType) (T : semiRingOfSetsType d) := {
  mu & isAdditiveMeasure d R T mu }.
```

We do not restrict ourselves to the type of real numbers `realType` but instead use the more general `numFieldType`. Similarly, we take the domain of the measure to be over a semiring of sets. The fact that the measure of the empty set is 0 is a consequence of semiadditivity.

Measures extend semiadditive measures but by adding semi-σ-additivity:

```
HB.mixin Record isMeasure0 d
    (R : numFieldType) (T : semiRingOfSetsType d)
    mu of isAdditiveMeasure d R T mu := {
  measure_semi_sigma_additive : semi_sigma_additive mu }.

#[short(type=measure)]
HB.structure Definition Measure d
    (R : numFieldType) (T : semiRingOfSetsType d) :=
  {mu of isMeasure0 d R T mu & AdditiveMeasure d mu}.
```

The result is the hierarchy of Fig. 5.2. Of course, it is not necessary to go through the trouble of defining an intermediate semiadditive measure to define a measure, the user can instantiate directly a measure by using the `isMeasure` factory, see the code [Affeldt et al., 2017, file `measure.v`].

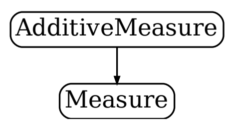


Figure 5.2: Hierarchy of measure structures

5.5.1 Example: the Dirac Measure

In the same way we instantiated the interface of σ-algebra with generated σ-algebras, we can populate the interface of measures with concrete measure. The *Dirac measure* δ_a is the measure that is 1 for sets A such that $a \in A$ and 0 otherwise:

```
Context d (T : measurableType d) (a : T) (R : realFieldType).
Definition dirac (A : set T) : \bar R := (\1_A a)%:E.
```

`\1_A a` is a notation for the *indicator function* defined in MathComp-Analysis `numfun.v`. It is really just a wrapper for the boolean test `a \in A`.

To declare it as a measure we need three proofs

```
Let dirac0 : dirac set0 = 0. Proof. by rewrite /dirac indic0. Qed.
Let dirac_ge0 B : 0 <= dirac B. Proof. by rewrite /dirac indicE. Qed.
Let dirac_sigma_additive : semi_sigma_additive dirac.
Proof. (* see page 80 *) Qed.
```

and an invocation of `HB.instance`:

```
HB.instance Definition _ := isMeasure.Build _ _ _
  dirac dirac0 dirac_ge0 dirac_sigma_additive.
```

.../...

```
Let dirac_sigma_additive : semi_sigma_additive dirac.
Proof.
move=> F mF tF mUF; rewrite /dirac indicE; have [|aFn] /= := boolP (a \in _).
```

```
F : nat -> set T
mF : forall i : nat, d.-measurable (F i)
tF : trivIset [set: nat] F
mUF : d.-measurable (\bigcup_n F n)
============================
a \in \bigcup_n F n ->
  (fun n : nat => \sum_(0 <= i < n) (\1_(F i) a)%:E) --> 1
```

```
rewrite inE => -[n _ Fna].
```

```
n : nat
Fna : F n a
============================
(fun n0 : nat => \sum_(0 <= i < n0) (\1_(F i) a)%:E) --> 1
```

```
apply/cvg_ballP => e e_gt0; near=> m.
```

```
e : R
e_gt0 : (0 < e)%R
m : nat
_Hyp_ : m \is_near (nbhs \oo)
============================
ball 1 e (\sum_(0 <= i < m) (\1_(F i) a)%:E)
```

```
have mn : (n < m)%N by near: m; exists n.+1.
```

```
mn : (n < m)%N
============================
ball 1 e (\sum_(0 <= i < m) (\1_(F i) a)%:E)
```

```
rewrite [X in ball _ _ X](_ : _ = 1)//; first exact: ballxx.
```

```
============================
\sum_(0 <= i < m) (\1_(F i) a)%:E = 1
```

This last goal is easy to prove since there is only one $n < m$ such that $a \in F\,n$. The second goal (that was generated by the case analysis at the first line of the script) is:

```
aFn : a \notin \bigcup_n F n
============================
(fun n : nat => \sum_(0 <= i < n) (\1_(F i) a)%:E) --> 0
```

This holds because the left-hand side is the constant function 0. .../...

5.5.2 Other Measures

MathComp-Analysis already provides many measures. In [Affeldt et al., 2017, file `measure.v`], besides the Dirac measure, the pushforward measure, the null measure, the sum of measures (be it finite or countable), the scaled (be a non-negative number) measure, the restriction of a measure, the counting measure, the product measure, and other kind of a bit more abstract measures that are involved in the construction of the *Lebesgue measure* [Affeldt et al., 2017, file `lebesgue_measure.v`] [Affeldt and Cohen, 2022] or the Lebesgue-Stieltjes measure.

5.6 Measurable Functions

A function with domain D is *measurable* when the preimage of any measurable set is measurable:

```
Definition measurable_fun d d' (T : measurableType d) (U : measurableType d')
    (D : set T) (f : T -> U) :=
  measurable D -> forall Y, measurable Y -> measurable (D `&` f @^-1` Y).
```

Note that this definition does not rely on the definition of measures, only on the definition of σ-algebra. Measurable functions are precisely the functions that we will integrate in the next chapter.

There is fairly large theory of measurable functions developed in MathComp-Analysis `measure.v` and MathComp-Analysis `lebesgue_measure.v`. You may want to know, e.g., whether the set of measurable functions is stable by composition (lemma `measurable_fun_comp`), or whether the set of real number-valued measurable functions is stable by pointwise addition:

```
Context d (T : measurableType d) (R : realType).
Lemma measurable_funD D (f g : T -> R) :
  measurable_fun D f -> measurable_fun D g -> measurable_fun D (f \+ g).
```

Exercise 5.6.1. Show that $\lambda x.x^2 + x^3$ is a measurable function. You need to use `lebesgue_measure.v`.

Chapter 6

Integration Theory with MathComp-Analysis

Goal of this chapter: We explain how we use the measure theory of MathComp-Analysis *to develop integration theory.*

6.1 Simple Functions

On paper, a *simple function* f is typically defined by a sequence of pairwise-disjoint and measurable sets $A_0, \dots A_{n-1}$ and a sequence of elements $a_0, \dots, a_{n-1}$ such that $f(x) = \sum_{k=0}^{n-1} a_k \mathbf{1}_{A_k}(x)$. One can choose to formalize this definition directly, for example by representing the a_k's by a list without duplicate, and use this representation to develop the necessary theory to formalize integration.

MathComp-Analysis is taking a bit more abstract and compositional approach by formalizing the hierarchy of Fig. 6.1 where simple functions are measurable functions with a finite image.

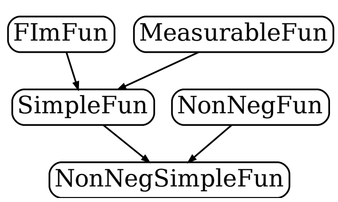


Figure 6.1: Hierarchy for non-negative simple functions

At the bottom of Fig. 6.1, we find functions with a finite image. Functions with a finite image are defined in MathComp-Analysis `cardinality.v` by:

```
HB.mixin Record FiniteImage aT rT (f : aT -> rT) := {
  fimfunP : finite_set (range f)
```

```
}.
HB.structure Definition FImFun aT rT := {f of @FiniteImage aT rT f}.
```

`range f` is a notation for `[set f x | x in setT]`. Let `{fimfun aT >-> rT}` be a notation for functions from `T` to `R` with a finite image. Given a function with a finite image, we can prove that it decomposes into a sum of indicator functions, like a simple function should, except that we do not need any measurability hypotheses:

```
Lemma fimfunE T (R : ringType) (f : {fimfun T >-> R}) x :
  f x = \sum_(y \in range f) (y * \1_(f @^-1` [set y]) x).
```

See MathComp-Analysis `numfun.v` for this lemma.

At the time of this writing, the interface for measurable functions is defined in MathComp-Analysis `lebesgue_integral.v` by:

```
HB.mixin Record IsMeasurableFun d (aT : measurableType d) (rT : realType)
    (f : aT -> rT) := {
  measurable_funP : measurable_fun setT f
}.
HB.structure Definition MeasurableFun d aT rT :=
  {f of @IsMeasurableFun d aT rT f}.
```

The interface uses the `measurable_fun` predicate from Sect. 5.6. This interface is restricted to real-valued functions. Let `{mfun aT >-> R}` be the notation for Hierarchy-Builder-defined measurable functions.

The structure of *simple functions* is obtained by combining the interfaces of functions with a finite image and of measurable functions. Notation for simple functions: `{sfun aT >-> R}`

Similarly, we can define the interface of non-negative functions with a notation `{nnfun T >-> R}` and combine with simple functions to get *non-negative simple functions* with notation `{nnsfun T >-> R}`.

6.1.1 Approximation Theorem

Before defining integration, we prove a theorem to approximate measurable functions using simple functions. This is an important theorem, used pervasively, in particular to establish the monotone convergence theorem (Sect. 6.3.3). The idea to build the approximation function is explained in Fig. 6.2.

Definition 6.1.1 (Dyadic Interval). Given n and k, we call *dyadic interval* the interval $I_{n,k} \stackrel{\text{def}}{=} [\frac{k}{2^n}; \frac{k+1}{2^n}[$.

Let f be an extended real number-valued measurable function with domain D. Given an n, we define B_n to be the set $D \cap \{x \mid n \le fx\}$:

```
Let B n := D `&` [set x | n%:R%:E <= f x]%E.
```

Given n and k, we define $A_{n,k}$ to be the set $D \cap \{x \mid f(x) \in I_{n,k}\}$ if $k < n2^n$ and $\varnothing$ otherwise:

```
Let A n k := if (k < n * 2 ^ n)%N then
  D `&` [set x | f x \in EFin @` [set` I n k]] else set0.
```

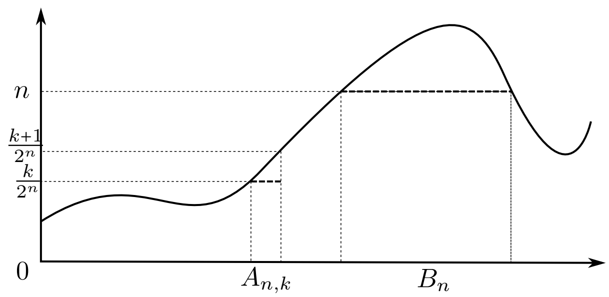


Figure 6.2: Approximation of a measurable function using simple functions

Then, we define nth approximation of f to be the function

$$x \mapsto \sum_{k<n2^n} \frac{k}{2^n} \mathbf{1}_{A_{n,k}}(x) + n\mathbf{1}_{B_n}(x)$$

```
Definition approx : (T -> R)^nat := fun n x =>
  \sum_(k < n * 2 ^ n) k%:R * 2 ^- n * \1_(A n k) x + n%:R * \1_(B n) x.
```

Theorem 6.1.2 (Approximation Theorem)**.** *For any (*1*) measurable set D, any (*2*) function f that is (*3*) measurable and (*4*) non-negative, there exists a (*5*) sequence of non-negative simple functions g that is (*6*) non-decreasing and that (*7*) converges towards f.*

```
Context d (T : measurableType d) (R : realType).
Variables (D : set T) (mD : measurable D)(*1*).
Variables (f : T -> \bar R)(*2*) (mf : measurable_fun D f)(*3*).

Lemma approximation : (forall t, D t -> (0 <= f t)%E) -> (*4*)
  exists g : {nnsfun T >-> R}^nat,                          (*5*)
    nondecreasing_seq (g : (T -> R)^nat) /\                 (*6*)
    (forall x, D x -> EFin \o g ^~ x --> f x).              (*7*)
```

The notation `nondecreasing_seq f` is for

```
{homo f : n m / (n <= m)%nat >-> (n <= m)%O}
```

that we saw in Sect. 3.2. The notation `^~` has been explained in Sect. 2.1.1; `EFin \o g ^~ x` means $\lambda n.g_n(x)$.

6.2 Integral of Measurable Functions

6.2.1 Integral of a Simple Function

Let f be a simple function and μ be a non-negative measure. The integral of f w.r.t. μ is defined by

$$\sum_{x \in \mathbb{R}} x\mu(f^{-1}\{x\}).$$

This definition using summation over a finite support (see Sect. 3.4.7), so that we are truly using the fact that f has a finite image.

```
Variables (T : Type) (R : numDomainType) (mu : set T -> \bar R) (f : T -> R).
Definition sintegral := \sum_(x \in [set: R]) x%:E * mu (f @^-1` [set x]).
```

6.2.2 Integral of a Non-negative Function

Let f be an extended real-valued function over some `measurableType`. Its integral is defined by (note that we are abusing the integral sign notation)

$$\sup_h \left\{ \int_x h(x)(\mathbf{d}\,\mu) \mid \ h \text{ non-negative simple function} \leq f \ \right\}.$$

```
Let nnintegral mu f := ereal_sup [set sintegral mu h |
  h in [set h : {nnsfun T >-> R} | forall x, (h x)%:E <= f x]].
```

The definition does not insist on having f non-negative but this will be necessary to obtain the desired properties.

6.2.3 Integral of a Measurable Function

Let f be a extended real-valued function over some `measurableType`. We define its positive part as the non-negative function $\lambda x. \max(\mathtt{f}(x), 0)$ defined in MathComp-Analysis `numfun.v` with the notation `f ^\+`. Similarly, we define its negative part as the non-negative function $\lambda x. \max(-\mathtt{f}(x), 0)$ with the notation `f ^\-`.

The *integral* of f over the domain D is defined by the difference between the positive part and the negative part of the restriction of f to D:

```
Definition integral mu D f (g := f \_ D) :=
  nnintegral mu (g ^\+) - nnintegral mu (g ^\-).
```

Remember the definition of `f \_ D` from Sect. 4.4.1.

We introduce the ASCII notation `\int[mu]_(x in D) f x` for $\int_{x \in D} f(x)(\mathbf{d}\,\mu)$. Observe that this notation has the same form as the iterated operations. Compare

```
\int[mu]_(x in D) f x
```

with, say,

```
\big[op/idx]_(x in D) f x
```

After all, we expect integration to have properties reminiscent of sums, so we'd better have similar-looking notations and naming conventions to help us name and search lemmas of the forthcoming theory of integration.

6.2.4 Properties of the Integral

Let `f1` and `f2` be two non-negative, measurable, extended real-valued functions with domain `D`. We have the monotone integral property:

```
Lemma ge0_le_integral : (forall x, D x -> f1 x <= f2 x) ->
  \int[mu]_(x in D) f1 x <= \int[mu]_(x in D) f2 x.
```

The proof uses the approximation theorem (Theorem 6.1.2).

6.3 Monotone Convergence Theorem

Informal statement: For any non-decreasing sequence of non-negative measurable functions g_n, we have $\int_{x \in D} (\lim g_n)(x)(\mathbf{d}\,\mu) = \lim(\int_{x \in D} g_n(x)(\mathbf{d}\,\mu))$

The proof of the monotone convergence theorem is in 3 steps:

- proof for simple functions only (Sect. 6.3.1),
- proof for simple functions converging to a measurable function (Sect. 6.3.2),
- proof for measurable functions converging to a measurable function (Sect. 6.3.3).

6.3.1 Monotone Convergence for Simple Functions

For any (*1*) sequence of non-negative simple functions g that is (*3*) nondecreasing and that (*4*) converges towards a (*2*) non-negative simple function f, we have

$$\int_x f(x)(\mathbf{d}\,\mu) = \lim_{n\to\infty} \int_x g_n(x)(\mathbf{d}\,\mu).$$

```
Context d (T : measurableType d) (R : realType).
Variable mu : {measure set T -> \bar R}.
Variables (g : {nnsfun T >-> R}^nat)(*1*) (f : {nnsfun T >-> R})(*2*).
Hypothesis nd_g : forall x, nondecreasing_seq (g^~ x)(*3*).
Hypothesis gf : forall x, g ^~ x --> f x(*4*).

Lemma nd_sintegral_lim : sintegral mu f = lim (sintegral mu \o g).
```

The proof is by proving the `<=` part and the `>=` part. The difficult part is `sintegral mu f <= lim (sintegral mu \o g)`. See MathComp-Analysis `lebesgue_integral.v`.

6.3.2 Monotone Convergence Intermediate Lemma

For any (*1*) function $f: T \to \overline{\mathbb{R}}$ that is (*2*) non-negative and (*3*) measurable, any (*4*) sequence g of non-negative simple functions that is (*5*) non-decreasing and (*6*) converging towards f, we have

$$\int_x f(x)(\mathbf{d}\,\mu) = \lim_{n\to\infty} \int_x g_n(x)(\mathbf{d}\,\mu).$$

```
Context d (T : measurableType d) (R : realType).
Variables (mu : {measure set T -> \bar R}) (f : T -> \bar R)(*1*)
          (g : {nnsfun T >-> R}^nat)(*4*).
Hypothesis f0 : forall x, 0 <= f x(*2*).
Hypothesis mf : measurable_fun setT f(*3*).
Hypothesis nd_g : forall x, nondecreasing_seq (g^~x)(*5*).
Hypothesis gf : forall x, EFin \o g^~x --> f x(*6*).

Lemma nd_ge0_integral_lim : \int[mu]_x f x = lim (sintegral mu \o g).
```

Again, the proof is by proving successively the <= part and the >= part.

6.3.3 Proof of the Monotone Convergence Theorem

For (*1*) any measurable set D and any (*3*) non-decreasing sequence of functions (*2*) $g_n : T \to \overline{\mathbb{R}}$ that are (*4*) measurable and (*5*) non-negative, we have

$$\int_{x\in D} \left(\lim_{n\to\infty} g_n(x)\right)(\mathbf{d}\,\mu) = \lim_{n\to\infty} \int_{x\in D} g_n(x)(\mathbf{d}\,\mu).$$

```
Context d (T : measurableType d) (R : realType).
Variables (mu : {measure set T -> \bar R}) (D : set T).
Variables (mD : measurable D)(*1*) (g : (T -> \bar R)^nat)(*2*).
Hypothesis mg : forall n, measurable_fun D (g n)(*4*).
Hypothesis g0 : forall n x, D x -> 0 <= g n x(*5*).
Hypothesis nd_g : forall x, D x -> nondecreasing_seq (g^~ x)(*3*).

Lemma monotone_convergence :
 \int[mu]_(x in D) lim (g^~ x) = lim (fun n => \int[mu]_(x in D) g n x).
```

Easy Direction

$$\lim_{n\to\infty} \int_{x\in D} g_n(x)(\mathbf{d}\,\mu) \le \int_{x\in D} \left(\lim_{n\to\infty} g_n(x)\right)(\mathbf{d}\,\mu)$$

In Coq, this appears as

```
lim (fun n => \int[mu]_x g n x) <= \int[mu]_x f x
```

where `f` is `fun x => lim (g ^~ x)`. In particular, the domain of integration `D` has been put under the integral by using the notation of restriction of a function (notation `\_`, see Sect. 4.4.1).

The proof is by appealing to properties of sequences of extended real numbers and to the fact that the integral is monotone (Sect. 6.2.4). Indeed, we can use `ge0_le_integral` to show that:

- the sequence of integrals on the left-hand side is non-decreasing:

  ```
  nondecreasing_seq (fun n => \int[mu]_x g n x)
  ```

- each of its terms is upper bounded by the right-hand side:

  ```
  \int[mu]_x g n x <= \int[mu]_x f x
  ```

Therefore, the sequence on the left-hand side is convergent and its limit is bounded by the right-hand side.

Difficult Direction

$$\int_{x\in D} \underbrace{\left(\lim_{n\to\infty} g_n\right)(x)}_{f(x)}(\mathbf{d}\,\mu) \le \lim_{n\to\infty} \int_{x\in D} g_n(x)(\mathbf{d}\,\mu)$$

In Coq, this appears as

```
\int[mu]_x f x <= lim (fun n => \int[mu]_x g n x)
```

with the same proviso as above.

The idea is to build a sequence of non-negative simple functions h_n (see below) that is non-decreasing and such that $h_n \le g_n$ and $\lim_{n\to\infty} h_n = f$.

Then we can use the lemma from Sect. 6.3.2 to show

$$\int_{x\in D} f(x)(\mathbf{d}\,\mu) = \lim_{n\to\infty} \int_{x\in D} h_n(x)(\mathbf{d}\,\mu)$$

which leads to

$$\lim_{n\to\infty} \int_{x\in D} h_n(x)(\mathbf{d}\,\mu) \le \lim_{n\to\infty} \int_{x\in D} g_n(x)(\mathbf{d}\,\mu).$$

and then we are able to conclude by appealing to the monotone properties of limits and of integrals.

So, our problem is to find simple functions h_n such that $\lim_{n\to\infty} h_n = f$.

We approximate (in the sense of the approximation theorem—Sect. 6.1.2) each measurable function g by a function g_2:

```
(* raw functions *)
Let g2' n : (T -> R)^nat := approx setT (g n).
(* same functions but with their properties embedded in types *)
Let g2 n : {nnsfun T >-> R}^nat := nnsfun_approx measurableT (mg n).
```

(Note that `g2'` is a sequence of sequences.) And then use these `g2` functions to create the desired function h that we call `max_g2` because it is defined by taken the max of all `g2` functions:

```
(* raw functions *)
Let max_g2' : (T -> R)^nat :=
  fun k t => (\big[maxr/0]_(i < k) (g2' i k) t)%R.
(* same functions but with their properties embedded in types *)
Let max_g2 : {nnsfun T >-> R}^nat := fun k => bigmax_nnsfun (g2^~ k) k.
```

Does `max_g2`/h has the right properties? (I.e., non-decreasing, upper-bounded by `g`, and converging to `f`.)

- h_n non-decreasing? Yes, essentially because each `g2` is.
- $h_n \leq g_n$? Yes, essentially because $\texttt{g2}_n \leq g_n$.
- $\lim_{n\to\infty} h_n = f$? This is a bit more technical, this is `cvg_max_g2_f` in the MathComp-Analysis file `lebesgue_integral.v`.

How do we prove $\lim_{n\to\infty} h_n = f = \lim_{n\to\infty} g_n$?

- $\lim_{n\to\infty} h_n \leq \lim_{n\to\infty} g_n$ is easy, this is by construction.
- $\lim_{n\to\infty} g_n \leq \lim_{n\to\infty} h_n$ requires a bit of work.
  - Suppose that the right-hand side is $< +\infty$ (otherwise this is obvious)
  - It suffices to prove:

    `\forall n \near \oo, g n t <= lim (EFin \o max_g2 ^~ t)`

    Use the `near=> n` tactic (Sect. 4.7) to get a large enough `n`.
    * If `g n t` is $+\infty$:
      then `(approx D (g n))^~ t` diverges,
      then `lim (EFin \o g2 n ^~ t) = +oo`,
      then `lim (EFin \o max_g2 ^~ t) = +oo`
    * If `g n t` $< +\infty$:
      then `(approx D (g n))^~ t` converges towards `g n t`,
      then `lim (EFin \o g2 n ^~ t) = g n t`,
      we conclude because each `g2` is smaller or equal to `max_g2`.

That concludes the proof of the monotone convergence theorem.

6.4 Fubini's Theorem

In Sect. 1.4, we set our goal as going as far as Fubini's theorem. The reader should now be in a position to read MathComp-Analysis `lebesgue_integral.v` to understand how the proof is carried out.

Fubini's theorem is about a function with two arguments that is measurable and *integrable* (i.e., the integral of its absolute value is not ∞, see definition `integrable`). See also [Affeldt and Cohen, 2022].

As an intermediate theorem, one uses Fubini-Tonelli's theorem, which is a similar statement for non-negative functions. Let us state one part of Fubini-Tonelli's theorem, which states the equality of the integration over the product measure of `m1` and `m2` and of the successive integration over `m2` and then over `m1` (Fubini-Tonelli's theorem is obtained by combining with the symmetric statement):

```
Context d1 d2 (T1 : measurableType d1) (T2 : measurableType d2) (R : realType).
Variables (m1 : {measure set T1 -> \bar R}) (m2 : {measure set T2 -> \bar R}).
Hypothesis sm2 : sigma_finite [set: T2] m2.
Let m := product_measure1 m1 sm2.
Variable f : T1 * T2 -> \bar R.
Hypothesis mf : measurable_fun setT f.
Hypothesis f0 : forall x, 0 <= f x.

Lemma fubini_tonelli1 : \int[m]_z f z = \int[m1]_x \int[m2]_y f (x, y).
```

This is proved by using the approximation theorem (Theorem 6.1.2) and the monotone convergence theorem (Sect. 6.3.3).

Using the approximation theorem, we turn the function `f` into a non-decreasing sequence of non-negative simple functions that converges towards `f`, thus transforming the problem into an integration problem of non-negative simple functions, which is arguably simpler (see lemma `sfun_fubini_tonelli` in the file MathComp-Analysis `lebesgue_integral.v`).

Since non-negative simple functions can be expressed as sums of indicator functions (see lemma `fimfunE` in Sect. 6.1), we can furthermore simplify the problem to the integration of indicator functions (see lemma `indic_fubini_tonelli` in the file MathComp-Analysis `lebesgue_integral.v`).

The formal proof of Fubini's theorem is an application of Fubini-Tonelli's theorem with a bit of "almost everywhere" reasoning, which is provided by the notation `{ae mu, forall x, P x}` where `mu` is a measure and `P` is a predicate in the file MathComp-Analysis `measure.v`. At the time of this writing, the formal proof of Fubini's theorem is the last lemma of the file MathComp-Analysis `lebesgue_integral.v`) of the stable version of MathComp-Analysis.

Bibliography

[Affeldt, 2017] Affeldt, R. (2017). Formal verification using the Coq proof assistant. `https://staff.aist.go.jp/reynald.affeldt/ssrcoq/ssrcoq.pdf`. In Japanese. Slides.

[Affeldt et al., 2017] Affeldt, R., Bertot, Y., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Roux, P., Sakaguchi, K., Stone, Z., Strub, P.-Y., and Théry, L. (2017). Mathematical components compliant analysis library. `https://github.com/math-comp/analysis`. Last stable version 0.5.3 (2022).

[Affeldt and Cohen, 2017] Affeldt, R. and Cohen, C. (2017). Formal foundations of 3D geometry to model robot manipulators. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017), Paris, France, January 16–17, 2017*, pages 30–42. ACM Press. doi.

[Affeldt and Cohen, 2022] Affeldt, R. and Cohen, C. (2022). Measure construction by extension in dependent type theory with application to integration.

[Affeldt et al., 2020a] Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., and Sakaguchi, K. (2020a). Competing inheritance paths in dependent type theory: a case study in functional analysis. In *10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June 29–July 6, 2020*, volume 12167 of *Lecture Notes in Artificial Intelligence*, pages 3–20. Springer. doi.

[Affeldt et al., 2018] Affeldt, R., Cohen, C., and Rouhling, D. (2018). Formalization techniques for asymptotic reasoning in classical analysis. *J. Formalized Reasoning*, 11(1):43–76.

[Affeldt et al., 2023] Affeldt, R., Cohen, C., and Saito, A. (2023). Semantics of probabilistic programs using s-finite kernels in coq. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023), January 16–17, 2023, Boston, Massachusetts, USA*. ACM Press.

[Affeldt et al., 2021] Affeldt, R., Garrigue, J., Nowak, D., and Saikawa, T. (2021). A trustful monad for axiomatic reasoning with probability and nondeterminism. *Journal of Functional Programming*, 31(E17). doi arXiv: cs.LO 2003.09993.

[Affeldt et al., 2020b] Affeldt, R., Garrigue, J., and Saikawa, T. (2020b). A library for formalization of linear error-correcting codes. *Journal of Automated Reasoning*, 64:1123–1164.

[Affeldt et al., 2014] Affeldt, R., Hagiwara, M., and Sénizergues, J. (2014). Formalization of Shannon's theorems. *Journal of Automated Reasoning*, 53(1):63–103.

[Affeldt and Nowak, 2021] Affeldt, R. and Nowak, D. (2021). Extending equational monadic reasoning with monad transformers. In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics*, pages 2:1–2:21. Schloss Dagstuhl.

[Appel, 2022] Appel, A. W. (2022). Coq's vibrant ecosystem for verification engineering (invited talk). In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 2–11. ACM.

[Bernard et al., 2021] Bernard, S., Cohen, C., Mahboubi, A., and Strub, P.-Y. (2021). Unsolvability of the Quintic Formalized in Dependent Type Theory. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

[Bertot and Castéran, 2015] Bertot, Y. and Castéran, P. (2015). *Le Coq'Art (V8)*. In French.

[Bertot et al., 2008] Bertot, Y., Gonthier, G., Biha, S. O., and Pasca, I. (2008). Canonical big operators. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008), Montreal, Canada, August 18–21, 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer.

[Broué, 2012] Broué, I. (2012). *Actes des journées X-UPS 2012*, chapter Quelques Moments de Vie Privilégiés avec Henri et Nicole Cartan. Éditions de l'École polytechnique.

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68.

[Cohen and Sakaguchi, 2015] Cohen, C. and Sakaguchi, K. (2015). A finset and finmap library: Finite sets, finite maps, multisets and order. Available at `https://github.com/math-comp/finmap`. Last stable release: 1.5.0 (2020).

[Cohen et al., 2020] Cohen, C., Sakaguchi, K., and Tassi, E. (2020). Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Corry, 1992] Corry, L. (1992). Nicolas bourbaki and the concept of mathematical structure. *Synthese*, 92(3):315–348.

[Garillot et al., 2009] Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L. (2009). Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer.

[Gonthier, 2008] Gonthier, G. (2008). Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393.

[Gonthier et al., 2013] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. (2013). A machine-checked proof of the odd order theorem. In *4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer.

[Gonthier et al., 2016] Gonthier, G., Mahboubi, A., and Tassi, E. (2016). A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France.

[Gonthier and Tassi, 2012] Gonthier, G. and Tassi, E. (2012). A language of patterns for subterm selection. In *Proceedings of the 3rd International Conference on Interactive Theorem Proving (ITP 2012), Princeton, NJ, USA, August 13–15, 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 361–376. Springer.

[Hagiwara and Affeldt, 2018] Hagiwara, M. and Affeldt, R. (2018). *Formal Proof using Coq/SSReflect/MathComp: Start Formalization of Mathematics with Free Software*. Morikita Publishing. In Japanese.

[Harrison, 2018] Harrison, J. (2018). Let's make set theory great again. Invited talk at the 3rd Conference on Artificial Intelligence and Theorem Proving (AITP 2018), March 25–30, 2018, Aussois, France.

[Howard, 1980] Howard, W. A. (1980). The formulae-as-types notion of construction.

[Infotheo, 2022] Infotheo (2022). A Coq formalization of information theory and linear error-correcting codes. `https://github.com/affeldt-aist/infotheo`. Open source software. Since 2009. Version 0.5.0.

[Mahboubi and Tassi, 2021] Mahboubi, A. and Tassi, E. (2021). *Mathematical Components*. Zenodo.

[Martin-Dorel and Tassi, 2019] Martin-Dorel, E. and Tassi, E. (2019). SSReflect in Coq 8.10. In *The Coq Workshop 2019, September 8, 2019, Portland, OR, USA*.

[Rouhling, 2019] Rouhling, D. (2019). *Outils pour la Formalisation en Analyse Classique: Une Étude de Cas en Théorie du Contrôle*. PhD thesis, Université Côte d'Azur.

[Russell, 1908] Russell, B. (1908). Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262.

[Saito and Affeldt, 2022] Saito, A. and Affeldt, R. (2022). Towards a practical library for monadic equational reasoning in Coq. In *14th International Conference on Mathematics of Program Construction (MPC 2022), Tbilisi, Georgia, September 26–28, 2022*, volume 13544 of *Lecture Notes in Computer Science*, pages 151–177. Springer.

[Sozeau, 2009] Sozeau, M. (2009). Equations—a function definitions plugin. Available at `https://mattam82.github.io/Coq-Equations/`. Last stable release: 1.3 (2021).

[The Coq Development Team, 2022] The Coq Development Team (2022). *The Coq Proof Assistant Reference Manual*. Inria. Available at `https://coq.inria.fr`. Version 8.16.0.

[Voevodsky, 2014] Voevodsky, V. (2014). Univalent foundations. `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2014_IAS.pdf`. Lecture at IAS.

[Whitehead and Russell, 1927] Whitehead, A. N. and Russell, B. A. W. (1927). *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge.

[Wilansky, 2008] Wilansky, A. (2008). *Topology for Analysis*. Dover Publications. Republication. Originally work from 1970.

Appendix A

Cheat Sheets

The following cheat sheets might be useful to memorize naming conventions at first, but after a while the `Search` command and navigation into the appropriate files of MathComp should be a better substitute.

cheat sheet ssrbool.v (Coq v8.16)

ssrfun.v naming conventions

K	cancel
LR	move an op from the lhs of a rel to the rhs
RL	move an op from the rhs to the lhs

ssrfun.v notations

`f ^~ y`	`fun x => f x y`
`p .1`	`fst p`
`p .2`	`snd p`
`f =1 g`	`f x = g x`
`{morph f : x / aF x >-> rR x}`	`f (aF x) = rF (f x)`
`{morph f : x y / aOp x y >-> rOp x y}`	`f (aOp x y) = rOp (f x) (f y)`

ssrfun.v definitions

`injective f`	`forall x1 x2, f x1 = f x2 -> x1 = x2`
`cancel f g`	`g (f x) = x`
`involutive f`	`cancel f f`
`left_injective op`	`injective (op^~ x)`
`right_injective op`	`injective (op y)`
`left_id e op`	`e □ x = x`
`right_id e op`	`x □ e = x`
`left_zero z op`	`z □ x = z`
`right_zero z op`	`x □ z = z`
`self_inverse e op`	`x □ x = e`
`idempotent op`	`x □ x = x`
`commutative op`	`x □ y = y □ x`
`associative op`	`x □ (y □ z) = (x □ y) □ z`
`right_commutative op`	`(x □ y) □ z = (x □ z) □ y`
`left_commutative op`	`x □ (y □ z) = y □ (x □ z)`
`left_distributive op add`	`(x + y) * z = (x * z) + (y * z)`
`right_distributive op add`	`x * (y + z) = (x * y) + (x * z)`
`left_loop inv op`	`cancel (op x) (op (inv x))`

ssrbool.v naming conventions

A	associativity
AC	right commutativity
b	a boolean argument
C	commutativity/complement
D	predicate difference
E	elimination
F/f	boolean false
T/t	boolean truth
U	predicate union

```
(* bool_scope *)
Notation "~~ b":= (negb b)
Notation "b ==> c" := (implb b c).
Notation "b1 (+) b2":= (addb b1 b2).
Notation "a && b":= (andb a b)
Notation "a || b" := (orb a b)
Notation "x \in A" := (in_mem x (mem A)).
Notation "x \notin A" := (~~ (x \in A)).
```

Generalized to `[&& b1 , b2 , ... , bn & c ]` (for `a && b`)

Generalized to `[|| b1 , b2 , ... , bn | c ]` (for `a || b`)

```
negbT      b = false -> ~~ b
negbTE     ~~ b -> b = false
negbK      involutive negb
contra     (c -> b) -> ~~ b -> ~~ c
contraNF   (c -> b) -> ~~ b -> c = false
contraFF   (c -> b) -> b = false -> c = false
ifP        if_spec (b = false) b (if b then vT else vF)
ifT        b -> (if b then vT else vF) = vT
ifF        b = false -> (if b then vT else vF) = vF
ifN        ~~ b -> (if b then vT else vF) = vF
boolP      alt_spec b1 b1 b1
negP       reflect (~ b1) (~~ b1)
negPn      reflect b1 (~~ ~~ b1)
andP       reflect (b1 /\ b2) (b1 && b2)
orP        reflect (b1 \/ b2) (b1 || b2)
nandP      reflect (~~ b1 \/ ~~ b2) (~~ (b1 && b2))
norP       reflect (~~ b1 /\ ~~ b2) (~~ (b1 || b2))
implyP     reflect (b1 -> b2) (b1 ==> b2)
andTb      left_id true andb
andbT      right_id true andb
andbb      idempotent andb
andbC      commutative andb
andbA      associative andb
orFb       left_id false orb
orbN       b || ~~ b = true
negb_and   ~~ (a && b) = ~~ a || ~~ b
negb_or    ~~ (a || b) = ~~ a && ~~ b
```

```
Variant if_spec (not_b : Prop) : bool -> A -> Set :=
  | IfSpecTrue  of       b : if_spec not_b true vT
  | IfSpecFalse of   not_b : if_spec not_b false vF.

Inductive reflect (P : Prop) : bool -> Set :=
  | ReflectT  of    P : reflect P true
  | ReflectF of ~ P : reflect P false.

Variant alt_spec (P : Prop) (b : bool) : bool -> Type :=
  | AltTrue of     P : alt_spec P b true
  | AltFalse of ~~ b : alt_spec P b false.
```

```
Notation xpred0 := (fun=> false).
Notation xpredT := (fun=> true).
Notation xpredU := (fun (p1 p2 : pred _) x => p1 x || p2 x).
Notation xpredC := (fun (p : pred _) x => ~~ p x).
Notation "A =i B" := (eq_mem (mem A) (mem B)).
```

cheat sheet ssrnat.v (SSReflect v1.15)

ssrfun.v naming conventions	
K	cancel
LR	move an op from the lhs of a rel to the rhs
RL	move an op from the rhs to the lhs

ssrfun.v notations	
f ^~ y	fun x => f x y
p .1	fst p
p .2	snd p
f =1 g	f x = g x
{morph f : x / aF x >-> rR x}	f (aF x) = rF (f x)
{morph f : x y / aOp x y >-> rOp x y}	f (aOp x y) = rOp (f x) (f y)

ssrfun.v definitions	
injective f	forall x1 x2, f x1 = f x2 -> x1 = x2
cancel f g	g (f x) = x
involutive f	cancel f f
left_injective op	injective (op^~ x)
right_injective op	injective (op y)
left_id e op	e □ x = x
right_id e op	x □ e = x
left_zero z op	z □ x = z
right_zero z op	x □ z = z
self_inverse e op	x □ x = e
idempotent op	x □ x = x
commutative op	x □ y = y □ x
associative op	x □ (y □ z) = (x □ y) □ z
right_commutative op	(x □ y) □ z = (x □ z) □ y
left_commutative op	x □ (y □ z) = y □ (x □ z)
left_distributive op add	(x + y) * z = (x * z) + (y * z)
right_distributive op add	x * (y + z) = (x * y) + (x * z)
left_loop inv op	cancel (op x) (op (inv x))

ssrbool.v naming conventions	
A	associativity
AC	right commutativity
b	a boolean argument
C	commutativity/complement
D	predicate difference
E	elimination
F/f	boolean false
T/t	boolean truth
U	predicate union

ssrnat.v naming conventions	
A(infix)	conjunction
B	subtraction
D	addition
p(prefix)	positive
S	successor
V(infix)	disjunction

```
(* nat_scope *)
Notation "n .+1":= (succn n).   Notation "n .*2":= (double n).   Notation "n `!":=(factorial n).
Notation "n .-1":= (predn n).   Notation "n ./2":= (half n).
Notation "m <n":= (m.+1 <=n).   Notation "m ^ n":= (expn m n).
```

add0n/addn0	left_id 0 addn/right_id 0 addn
add1n/addn1	1 + n = n.+1/n + 1 = n.+1
addn2	n + 2 = n.+2
addSn	m.+1 + n = (m + n).+1
addnS	m + n.+1 = (m + n).+1
addSnnS	m.+1 + n = m + n.+1
addnC	commutative addn
addnA	associative addn
addnCA	left_commutative addn
eqn_add2l	(p + m == p + n) = (m == n)
eqn_add2r	(m + p == n + p) = (m == n)
sub0n/subn0	left_zero 0 subn/right_id 0 subn
subnn	self_inverse 0 subn
subSS	m.+1 - n.+1 = m - n
subn1	n - 1 = n.-1
subnDl	(p + m) - (p + n) = m - n
subnDr	(m + p) - (n + p) = m - n
addKn	cancel (addn n) (subn^~ n)
addnK	cancel (addn^~ n) (subn^~ n)
subSnn	n.+1 - n = 1
subnDA	n - (m + p) = (n - m) - p
subnAC	right_commutative subn
ltnS	(m < n.+1) = (m <= n)
prednK	0 < n -> n.-1.+1 = n
leqNgt	(m <= n) = ~~ (n < m)
ltnNge	(m < n) = ~~ (n <= m)
ltnn	n < n = false
subnDA	n - (m + p) = (n - m) - p
leq_eqVlt	(m <= n) = (m == n) || (m < n)
ltn_neqAle	(m < n) = (m != n) && (m <=n)
ltn_add2l	(p + m < p + n) = (m < n)
leq_addr	n <= n + m
addn_gt0	(0 < m + n) = (0 < m) || (0 < n)
subn_gt0	(0 < n - m) = (m < n)
leq_sub2r	m <= n -> m - p <= n - p
leq_subLR	(m - n <= p) = (m <= n + p)
ltn_sub2r	p < n -> m < n -> m - p < n - p
ltn_subRL	(n < p - m) = (m + n < p)
subnKC	m <= n -> m + (n - m) = n
subnK	m <= n -> (n - m) + m = n
addnBA	p <= n -> m + (n - p) = m + n - p
subnBA	p <= n -> m - (n - p) = m + p - n
subKn	m <= n -> n - (n - m) = m
mul0n/muln0	left_zero 0 muln/right_zero 0 muln
mul1n/muln1	left_id 1 muln/right_id 1 muln
mul2n/muln2	2 * m = m.*2/m * 2 = m.*2
mulnC	commutative muln
mulnA	associative muln
mulSn	m.+1 * n = n + m * n
mulnS	m * n.+1 = m + m * n
mulnDl	left_distributive muln addn
mulnDr	right_distributive muln addn
mulnBl	left_distributive muln subn
mulnBr	right_distributive muln subn
mulnCA	left_commutative muln
muln_gt0	(0 < m * n) = (0 < m) && (0 < n)
leq_pmulr	n > 0 -> m <= m * n
leq_mul2l	(m * n1 <= m * n2) = (m == 0) || (n1 <= n2)
leq_pmul2r	0 < m -> (n1 * m <= n2 * m) = (n1 <= n2)
ltn_pmul2r	0 < m -> (n1 * m < n2 * m) = (n1 < n2)
leqP	leq_xor_gtn m n (m <= n) (n < m)
ltngtP	compare_nat m n (m < n) (n < m) (m == n)
expn0	m ^ 0 = 1
expn1	m ^ 1 = m
expnS	m ^ n.+1 = m * m ^ n
exp0n	0 < n -> 0 ^ n = 0
exp1n	1 ^ n =1
expnD	m ^ (n1 + n2) =m ^ n1 * m ^ n2
expn_gt0	(0 < m ^ n) = (0 < m) || (n == 0)
fact0	0`! = 1
factS	(n.+1)`! = n.+1 * n`!
odd_add	odd (m + n) = odd m (+) odd n
odd_double_half	odd n + n./2.*2 = n

```
Variant leq_xor_gtn m n : nat -> nat -> nat -> nat -> bool -> bool -> Set :=
  | LeqNotGtn of m <= n : leq_xor_gtn m n m m n n true false
  | GtnNotLeq of n < m  : leq_xor_gtn m n n n m m false true.
Variant compare_nat m n : nat -> nat -> nat -> nat -> bool -> bool -> bool -> bool -> bool -> bool -> Set :=
  | CompareNatLt of m < n : compare_nat m n m m n n false false false true false true
  | CompareNatGt of m > n : compare_nat m n n n m m false false true false true false
  | CompareNatEq of m = n : compare_nat m n m m m m true true true true false false.
```

cheat sheet `bigop.v` (SSReflect v1.15)

`big_morph`	$(\forall x\, y, f(x+y) = f(x) \hat{+} f(y)) \to f(0) = \hat{0} \to f\left(\sum_{\substack{i\leftarrow r\\P(i)}} F(i)\right) = \hat{\sum}_{\substack{i\leftarrow r\\P(i)}} f(F(i))$

Section `Extensionality`

`eq_bigl`	$P_1 =_1 P_2 \to \sum_{\substack{i\leftarrow r\\P_1(i)}} F(i) = \sum_{\substack{i\leftarrow r\\P_2(i)}} F(i)$
`eq_bigr`	$(\forall i, P(i) \to F_1(i) = F_2(i)) \to \sum_{\substack{i\leftarrow r\\P(i)}} F_1(i) = \sum_{\substack{i\leftarrow r\\P(i)}} F_2(i)$
`big_nil`	$\sum_{\substack{i\leftarrow \emptyset\\P(i)}} F(i) = 0$
`big_pred0`	$P =_1 \texttt{xpred0} \to \sum_{\substack{i\leftarrow r\\P(i)}} F(i) = 0$
`big_pred1`	$P =_1 \texttt{pred1}(i) \to \prod_{\substack{j\\P(j)}} F(j) = F(i)$
`big_ord0`	$\sum_{\substack{i<0\\P(i)}} F(i) = 0$
`big_tnth`	$\sum_{\substack{i\leftarrow r\\P(i)}} F(i) = \sum_{\substack{i<\texttt{size}(r)\\P(r_i)}} F(r_i)$
`big_nat_recl`	$m \le n \to \sum_{m\le i<n+1} F(i) = F(m) + \sum_{m\le i<n} F(i+1)$
`big_ord_recl`	$\sum_{i<n+1} F(i) = F(\texttt{ord0}) + \sum_{i<n} F(\texttt{lift}((n+1), \texttt{ord0}, i))$
`big_const_ord`	$\sum_{i<n} x = \texttt{iter}(n, \lambda y. x + y, 0)$

Section `MonoidProperties`

`big1`	$(\forall i, P(i) \to F(i) = 1) \to \prod_{\substack{i\leftarrow r\\P(i)}} F(i) = 1$
`big_nat_recr`	$m \le n \to \prod_{m\le i<n+1} F(i) = \left(\prod_{i<n} F(i)\right) \times F(n)$

Section `Abelian`

`big_split`	$\prod_{\substack{i\leftarrow r\\P(i)}} (F_1(i) \times F_2(i)) = \prod_{\substack{i\leftarrow r\\P(i)}} F_1(i) \times \prod_{\substack{i\leftarrow r\\P(i)}} F_2(i)$
`bigU`	$A \cap B = \emptyset \to \prod_{i\in A\cup B} F(i) = (\prod_{i\in A} F(i)) \times (\prod_{i\in B} F(i))$
`partition_big`	$(\forall i, P(i) \to Q(p(i))) \to \prod_{\substack{i\leftarrow s\\P(i)}} F(i) = \prod_{\substack{j:J\\Q(j)}} \prod_{P(i)\ \substack{i\\p(i)=j}} F(i)$
`reindex_onto`	$(\forall i, P(i) \to h(h'(i)) = i) \to \prod_{\substack{i\\P(i)}} F(i) = \prod_{P(h(j))\ \substack{j\\h'(h(j))=j}} F(h(j))$
`pair_big`	$\prod_{\substack{i\\P(i)}} \prod_{\substack{j\\Q(j)}} F(i,j) = \prod_{\substack{(p,q)\\P(p)\wedge Q(q)}} F(p,q)$
`exchange_big`	$\prod_{\substack{i\leftarrow rI\\P(i)}} \prod_{\substack{j\leftarrow rJ\\Q(j)}} F(i,j) = \prod_{\substack{j\leftarrow rJ\\Q(j)}} \prod_{\substack{i\leftarrow rI\\Q(i)}} F(i,j)$

Section `Distributivity`

`big_distrl`	$\left(\sum_{\substack{i\leftarrow r\\P(i)}} F(i)\right) \times a = \sum_{\substack{i\leftarrow r\\P(i)}} (F(i) \times a)$ (also `big_distrr`)	
`big_distr_big_dep`	$\prod_{\substack{i\\P(i)}} \sum_{\substack{j\\Q(i,j)}} F(i,j) = \sum_{f\in\texttt{pfamily}(j_0,P,Q)} \prod_{\substack{i\\P(i)}} F(i, f(i))$	$\texttt{pfamily}(j_0, P, Q) \simeq$ functions Q^P
`big_distr_big`	$\prod_{\substack{i\\P(i)}} \sum_{\substack{j\\Q(j)}} F(i,j) = \sum_{f\in\texttt{pffun\_on}(j_0,P,Q)} \prod_{\substack{i\\P(i)}} F(i, f(i))$	
`bigA_distr_big`	$\prod_i \sum_{\substack{i\\Q(j)}} F(i, f(i)) = \sum_{f\in\texttt{ffun\_on}(Q)} \prod_i F(i, f(i))$	
`bigA_distr_bigA`	$\prod_{i\in I} \sum_{j\in J} F(i,j) = \sum_{f\in J^I} \prod_{i\in I} F(i, f(i))$	

from `finset.v` also (SSReflect v1.15)

`partition_big_imset`	$\sum_{i\in A} F(i) = \sum_{i\in h\ \texttt{@:}\ A} \sum_{\substack{i\in A\\h(i)=j}} F(i)$
`big_trivIset`	$\texttt{trivIset}(P) \to \sum_{x\in\texttt{cover}(P)} E(x) = \sum_{A\in P} \sum_{x\in A} E(x)$
`partition_disjoint_bigcup`	$(\forall i\, j, i \ne j \to F(i) \cap F(j) = \emptyset) \to \sum_{x\in\bigcup_i F(i)} E(x) = \sum_i \sum_{x\in F(i)} E(x)$

cheat sheet `finset.v` (SSReflect v1.15)

`ssrfun.v` naming conventions	
`K`	cancel
`LR`	move an op from the lhs of a rel to the rhs
`RL`	move an op from the rhs to the lhs

`ssrfun.v` definitions	
`injective f`	`forall x1 x2, f x1 = f x2 -> x1 = x2`
`cancel f g`	`g (f x) = x`
`involutive f`	`cancel f f`
`left_injective op`	`injective (op^~ x)`
`right_injective op`	`injective (op y)`
`left_id e op`	`e □ x = x`
`right_id e op`	`x □ e = x`
`left_zero z op`	`z □ x = z`
`right_zero z op`	`x □ z = z`
`self_inverse e op`	`x □ x = e`
`idempotent op`	`x □ x = x`
`commutative op`	`x □ y = y □ x`
`associative op`	`x □ (y □ z) = (x □ y) □ z`
`right_commutative op`	`(x □ y) □ z = (x □ z) □ y`
`left_commutative op`	`x □ (y □ z) = y □ (x □ z)`
`left_distributive op add`	`(x + y) * z = (x * z) + (y * z)`
`right_distributive op add`	`x * (y + z) = (x * y) + (x * z)`
`left_loop inv op`	`cancel (op x) (op (inv x))`

`ssrbool.v` naming conventions	
`A`	associativity
`AC`	right commutativity
`b`	a boolean argument
`C`	commutativity/complement
`D`	predicate difference
`E`	elimination
`F/f`	boolean false
`T/t`	boolean truth
`U`	predicate union

`finset.v` naming conventions	
`0`	the empty set
`T`	the full set
`1`	singleton set
`C`	complement
`U`	union
`I`	intersection
`D`	difference

`(* set_scope *)`

<table>
<tr><td></td><td><code>set0</code></td><td>$\emptyset$</td></tr>
<tr><td><code>A :|: B</code></td><td><code>setU</code></td><td>$A \cup B$</td></tr>
<tr><td><code>a |: A</code></td><td><code>[set a] :|: A</code></td><td>$\{a\} \cup A$</td></tr>
<tr><td><code>A :&: B</code></td><td><code>setI</code></td><td>$A \cap B$</td></tr>
<tr><td><code>~: A</code></td><td><code>setC A</code></td><td>A^C</td></tr>
<tr><td><code>A :\: B</code></td><td><code>setD A B</code></td><td>$A \backslash B$</td></tr>
<tr><td><code>A :\ a</code></td><td><code>A :\: [set a]</code></td><td>$A \backslash \{a\}$</td></tr>
<tr><td><code>f @^-1: A</code></td><td><code>preimset f (mem A)</code></td><td>$f^{-1}(A)$</td></tr>
<tr><td><code>f @: A</code></td><td><code>imset f (mem A)</code></td><td>$f(A)$</td></tr>
<tr><td><code>f @2: (A , B)</code></td><td><code>imset2 f (mem A) (fun _ =>mem B)</code></td><td>$f(A, B)$</td></tr>
</table>

`(* bool_scope *)`

<table>
<tr><td><code>a \in A</code></td><td>see <code>ssrbool.v</code></td><td>$a \in A$</td></tr>
<tr><td><code>A \subset B</code></td><td>see <code>fintype.v</code></td><td>$A \subseteq B$</td></tr>
<tr><td><code>[disjoint A & B]</code></td><td>see <code>fintype.v</code></td><td>$A \cap B = \emptyset$</td></tr>
</table>

<table>
<tr><td><code>setP</code></td><td><code>A =i B <-> A = B</code></td></tr>
<tr><td><code>in_set0</code></td><td><code>x \in set0 = false</code></td></tr>
<tr><td><code>subset0</code></td><td><code>(A \subset set0) = (A == set0)</code></td></tr>
<tr><td><code>in_set1</code></td><td><code>(x \in [set a]) = (x == a)</code></td></tr>
<tr><td><code>in_setD1</code></td><td><code>(x \in A :\ b) = (x != b) && (x \in A)</code></td></tr>
<tr><td><code>in_setU</code></td><td><code>(x \in A :|: B) = (x \in A) || (x \in B)</code></td></tr>
<tr><td><code>in_setC</code></td><td><code>(x \in ~: A) = (x \notin A)</code></td></tr>
</table>

(NB: `inE` is a multi-rule corresponding to `in_set0`, `in_set1`, `in_setD1`, `in_setU`, `in_setC`, etc.)

<table>
<tr><td><code>setUC</code></td><td><code>A :|: B = B :|: A</code></td></tr>
<tr><td><code>setIC</code></td><td><code>A :&: B = B :&: A</code></td></tr>
<tr><td><code>setKI</code></td><td><code>A :|: (B :&: A) = A</code></td></tr>
<tr><td><code>setCI</code></td><td><code>~: (A :&: B) = ~: A :|: ~: B</code></td></tr>
<tr><td><code>setCK</code></td><td><code>involutive (@setC T)</code></td></tr>
<tr><td><code>setD0</code></td><td><code>A :\: set0 =A</code></td></tr>
<tr><td><code>cardsE</code></td><td><code>#|[set x in pA]| = #|pA|</code> (NB: <code>cardE : #|A|= size (enum A)</code> in <code>fintype.v</code>)</td></tr>
<tr><td><code>cards0</code></td><td><code>#|@set0 T| = 0</code> (NB: <code>card0 : #|@pred0 T|=0</code> in <code>fintype.v</code>)</td></tr>
<tr><td><code>cards_eq0</code></td><td><code>(#|A| == 0) = (A == set0)</code></td></tr>
<tr><td><code>cardsU</code></td><td><code>#|A :|: B| = #|A| + #|B| - #|A :&: B|</code></td></tr>
<tr><td><code>cardsT</code></td><td><code>#|[set: T]| = #|T|</code> (NB: <code>cardT : #|T|= size (enum T)</code> in <code>fintype.v</code>)</td></tr>
<tr><td><code>set0Pn</code></td><td><code>reflect (exists x, x \in A) (A != set0)</code></td></tr>
<tr><td><code>subsetIl</code></td><td><code>A :&: B \subset A</code></td></tr>
<tr><td><code>subsetUr</code></td><td><code>B \subset A :|: B</code></td></tr>
<tr><td><code>subsetI</code></td><td><code>(A \subset B :&: C) = (A \subset B) && (A \subset C)</code></td></tr>
<tr><td><code>setI_eq0</code></td><td><code>(A :&: B == set0) = [disjoint A & B]</code></td></tr>
<tr><td><code>imsetP</code></td><td><code>reflect (exists2 x, in_mem x D & y = f x) (y \in imset f D)</code></td></tr>
<tr><td><code>card_imset</code></td><td><code>injective f ->#|f @: D|=#|D|</code></td></tr>
</table>

Section Partitions

<table>
<tr><td><code>cover P</code></td><td>$\bigcup_{B \in P} B$</td></tr>
<tr><td><code>trivIset P</code></td><td>$\sum_{B \in P} |B| = |\texttt{cover}(P)|$</td></tr>
</table>

see also `bigop_doc.pdf`

Appendix B

Coq and MathComp Installation

In case of emergency, you can use Coq in a web browser, just search for jsCoq ("JavaScript Coq") on the web. Maybe try `https://coq.vercel.app/`.

You can also install the Coq platform. It is a set of compatible packages for Coq that is easy to install.

It is however much more convenient to install Coq from the source code on your computer. You can find installation instructions online, e.g., installation on Linux and Windows. (Please, PR on github if you find any error in these installation notes.)

Coq is typically used through a customizable text editor. The most popular choice is Emacs with the Proof General extension (and possibly also the Company-coq extension). It is arguably the best solution in terms of speed of edition and integration with other tools, in particular in a Unix-like environment such as Linux or MacOS.

CoqIDE is a text editor specific Coq. It is a popular choice for beginners, some advanced users also manage to be productive with it. It comes with the Coq platform.

Visual Studio Code is a more recent alternative. It might be a better choice than Emacs on Windows thanks to a good interaction with WSL. It is however still a bit less stable than Emacs, but definitely *looks* more modern.

List of Tables

List of Figures

Index