Measure Construction by Extension in Dependent Type Theory with Application to Integration

Reynald Affeldt, Cyril Cohen

Measure Construction by Extension in Dependent Type Theory with Application to Integration

Reynald Affeldt[1] and Cyril Cohen[2]

[1]National Institute of Advanced Industrial Science and Technology (AIST)
[2]Université Côte d'Azur and Inria

Abstract

We report on an original formalization of measure and integration theory in the Coq proof assistant. We build the Lebesgue measure following a standard construction that had not yet been formalized in proof assistants based on dependent type theory: by extension of a measure over a semiring of sets. We achieve this formalization by leveraging on existing techniques from the Mathematical Components project. We explain how we extend Mathematical Components' iterated operators and mathematical structures for analysis to provide support for infinite sums and extended real numbers. We introduce new mathematical structures for measure theory and incidentally provide an illustrative, concrete application of Hierarchy-Builder, a generic tool for the formalization of hierarchies of mathematical structures. This formalization of measure theory provides the basis for a new formalization of the Lebesgue integration compatible with the Mathematical Components project.

1 Introduction

Measure theory and integration theory are major topics in mathematics with practical applications. For example, they serve as the foundation of probability theory whose formalization in proof assistants is used to verify information security (e.g., [1]) or artificial intelligence (e.g., [39]). It is therefore no wonder that the topic of formalization of measure and integration theory in proof assistants has already been tackled several times (e.g., [11, 12, 27, 17, 26]). In fact, experiments are still going on [45], some still dealing with the basics [22, 14].

Our motivation is to develop measure and integration theory on top of MathComp [33], a library of formalized mathematics developed using the Coq proof assistant [41]. The MathComp library consists of several algebraic theories that made it possible to formalize the Odd Order theorem[1] by follow-

[1]The Odd Order theorem, a.k.a. the Feit-Thompson Theorem, states that groups of odd

ing its published, revised proof [24, Sect. 6]. There are now several libraries that are built on top of MathComp, the main ones being made available as parts of the Mathematical Components project[2]. Among them, MathComp-Analysis [3, 2] aims at taking advantage of the algebraic theories provided by MathComp to develop classical analysis (topology, real and complex analysis, etc.).

In this paper, we report on an original formalization of measure and integration theory. Our approach is to extend MathComp-Analysis with reusable theories while following textbook presentations [29, 32]. The best illustration is the construction of the Lebesgue measure that we formalize. This is a standard construction from a semiring of sets, using the Measure Extension theorem. To the best of our knowledge, it has never been formalized with the abstraction of ring of sets or semiring of sets in a proof assistant based on dependent type theory. Yet, its formalization is the occasion to develop new mathematical structures of general interest for Coq users. Similarly, the construction of the Lebesgue integral gives us the opportunity to develop a generic formalization of simple functions and to extend the formalization of the iterated operators of MathComp [10], one key to the successful formalization of the Odd Order theorem.

Our contribution in this paper is twofold. First, we bring to the Coq proof assistant a formalization of measure and integration theory that is compatible with the algebraic theories of MathComp. Second, we demonstrate recent formalization techniques developed in the context of the Mathematical Components project. In particular, we use Hierarchy-Builder [19] to formalize a hierarchy of mathematical structures for measure theory and to provide a compositional formalization of simple functions. Our technical contributions materialize as extensions to MathComp-Analysis in the form of reusable formal theories about sequences (of reals and of extended real numbers) and about sums over general sets and over finitely-supported functions.

Paper Outline In Sect. 2, we explain how we develop the theory of extended real numbers by extending MathComp and MathComp-Analysis. In Sect. 3, we explain how we encode the basic definitions of measure theory, demonstrating the use of Hierarchy-Builder. In Sect. 4, we formalize the Measure Extension theorem which shows how to extend a measure over a semiring of sets to a *σ-algebra*. This is a standard and generic approach to the construction of measures. In Sect. 5, we obtain the Lebesgue measure by extending a measure over the semiring of sets of intervals. In Sect. 6, we show that the framework developed so far allows for a formalization of the Lebesgue integral up to the dominated convergence and Fubini's theorems. We review related work in Sect. 7 and conclude in Sect. 8.

order are solvable. This theorem relies on finite group theory, character theory and galois theory.

[2] `https://github.com/math-comp/`

Note on Notation The Mathematical Components project have been favoring ASCII notations. Most of them are unsurprising because they are inspired by LaTeX commands. This paper follows this tradition; ASCII notations will be explained in the prose or in Tables 1 and 2 As a consequence, we can display the Coq code almost verbatim; we allow pretty-printing only for a few standard symbols (such as ← instead of `<-`, → instead of `->`, ∀ instead of `forall`, ∃ instead of `exists`, ≤ instead of `<=`, ≠ instead of `!=`, ∧ instead of `/\`, etc.). The accompanying development [42] can be found online and we will refer to it as a citation possibly indicating the name of the relevant file (as in [42, file `filename.v`]).

2 Support for Extended Real Numbers

Since a measure is potentially infinite, it is represented by extended real numbers. A prerequisite for the construction of measures is therefore the development of the theory of extended real numbers and of their sequences. This actually calls for a substantial extension of MathComp-Analysis [3].

Our starting point is the hierarchy of numeric and real interfaces provided by MathComp and MathComp-Analysis. It contains (among others) the type `numDomainType` for numeric integral domains, the type `numFieldType` for numeric fields, the type `realFieldType` for real fields (see [18, Chapter 4]), and the type `realType` for real numbers). They form an inheritance chain as depicted in Fig. 1.

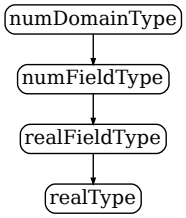

Figure 1: Numeric types provided by MathComp and MathComp-Analysis used in this paper

The definition of extended real numbers is unsurprising (and predates the work presented in this paper):

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

Hereafter, the notation `+oo` (resp. `-oo`) is for the constructor `EPInf` (resp. `ENInf`). The constructor `EFin` injects a real number `r` into the set of extended real numbers; we also use the notation `r%:E` for that purpose. The type `extended R` appears as the notation `\bar R`.

2.1 Algebraic Aspects of Extended Real Numbers

The expression $\infty - \infty$ is undefined in the mathematical practice. How to deal with this is a crucial aspect of our formalization. We define it to be $-\infty$ because it makes the extended real numbers a commutative monoid, so we can use MathComp's iterated operators [10].

Furthermore, we can combine the iterated operators of MathComp the notion of limit, which comes from MathComp-Analysis [3], to introduce a notation for infinite sums. On the one hand, MathComp comes with a generic definition of iterated operators `\big[op/idx]_(i ← s | P i) f i` where `f` is a function whose domain corresponds to the list of indices `s` and `P` is a boolean predicate. Depending on the properties of the binary operator `op` and the element `idx`, many lemmas are available that have been key to important formalizations in Coq (e.g., [24]). The notation `\big[op/idx]_(i < n | P i) f i` is a special case where the indices are the natural numbers less than `n`. As for the notation `\sum_(i ← s | P i) f i`, it is a special case for the iterated addition when `f` is a numerical type-valued function. On the other hand, MathComp-Analysis comes with a definition of limit [3, Sect. 2.3]. It can be applied to sequences, i.e., functions of type `nat → T` (notation `T^nat`). Given a sequence `u`, `lim u` is the limit of the sequence $\mathtt{u}_n$ when $n \to \infty$. We combine these two notations into a family of notations `\sum_(i <oo | P i) f i`, which is simply defined as `lim (fun n ⇒ \big[op/idx]_(i < n | P i) f i)`. Of course, these new notations need to be instrumented with many lemmas, the rest of this paper will provide several examples. Table 1 contains a summary of the notations for iterated operators we have discussed so far[3].

2.2 Topological Aspects of Extended Real Numbers

MathComp-Analysis provides several mathematical structures (topological, uniform, pseudometric spaces, etc.) together with generic lemmas. To enjoy these lemmas, it is necessary to equip extended real numbers with these structures by showing they meet their interfaces.

Extended real numbers form a pseudometric space. The instantiation of the mathematical structures essentially relies on the definition and properties of an order-preserving bijective function from the set of extended real numbers to $[-1; 1]$ (see [42, file `constructive_ereal.v`] for details):

```
Definition contract (x : \bar R) : R :=
  match x with r%:E ⇒ r / (1 + `|r|) | +oo ⇒ 1 | -oo ⇒ -1 end.
```

There is no hope to get a richer structure (say, MathComp's `zmodType`) on the full type though, because as we already discussed above $\infty - \infty$ is taken to be $-\infty$.

[3] Table 1 also contains notations that we will introduce later in this paper. We summarize these notations together to highlight their resemblances and serve as a reading guide.

Table 1: Summary of iterated operators and alike used or newly introduced in this paper. The symbol $\boxed{\texttt{op}}$ is the iterated operator corresponding to `op`.

Finitely iterated operators [10]:	
`\big[op/idx]_(i ← s | P i) f i`	$\boxed{\texttt{op}}_{i<\lvert s\rvert, i\in P} f(s_i)$
`\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 (see Table 2 for application to sets):	
`\sum_(i ← s | P i) f i`	$\sum_{i<\lvert s\rvert, i\in P} f(s_i)$
Countably iterated sum of numeric functions (Sect. 2.1):	
`\sum_(i <oo | P i) f i`	$\sum_{i=0, i\in P}^{\infty} f(i)$
`\sum_(m ≤ i <oo | P i) f i`	$\sum_{i=m, i\in P}^{\infty} f(i)$
Iterated operators over finite supports (Sect. 2.4):	
`\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`
Sum of extended real numbers over general sets (Sect. 2.5):	
`\esum_(i in P) f i`	$\sum_{i\in P} f(i)$
Integral (Sect. 6.4):	
`\int[mu]_(x in D) f x`	$\int_{x\in D} f(x) d\mu(x)$

2.3 Sequences of Extended Real Numbers

The preparatory steps (Sections 2.1 and 2.2) we briefly overviewed above are necessary to produce a theory about sequences of extended real numbers that blends in MathComp-Analysis in a satisfactory way. For the sake of illustration, let us present two sample lemmas. The first one shows that the limit of a sum is the sum of limits:

```
Lemma ereal_limD (R : realType) (f g : (\bar R)^nat) :
  cvg f → cvg g → lim f +? lim g → lim (f \+ g) = lim f + lim g.
```

We already explained the notation `lim` in Sect. 2.1. See Fig. 1 for `realType`. The definition `cvg f` (`cvg` is for "convergence") states that `lim f` exists without naming it explicitly. The notation `a +? b` is a predicate that checks whether the addition of `a` and `b` is well-defined; the notation `f \+ g` is for the pointwise addition of two functions.

The second illustrative lemma shows the commutation of finite and infinite sums of sequences of non-negative terms:

```
Lemma nneseries_sum_nat (R : realType) n (f : nat → nat → \bar R) :
  (∀ i j, 0 ≤ f i j) →
  \sum_(j <oo) (\sum_(0 ≤ i < n) f i j) =
  \sum_(0 ≤ i < n) (\sum_(j <oo) (f i j)).
```

There are many lemmas dealing with sequences of extended real numbers that have been added to MathComp-Analysis for the purpose of this work (see [42, file `sequences.v`] and [42, file `normedtype.v`]). These are reusable lemmas that make the rest of our formalization possible.

2.4 Iterated Operators over Finite Supports

To be able to succinctly formalize some proofs relying on iterated operators, we also extend the library of iterated operators of MathComp-Analysis with *iterated operators over finite supports*. They take the form of the notation `\big[op/idx]_(i \in A) f i` for the iterated application of the operator `op` to `f i`'s where `i` ranges over `A` and `f` as a finite support.

The definition of the finite support of a function relies on a theory about the cardinality properties of sets that was also triggered by the work presented in this paper. From this theory, we use in particular the function `fset_set` (`fset_set A` returns a list when the set `A` is indeed finite and the empty list otherwise) to define the finite support of a function:

```
Definition finite_support {I : choiceType} {T : Type}
    (idx : T) (D : set I) (F : I → T) : seq I :=
  fset_set (D ∩ F @^-1` [set~ idx] : set I).
```

The notation for iterated operators over finite supports combines this definition with MathComp's iterated operators:

```
Notation "\big [ op / idx ]_ ( i '\in' D ) F" :=
    (\big[op/idx]_(i ← finite_support idx D (fun i ⇒ F)) F) :
  big_scope.
```

Table 2: Summary of the set-theoretic notations used in this paper. The type `set T` is defined as `T → Prop`. Most set-theoretic constructs are given ASCII notations, otherwise we use the Coq identifier directly (as with `set0` or `trivIset`).

ASCII notation	Coq identifier	Meaning
`set0`	`set0`	The empty set
`[set: A]`	`setT`	The full set of elements of type `A`
`` `|` ``	`setU`	Set union
`` `&` ``	`setI`	Set intersection
`` `\` ``	`setD`	Set difference
`~`	`setC`	Set complement
`` `<=` ``	`subset`	Set inclusion
`f @` A`	`image`	Image by `f` of `A`
`f @^-1` A`	`preimage`	Preimage by `f` of `A`
`[set x]`	`set1`	The singleton set $\{x\}$
`[set~ x]`	see [42]	The complement of $\{x\}$
`[set E | x in P]`	see [42]	the set of `E` with `x` ranging in `P`
`range f`	see [42]	Image by `f` of the full set
`\big[setU/set0]_ (i ← s | P i) f i`	see Table 1	$\bigcup_{i<\|s\|, i\in P} f(s_i)$
`\bigcup_(k in P) F k`	`bigcup`	Countable union
`\bigcap_(k in P) F k`	`bigcap`	Countable intersection
`trivIset D F`	`trivIset`	`F` is a sequence of pairwise disjoint sets over the domain `D`
`[set` p]`	see [42]	Set corresponding to the boolean predicate `p`

The integral of simple functions in Sect. 6.3 will provide a concrete use of this new notation.

2.5 Sums over General Sets

Last, we extend MathComp-Analysis with *sums over general sets*, i.e.:

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

For that purpose, we introduce the definition `fsets S` for the finite sets included in `S`. It is defined using the predicate `finite_set` which is defined in such a way that `finite_set A` when there is a natural number n such that there is bijection between `A` and the set $\{k \mid k < n\}$, i.e., when the set `A` is finite (see [42, file `cardinality.v`] for details). Using `fsets` and the notation for iterated operators over finite supports from Sect. 2.4, the pencil-and-paper definition of sums over general sets translates directly:

```
Variables (R : realFieldType) (T : choiceType).
Definition fsets S : set (set T) := [set F | finite_set F ∧ F ⊆ S].

Definition esum (S : set T) (a : T → \bar R) :=
  ereal_sup [set \sum_(x \in A) a x | A in fsets S].
```

The type `realFieldType` is one of the numeric types of MathComp (see Fig. 1). The identifier `ereal_sup` corresponds to the supremum of a set of extended real numbers. The definition `esum` is equipped with the notation `\esum_(i in P) f i` of Table 1. It generalizes the notation for the limit of sequences of extended real numbers of Sect. 2.1. As an illustration of the theory of sums over general sets, let us consider the following partition property:

$$J_k \text{ pairwise-disjoint} \to (\forall j, j \in \bigcup_{k \in K} J_k \to 0 \le a_j) \to \sum_{i \in \bigcup_{k \in K} J_k} a_i = \sum_{k \in K} \left(\sum_{j \in J_k} a_j \right).$$

Here follows the corresponding formal statement, where the hypothesis about the pairwise-disjointness of the sets J_k is slightly generalized (see Table 2 for notations):

```
Lemma esum_bigcup J a : trivIset [set k | a @` J k ≠ [set 0]] J →
    (∀ x, (\bigcup_(k in K) J k) x → 0 ≤ a x) →
  \esum_(i in \bigcup_(k in K) J k) a i =
  \esum_(k in K) \esum_(j in J k) a j.
```

This property will turn out to be useful when developing the Measure Extension theorem later in this paper.

3 Basic Definitions of Measure Theory

The main mathematical definitions for measure theory are *σ-algebra* and measure. The goal of the construction of the Lebesgue measure is to build a function that satisfies the properties of a measure. This is not trivial because such a function does not exist in general when the domain is an arbitrary powerset, hence the introduction of *σ-algebra*s.

This section proposes a formalization of the basic definitions of measure theory using Hierarchy-Builder [19], a tool that automates the writing of *packed classes* [23], a methodology to build hierarchies of mathematical structures that is used pervasively in the Mathematical Components project.

3.1 Overview of Hierarchy-Builder

Hierarchy-Builder extends Coq with commands to define hierarchies of mathematical structures and functions. It is designed so that hierarchies can evolve (for example by splitting a structure into smaller structures) without breaking existing code. These commands are compiled to packed classes [23], but the technical details of their implementation in Coq (modules, records, coercions, implicit arguments, canonical structures instances, notations, etc.) are hidden to the user.

The main concept of Hierarchy-Builder is the one of *factory*. This is a record defined by the command `HB.factory` that packs a carrier, operations, and properties. This record usually corresponds to the standard definition of a mathematical structure. *Mixins* are factories used as the default definition for a mathematical structure; they are defined by the command `HB.mixin`. *Structures* defined by the command `HB.structure` are essentially sigma-types with a carrier paired with one or more factories. A mixin often extends a structure, so it typically takes as parameters a carrier and other structures.

Factories are instantiated using the command `HB.instance`. Instances are built with an `xyz.Build` function which is automatically generated for each `xyz` factory.

A *builder* is a function that shows that a factory is sufficient to build one or several mixins. To add builders, one uses the command `HB.builders` that opens a Coq section which starts with postulating a factory instance and lets the user declare several instances of mixins as builders.

In addition to commands to build hierarchies, Hierarchy-Builder also checks their validity by detecting missing interfaces or *competing inheritance paths* [2]. More than an inheritance mechanism, Hierarchy-Builder therefore provides help in the design of hierarchies of structures.

3.2 Mathematical Structures for Measure Theory

A *σ-algebra* is a mathematical structure that comprises a set of sets that contains the empty set, and is stable by complement and by countable union. It

is best defined as a hierarchy of mathematical structures because more general structures actually play a key role in the construction by extension of the Lebesgue measure.

3.2.1 Inheritance Chain from Semiring of Sets

The hierarchy of mathematical structures for measure theory starts with *semirings of sets*. They are formalized using HIERARCHY-BUILDER (see Sect. 3.1) as follows:

```
1  HB.mixin Record isSemiRingOfSets (d : measure_display) T := {
2    measurable : set (set T) ;
3    measurable0 : measurable set0 ;
4    measurableI : setI_closed measurable;
5    semi_measurableD : semi_setD_closed measurable }.
6
7  #[short(type=semiRingOfSetsType)]
8  HB.structure Definition SemiRingOfSets d :=
9    {T of isSemiRingOfSets d T}.
```

The declaration of the mixin starts at line 1. The parameter `d` is what we call a *display parameter*. It can be ignored on a first reading because it is not used in the definition; it is used implement user-friendly notations as will be explained in Sect. 3.4 where we will have enough material to demonstrate its use with a concrete example. Line 2 corresponds to the carrier. A semiring of sets contains the empty set (line 3). It is also stable by finite intersection; this is captured by line 4, where `setI_closed G` is formally defined as `∀ A B, G A → G B → G (A ∩ B)`. At line 5, `semi_setD_closed G` means that the relative complement of two sets in `G` can be partitioned into a finite number of sets in `G`:

```
Definition semi_setD_closed G := ∀ A B, G A → G B → ∃ D,
 [∧ finite_set D, D ⊆ G, A `\` B = \bigcup_(X in D) X & trivIset D id].
```

The definition of semiring of sets is completed at line 8 by declaring the structure (as explained in Sect. 3.1) and providing a conventional notation for the corresponding type (line 7). Hereafter, we call *measurable sets* the sets that form a semiring of sets.

A *ring of sets* is a non-empty set of sets that is closed under union and difference. It can be defined by extending a semiring of sets with the axiom that it is stable by finite union. Its interface can be defined using HIERARCHY-BUILDER as follows:

```
1  HB.mixin Record SemiRingOfSets_isRingOfSets d T
2      of SemiRingOfSets d T := {
3    measurableU : @setU_closed T measurable }.
4
5  #[short(type=ringOfSetsType)]
6  HB.structure Definition RingOfSets d :=
7    {T of SemiRingOfSets_isRingOfSets d T & SemiRingOfSets d T }.
```

This declaration provides a new mixin that extends the mixin for semiring of sets (note the `of` declaration at line 2). At line 3, the expression `setU_closed G` means that the class `G` is stable by finite unions and is formally defined as `∀ A B, G A → G B → G (A ∪ B)`. The modifier `@` at line 3 is Coq syntax to enforce the explicit input of implicit arguments. The corresponding structure is declared at line 6 where it is marked as satisfying the mixin `SemiRingOfSets_isRingOfSets` and extending the structure of semiring of sets `SemiRingOfSets` (line 7).

An *algebra of sets* is a set of sets that contains the empty set and is stable by (finite) union and complement. Algebras of sets are defined as extending rings of sets with the axiom that the full set belongs to the set of measurable sets. The Hierarchy-Builder declaration is similar to the one of semiring of sets and ring of sets:

```
HB.mixin Record RingOfSets_isAlgebraOfSets d T of RingOfSets d T :=
  { measurableT : measurable [set: T] }.

#[short(type=algebraOfSetsType)]
HB.structure Definition AlgebraOfSets :=
  {T of RingOfSets_isAlgebraOfSets T & RingOfSets d T}.
```

Finally, σ-*algebra*s are defined by extending algebras of sets with the axiom of stability by countable union:

```
HB.mixin Record Measurable_from_algebraOfSets d T
    of AlgebraOfSets d T :=
  { bigcupT_measurable :
    ∀ F, (∀ k, measurable (F k)) → measurable (\bigcup_k (F k)) }.

#[short(type=measurableType)]
HB.structure Definition Measurable :=
  {T of Measurable_from_algebraOfSets d T & AlgebraOfSets d T}.
```

These definitions form an inheritance chain (Fig. 2), so that σ-*algebra*s are also algebras of sets, which are also rings of sets, and therefore semirings of sets.

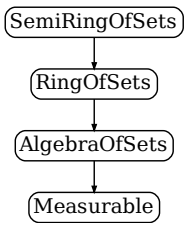

Figure 2: Inheritance chain from semiring of sets to σ-*algebra*'s

3.2.2 Direct Definition of Mesurable Spaces

The set of interfaces provided by the hierarchy of mathematical structures for measure theory is not the only way to instantiate structures. We also provide

factories (introduced in Sect. 3.1). For example, the following factory provides an alternative interface for *σ-algebra*s:

```
HB.factory Record isMeasurable d T := {
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableC : ∀ A, measurable A → measurable (~‘ A) ;
  measurable_bigcup :
    ∀ F, (∀ k, measurable (F k)) → measurable (\bigcup_k (F k)) }.
```

It is arguably closer to the textbook definition that we gave at the beginning of Sect. 3.2. In fact, though it may seen at first sight that mixins provide the definition of mathematical structures, we have observed in practice that the standard textbook definition often ought better be sought in factories provided afterwards.

3.3 Generated *σ-algebra*s

The notion of *generated σ-algebra* will come in handy to define the Measure Extension theorem and to develop the theory of measurable functions. The generated *σ-algebra* `σ≪ D, G ≫` is the smallest *σ-algebra* that contains the set of sets `G`, such that the complement is taken w.r.t. a set `D`. This is defined using a generic `smallest` predicate:

```
Definition smallest C G := \bigcap_(A in [set M | C M ∧ G ⊆ M]) A.
...
Context {T}.
Definition sigma_algebra D G := [∧ G set0, (∀ A, G A → G (D ‘\‘ A)) &
  (∀ A : (set T)^nat, (∀ n, G (A n)) → G (\bigcup_k A k))].
...
Notation "’σ≪’ D , G ’≫’" := (smallest (sigma_algebra D) G).
```

Below, the notation `σ≪ G ≫` is for the measurable sets of the *σ-algebra* generated from the set of sets `G` with complement take w.r.t. the full set.

Note that the definition `smallest` is well-defined (i.e., is indeed the smallest set in the class `C`) whenever the smallest fixpoint of the class `C` indeed exists. This is why the definition of a generated *σ-algebra* can also be found elsewhere [14, Sect. 4.2] defined as an inductive predicate instead. The choice of using the `smallest` predicate rather than an inductive definition is for the sake of genericity: we have a unique function symbol and a common theory to deal with all generated classes (Dynkin, *σ-algebra*, etc.), and since `smallest` itself is monotonous, we can reduce comparison of generated classes to the extent of the classes themselves. However, this has the drawback that the elimination principle and correctness lemmas are not automatically proven by Coq as they would with the `Inductive` command.

Since the set of sets of type `σ≪ G ≫` forms a *σ-algebra* we can equip it with the structure of `measurableType` from Sect. 3.2.1. The temptation would be to give a definition to the measurable type generated by `G`. However, for the sake of modularity it is a common practice in the MathComp libraries to define a

type alias of `T` (of type `pointedType`, i.e., it contains at least one element) with `G` as a phantom argument and provide a inferable instance of measurable type on this alias. Hence we introduce a dedicated identifier `salgebraType` (line 1 below) to alias `T` and a dedicated display parameter `sigma_display` (line 2) (remember from Sect. 3.2.1 that mathematical structures for measure theory are parameterized by a display parameter to be explained in Sect. 3.4). Let us furthermore assume that we are given the proofs `sigma_algebra{0,C,_bigcup}` corresponding to the *σ-algebra* properties of a generated *σ-algebra*. To associate to the identifier `salgebraType` a structure of *σ-algebra*, we use the HIERARCHY-BUILDER command `HB.instance`. At line 8, it equips `salgebraType` with a structure of pointed type (`Pointed.on` replicates the pointed type structure of the underlying type, here `T`). At line 9, it is used with the constructor of the factory `isMeasurable` of Sect. 3.2.2. The corresponding display appears at line 10 and the proofs of the *σ-algebra* properties appear at lines 12–13.

```
1  Definition salgebraType {T} (G : set (set T)) := T.
2  Definition sigma_display {T} : set (set T) → measure_display.
3  Proof. exact. Qed.
4
5  Section g_salgebra_instance.
6  Variables (T : pointedType) (G : set (set T)).
7
8  HB.instance Definition _ := Pointed.on (salgebraType G).
9  HB.instance Definition _ := @isMeasurable.Build
10   (sigma_display G)
11   (salgebraType G) σ≪ G ≫
12   (@sigma_algebra0 _ setT G) (@sigma_algebraC)
13   (@sigma_algebra_bigcup _ setT G).
14
15 End g_salgebra_instance.
```

3.4 Displays for Measurable Types

We saw in the previous sections that the structures for measure theory are parameterized by a display parameter. Its purpose is to disambiguate the printing of expressions of the (input) form `measurable A`. This is useful when several of them appear in the same local context or when `A` does not provide enough information to infer the right measurable type.

More concretely, let us consider the basic case of a measurable type `T` with display `d` (e.g., `T : ringOfSetsType d`). To assert that a set `A : set T` is measurable, one can always write `measurable A`. Yet, the display mechanism is such that COQ prints back `d.-measurable A`. This is achieved by providing a type for displays (line 1) and a notation (line 6):

```
1  Inductive measure_display := default_measure_display.
2
3  Declare Scope measure_display_scope.
4  Delimit Scope measure_display_scope with mdisp.
```

```
5
6  Notation "d .-measurable" := (@measurable d%mdisp).
```

The display mechanism can be used to disambiguate expressions. Let us consider the case of generated σ-*algebra*'s. We saw that the display for generated σ-*algebra*'s is parameterized by the generator set (`sigma_display` in the previous section—Sect. 3.3). We can therefore introduce a notation `G.-sigma` for the display associated with the generator set `G` and a notation `G.-sigma.-measurable` for the measurable sets of the σ-*algebra* generated by `G`:

```
Notation "G .-sigma" := (sigma_display G) : measure_display_scope.
Notation "G .-sigma.-measurable" :=
  (measurable : set (set (salgebraType G))) : classical_set_scope.
```

For example, we can use these notations to regard the empty set `set0` as a member of the σ-*algebra* generated by any set `G`:

```
Goal ∀ (T : pointedType) (G : set (set T)), G.-sigma.-measurable set0.
Proof. by move⇒ T G; exact: measurable0. Qed.
```

In comparison, the input `measurable set0` would not type check, because `T` does not have a default instance to give a meaning to `measurable`.

3.5 Functions on Classes of Sets

There are several notions of functions from classes of sets to the real numbers (or, implicitly, extended reals) which fall under the umbrella name of "measure". In the literature, they are named *contents* (a.k.a. additive measures), premeasures, outer measures, σ-*subadditive* measures, and σ-*additive* measures (a.k.a. measures). We define predicates for all of these notions, but we only define structures for the three most useful: contents, measures, and outer measures.

3.5.1 Contents

A *content* (or an additive measure) μ is a non-negative function defined over a semiring of sets such that the measure of the empty set is 0 and such that $\mu(\cup_{k=1}^{n} F_k) = \sum_{k=1}^{n} \mu(F_k)$ for a finite number of pairwise-disjoint measurable sets F. We first provide a definition for the latter condition:

```
Definition semi_additive mu := ∀ F n, (∀ k, measurable (F k)) →
  trivIset setT F → measurable (\big[setU/set0]_(k < n) F k) →
  mu (\big[setU/set0]_(k < n) F k) = \sum_(k < n) mu (F k).
```

The pairwise-disjointness of sets is captured by the generic predicate `trivIset` (Table 2). Asking $\cup_{k=1}^{n} F_k$ to be measurable is superfluous when taken on a ring of sets. Contents are eventually defined by the following mixin and structure:

```
HB.mixin Record isContent d (T : semiRingOfSetsType d)
    (R : numFieldType) (mu : set T → \bar R) := {
  measure_ge0 : ∀ x, 0 ≤ mu x ;
  measure_semi_additive : semi_additive mu }.
```

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

See Fig. 1 for the type `numFieldType`. In the Coq code, the type of contents is denoted by `{content set T → \bar R}`.

An essential property of contents is that they can be extended from a semiring of sets $\mathcal{S}$ to its generated ring of sets $R(\mathcal{S})$. We can define the latter similarly to how we defined generated *σ-algebra*s in Sect. 3.3:

```
Definition setring G := [∧ G set0, setU_closed G & setDI_closed G].
Notation "'ρ≪' G '≫'" := (smallest setring G).
```

The predicate `setDI_closed` is for stability by set difference and is defined by `∀ A B, G A → G B → G (A `\` B)`.

A generated ring of sets can be equipped with a canonical structure of ring of sets. It happens that the measurable sets of these generated rings of sets can in fact be expressed as the finite disjoint unions of (non-empty) sets from the original semiring of sets $\mathcal{S}$ (`rT` in the lemma below indicates sets from the generated ring of sets):

```
Lemma ring_finite_set (A : set rT) : measurable A → ∃ B : set (set T),
  [∧ finite_set B,
      (∀ X, B X → X ≠set0),
      trivIset B id,
      (∀ X, X \in B → measurable X) &
      A = \bigcup_(X in B) X].
```

Thanks to this lemma, we can make this decomposition explicit by the following function `decomp`, which given a set A in $R(\mathcal{S})$ returns a finite set of sets in $\mathcal{S}$ that cover A:

```
Definition decomp (A : set rT) : set (set T) :=
  if A == set0 then [set set0]
  else if pselect (measurable A) is left mA then
    projT1 (cid (ring_finite_set mA))
  else [set A].
```

The function `decomp` is written in an idiomatic way to retrieve in Coq a witness from an existential proof. The identifier `pselect` comes from MathComp-Analysis and is a strong version of the law of excluded middle [3, Sect. 5.2]; `cid` is the axiom of constructive indefinite description.

Using `decomp`, we can extend the function over the original semiring of sets by summing the components:

```
Definition measure (R : numDomainType) (mu : set T → \bar R)
  (A : set rT) : \bar R := \sum_(X \in decomp A) mu X.
```

We thus have a `measure mu` function for all functions `mu`, which is equal to `mu` on the sets of the semiring of sets where `mu` is defined, and which is a content on the generated ring of sets when `mu` is a content (section `content` in [42, file `measure.v`]), and which is *σ-additive* if `mu` is *σ-subadditive* (lemma `ring_semi_sigma_additive`). Furthermore, using the latter fact we prove that

when `mu` is *σ-subadditive*on a semiring of sets, it is in fact *σ-additive* (lemma `semiring_sigma_additive`).

3.5.2 Measures

A *measure* μ is defined similarly to a content. The difference is the additivity axiom: it is such that $\mu(\cup_k F_k) = \sum_k \mu(F_k)$ for any sequence F of pairwise-disjoint measurable sets. We provide a definition for the latter condition, but generalizing it for semirings of sets by requiring the union $\cup_k F_k$ to be measurable as a precondition, thus merging the notions of measure and premeasure into one:

```
Definition semi_sigma_additive mu := ∀ F, (∀ k, measurable (F k)) →
  trivIset setT F → measurable (\bigcup_k F k) →
  (fun n ⇒ \sum_(k < n) mu (F k)) ⇝ mu (\bigcup_k F k).
```

The notation `f ⇝ l` is a notation for convergence of functions that comes from MathComp-Analysis [3]. In particular, when `f ⇝ l` holds, we have `lim f = l` using the `lim` notation of Sect. 2 (provided that the range of the function `f` is a separated space, which is the case for the functions considered in this section). Note that in the definition above the precondition `measurable (\bigcup_k F k)` holds unconditionally whenever we know that the underlying type is a *σ-algebra*.

We use this definition to define the mixin corresponding to measures, which extends the one for contents:

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

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

In practice, to construct a measure, one would rather use the following factory (we introduced the notion of factory in Sect. 3.2.2) whose interface is closer to the textbook definition of measure:

```
HB.factory Record isMeasure d (T : semiRingOfSetsType d)
   (R : realFieldType) (mu : set T → \bar R) := {
  measure0 : mu set0 = 0 ;
  measure_ge0 : ∀ x, 0 ≤ mu x ;
  measure_semi_sigma_additive : semi_sigma_additive mu }.
```

The notation `{measure set T → \bar R}` corresponds to the type of measures.

3.5.3 Outer Measures

Outer measures are the object of study of the measure extension theorems. Contrarily to a measure, an outer measure `mu` is *σ-subadditive* on the full powerset rather than on a specific class of sets.

```
Definition sigma_subadditive
    (R : numFieldType) (T : Type) (mu : set T → \bar R) :=
  ∀ (F : (set T)^nat), mu (\bigcup_n (F n)) ≤ \sum_(n <oo) mu (F n).
```

Compared to *σ-additivity*, in *σ-subadditivity* the relation between the measure of the countable union and the sum of the measures is an inequality, there are no conditions on the sequence of sets, and the support type need not be a *σ-algebra*. Like for contents and measures (Sections 3.5.1 and 3.5.2), we encode an outer measure as a HIERARCHY-BUILDER mixin:

```
HB.mixin Record isOuterMeasure
    (R : numFieldType) (T : Type) (mu : set T → \bar R) := {
  outer_measure0 : mu set0 = 0 ;
  outer_measure_ge0 : ∀ x, 0 ≤ mu x ;
  le_outer_measure : {homo mu : A B / A ⊆ B ⟼ A ≤ B} ;
  outer_measure_sigma_subadditive : sigma_subadditive mu }.
```

The notation `{homo f : x y / r x y ⟼ s x y}` is a generic MATHCOMP notation for homomorphisms `f` with respect to the relations `r` and `s`. The type of outer measures comes with the notation `{outer_measure set T → \bar R}`.

4 Measure Extension

A standard approach to the construction of measures is to extend a function over a semiring of sets, a ring of sets, or an algebra of sets to a measure over an enclosing *σ-algebra*. These extension theorems are known under different names (Carathéodory/Carathéodory-Fréchet/Carathéodory-Hopf/Hopf/Hahn/Hahn-Kolmogorov/etc. extension theorems). In the following, we explain the formalization of a version starting from semiring of sets and refer to it as the Measure Extension theorem.

As in the textbooks we follow [29, 32], we decompose the Measure Extension theorem in reusable constructions and lemmas. The first, which we refer to as the *outer measure construction*, extends a non-negative function μ such that $\mu(\emptyset) = 0$ over a semiring of sets $\mathcal{S}$ to an outer measure (Sect. 4.1). This is then shown to be a measure over the *σ-algebra* of *Carathéodory-measurable sets* (Sect. 4.2). When restricted to this *σ-algebra*, we call it the *Carathéodory measure*. Now, if μ was a *σ-subadditive* content on $\mathcal{S}$, the *σ-algebra* of Carathéodory-measurable sets contains the *σ-algebra* generated by $\mathcal{S}$, and the Carathéodory measure is uniquely determined on it, by the values of μ on $\mathcal{S}$ (Sect. 4.3).

4.1 Outer Measure Construction

The first part of the Measure Extension theorem builds an outer measure (Sect. 3.5.3) given a function defined over a semiring of sets. In textbooks it is often stated in a weaker form starting from a ring of sets or an algebra of sets. The outer measure in question is more precisely defined as the infimum of the measures of covers, i.e., $\inf_F \{\sum_{k=0}^{\infty} \mu(F_k) \mid (\forall k, \texttt{measurable}(F_k)) \wedge X \subseteq \bigcup_k F_k\}$. The definition of these coverable measures translates directly in MATHCOMP-ANALYSIS:

```
Definition measurable_cover X :=
  [set F | (∀ k, measurable (F k)) ∧ X ⊆ \bigcup_k (F k)].
```

We use `measurable_cover` to define the desired outer measure:

```
Context d (T : semiRingOfSetsType d) (R : realType).
Variable mu : set T → \bar R.
Definition mu_ext (X : set T) : \bar R :=
  ereal_inf [set \sum_(k <oo) mu (F k) | F in measurable_cover X].
```

The identifier `ereal_inf` corresponds to the infimum of a set of extended real numbers. In the following, `mu_ext mu` is noted `mu^*`.

The difficulty to show that `mu^*` is an outer measure is to show that it is *σ-subadditive* (remember that we are working under the hypotheses that `mu set0 = 0` and that `mu` is non-negative). A typical textbook proof [32, Sect. X.1][29, Lemma 1.47] translates to a proof script of 54 lines of code (lemma `mu_ext_sigma_subadditive`, [42, file `measure.v`]). The main technical point is the use of sums over general sets. Precisely, in the course of proving *σ-subadditivity*, we run into a subgoal of the following shape (μ^* is the outer measure under construction):

$$\mu^*(\cup_i F_i) \leq \sum_i^\infty \left(\mu^*(F_i) + \frac{\varepsilon}{2^i}\right).$$

The proof goes on by showing $\mu^*(\cup_i F_i) \leq \sum_{i,j} \mu(G_i j) \leq \sum_i \sum_j \mu(G_i\, j)$ for some well-chosen G, such that $F_i \subseteq \cup_j G_i j$ and $\sum_j \mu(F_i j) \leq \mu^*(F_i) + \varepsilon/2^i$. This proof can be completed with the partition property using sums over general sets from Sect. 2.5.

Coming back to `mu^*`, we also show that it coincides with the input measure `mu` (lemma `measurable_mu_extE` in [42, file `measure.v`]).

4.2 From an Outer Measure to a Measure

The second part of the Measure Extension theorem builds, given an outer measure, a *σ-algebra* and a measure over it. The resulting *σ-algebra* is formed of Carathéodory measurable sets, i.e., sets A such that $\forall X, \mu^*(X) = \mu^*(X \cap A) + \mu^*(X \cap \bar{A})$ where μ^* is an outer measure. Hereafter, the set of Carathéodory measurable sets for an outer measure `mu` will appear as the notation `mu.-caratheodory`.

Given our newly developed theory of sequences of extended real numbers (Sect. 2.3), proving, for an outer measure `mu`, that `mu.-caratheodory` is actually a *σ-algebra* is essentially a translation of standard pencil-and-paper proofs (see lemmas `caratheodory_measurable_{set0,setC,bigcup}` in [42, file `measure.v`]). Hereafter, the *σ-algebra* of Carathéodory measurable sets is denoted by `mu.-cara.-measurable` (this notation is implemented using the display mechanism explained in Sect. 3.4).

Similarly, proving that the restriction of the outer measure `mu` to the *σ-algebra* `mu.-cara.-measurable` is a measure is also essentially a direct translation of standard pencil-and-paper proofs (see lemmas `caratheodory_measure{0,_ge0,_sigma_additive}`).

Finally, we formally prove a number of properties about the resulting measure, in particular that it is *complete*, i.e., *negligible sets* are measurable. Let `T`

be a semiring of sets and `R` be a `realFieldType`. A set N is negligible for μ when there exists a measurable set A such that $\mu(A) = 0$ and $N \subseteq A$:

```
Definition negligible (mu : set T → \bar R) N :=
  ∃ A, [∧ measurable A, mu A = 0 & N ⊆ A].
```

Let `mu.-negligible X` be a notation for `X` is negligible. The formal definition of a complete measure follows:

```
Definition measure_is_complete (mu : set T → \bar R) :=
  mu.-negligible ⊆ measurable.
```

4.3 The Measure Extension Theorem

Finally, we show that a measure over a semiring of sets can be extended to a measure over a *σ-algebra* that contains all the measurable sets of the smallest *σ-algebra* containing the semiring of sets. We place ourselves in the following context:

```
Context d (T : semiRingOfSetsType d) (R : realType).
Variable mu : {measure set T → \bar R}.
```

In this context, we can build an outer measure `mu^*` using the results of Sect. 4.1 and its *σ-algebra* `mu^*.-cara.-measurable` using the results of Sect. 4.2. We can show that this *σ-algebra* contains all the measurable sets generated from the semiring of sets:

```
Lemma sub_caratheodory :
  (d.-measurable).-sigma.-measurable ⊆ mu^*.-cara.-measurable.
```

Recall from Sect. 3.4 that `G.-sigma.-measurable` corresponds to the *σ-algebra* generated from `G` and that in our context `d.-measurable` corresponds to the measurable sets of the semiring of sets `T`. As for `m.-cara.-measurable`, we saw in Sect. 4.2 that it corresponds to the *σ-algebra* of Carathéodory measurable sets for the outer measure `m`.

We use this last fact to build a measure over the *σ-algebra* generated from the semiring of sets: this is the final result of the Measure Extension Theorem (recall from Sect. 3.3 that `salgebraType G` is the measurable type generated by `G`):

```
Let I := salgebraType (@measurable _ T).
Let measure_extension : set I → \bar R := mu^*.

HB.instance Definition _ := isMeasure.Build _ _ _ measure_extension
  measure_extension0 measure_extension_ge0
  measure_extension_semi_sigma_additive.
```

The proofs `measure_extension{0,ge0,_sigma_additive}` correspond to the properties of a measure as explained in Sect. 3.5.2. See [42, file `measure.v`] for details.

Furthermore, we prove that the measure extension is unique. This requires to prove beforehand the uniqueness of measures [42, lemma `measure_unique`].

We use monotone classes for that purpose [32, Sect. V.2.1]. This can also be proved using the equivalent notion of Dynkin systems (as mentioned in [26], which we also formalized in [42, file `measure.v`]). The uniqueness of measure extension is under the condition that the measure is *σ-finite*, i.e., the full set can be covered by a countable union of sets of finite measure:

```
Definition sigma_finite (A : set T) (mu : set T → \bar R) :=
  ∃ F : (set T)^nat, A = \bigcup_(i : nat) F i &
    ∀ i, measurable (F i) ∧ mu (F i) < +oo.
```

When this holds for the measure of the measure extension, any other measure `mu'` that coincides with `mu` on the original semiring of sets also coincides with the measure extension over the generated *σ-algebra*:

```
Lemma measure_extension_unique : sigma_finite [set: T] mu →
  (∀ mu' : {measure set I → \bar R},
    (∀ X, d.-measurable X → mu X = mu' X) →
    (∀ X, (d.-measurable).-sigma.-measurable X →
      measure_extension X = mu' X)).
```

Since *σ-finite* measures are actually pervasive in measure theory, we introduce, we extend using HIERARCHY-BUILDER the hierarchy of structures for contents and measures of Sections 3.5.1 and 3.5.2 with a structure `SigmaFiniteContent` for contents that are *σ-finite* and a structure `SigmaFiniteMeasure` for measures that are *σ-finite*. In particular, hereafter, `{sigma_finite_measure set T → \bar R}` is a notation for the type of *σ-finite* measures.

5 Construction of the Lebesgue Measure over a Semiring of Sets

In this section, we explain how we derive the Lebesgue measure from the semiring of sets of intervals of the form $]a, b]$ using the measure extension from the previous section.

5.1 The semiring of sets of Intervals

In MATHCOMP, the type `interval R`, where `R` is typically an ordered type, is defined as the pairs of bounds of type `itv_bound`:

```
Variant itv_bound (T : Type) : Type :=
  BSide : bool → T → itv_bound T | BInfty : bool → itv_bound T.
Variant interval (T : Type) := Interval of itv_bound T & itv_bound T.
```

The constructor `BSide` is for open or closed bounds, `BInfty` is for infinite bounds. How the boolean parameter distinguishes between open and closed bounds is better explained with illustrations. For example, the left bounds of the intervals `` `[x, +oo[ `` and `` `]x, +oo[ `` are respectively `BSide true x` and `BSide false x`, while the right bound of the interval `` `]-oo, x[ `` is `BSide true x`. This type allows for

the statements of generic lemmas about intervals, when they happen to hold independently of whether a bound is open or closed.

Let us define a type alias for `R` of type `realType` and the following set of open-closed intervals:

```
Definition ocitv_type : Type := R.
Definition ocitv := [set `]x.1, x.2]%classic | x in [set: R * R]].
```

This set forms a semiring of sets. Indeed, it contains `set0`, it is closed under finite intersection, and it satisfies the `semi_setD_closed` predicate from Sect. 3.2.1 (proofs `ocitv{0,I,D}` below):

```
Definition ocitv_display : Type → measure_display. Proof. exact. Qed.
HB.instance Definition _ := Pointed.on ocitv_type.
HB.instance Definition _ :=
  @isSemiRingOfSets.Build (ocitv_display R) ocitv_type
    ocitv ocitv0 ocitvI ocitvD.
```

5.2 Construction of the Lebesgue Measure

The length of an interval is defined by subtracting its left bound from its right bound. For the sake of generality, this is formally defined over arbitrary sets for which we take the hull using `Rhull` (see [42, file `normedtype.v`] for the definition of `Rhull`):

```
Definition hlength {R : realType} (A : set R) : \bar R :=
  let i := Rhull A in i.2 - i.1.
```

Now, the function `hlength` is a content on the semiring of sets of intervals. Indeed, it is non-negative (proof `hlength_ge0` below), and, more importantly, it is additive over `ocitv`:

```
Lemma hlength_semi_additive : semi_additive hlength.
Proof. (* see [42] *) Qed.
HB.instance Definition _ :=
  isContent.Build R _ hlength hlength_ge0 hlength_semi_additive.
```

Moreover, `hlength` is also *σ-subadditive* over `ocitv`, and hence a measure:

```
Lemma hlength_sigma_sub_additive : sigma_sub_additive hlength.
Proof. (* see [42] *) Qed.
HB.instance Definition _ := Content_SubSigmaAdditive_isMeasure.Build
  _ _ _ hlength hlength_sigma_sub_additive.
```

We obtain the Lebesgue measure as an application of the measure extension from Sect. 4.3. More precisely, we use the generic definition `measure_extension` to define the function corresponding to the Lebesgue measure, and it is directly a measure from Sect. 4.3.

```
Definition lebesgue_measure := measure_extension hlength.
HB.instance Definition _ := Measure.on lebesgue_measure.
```

The above construction provides a unique measure that applies to a *σ-algebra* generated from open-closed intervals (remember the use of `salgebraType` in Sect. 4.3), which include the Borel sets: this is the definition of the Lebesgue measure.

We have not introduced an explicit definition for Borel sets but their *σ-algebra* can be denoted by `ocitv.-sigma.-measurable` which is a notation that combines the definition `ocitv` of the set of open-closed intervals and the notation `.-sigma.-measurable` that we introduced in Sect. 3.4. This *σ-algebra* can easily be shown to be the same as the one generated by open intervals:

```
Module RGenOpens.
Section rgenopens.
Variable R : realType.
Definition G := [set A | ∃ x y, A = `]x, y[%classic].
Lemma measurableE :
  (@ocitv R).-sigma.-measurable = G.-sigma.-measurable.
Proof. (* see [42] *) Qed.
End RGenOpens.
```

Similarly, it can be shown to be the same as the one generated by open rays, etc. Furthermore, it can also be easily extended to a *σ-algebra* over extended real numbers. These facts (whose formal proofs can be found in [42, file `lebesgue_measure.v`]) are useful to establish the properties of measurable functions in the next section.

6 Construction of the Lebesgue Integral

We now show that the infrastructure we have developed for the Lebesgue measure can be used to develop the theory of the Lebesgue integral up to Fubini's theorem, which covers the typical set of properties that demonstrate the usefulness of such a formalization. This experiment improves in particular on related work in Coq by providing theorems for functions that are not necessary non-negative and that are extended-real valued, and also be experimenting with simpler encodings, in particular the one of simple functions. Hereafter, we shorten code snippets with the following convention: `T` has type `measurableType d` for some display parameter `d`, `R` has type `realType`, and `mu` is a measure of type `{measure set T → \bar R}`.

6.1 Mesurable Functions

Ultimately, the Lebesgue integral is about *measurable functions*. A function is measurable when any preimage of a measurable set is measurable. We defined it for functions with domain `D` as follows:

```
Definition measurable_fun d d' (T : measurableType d)
    (U : measurableType d') (D : set T) (f : T → U) :=
  measurable D → ∀ Y, measurable Y → measurable (D ∩ f @^-1` Y).
```

Note that when in the above definition `T` or `U` are actually `R` or `\bar R` with `R : realType`, a concrete instance of *σ-algebra* need to have been declared beforehand as explained in Sect. 5.2.

6.2 Simple Functions

The construction of the Lebesgue integral starts with simple functions. A *simple function* f is typically defined by a sequence of pairwise-disjoint and measurable sets $A_0, \ldots, A_{n-1}$ and a sequence of elements $a_0, \ldots, a_{n-1}$ such that $f(x) = \sum_{k=0}^{n-1} a_k \mathbf{1}_{A_k}(x)$. It might be tempting (in particular for a computer scientist) to encode this definition using lists to represent the range of simple functions. This actually turns out to be detrimental to formalization (see Sect. 7). Instead, we strive for modularity by obtaining simple functions from even more basic functions. For that purpose, we again put HIERARCHY-BUILDER to good use. We first define functions with a finite image (notation `{fimfun T >-> R}`):

```
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}.
```

We then package measurable functions (notation `{mfun T >-> R}`):

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

As a consequence, simple functions (notation `{sfun T >-> R}`) can be defined by combining both functions with a finite image and measurable functions:

```
HB.structure Definition SimpleFun
    d (aT : measurableType d) (rT : realType) :=
  {f of @isMeasurableFun d aT rT f & @FiniteImage aT rT f}.
```

Similarly, we introduce non-negative functions (notation `{nnfun T >-> R}`) and define non-negative simple functions (notation `{nnsfun T >-> R}`) resulting in the hierarchy displayed in Fig. 3.

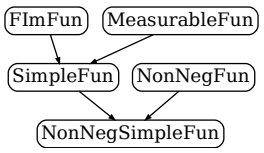

Figure 3: Definition of non-negative simple functions

The introduction for the above collection of types is a fertile ground for the formalization of the properties of simple functions. We show in particular that simple functions form a ring structure (a `comRingType` in MATHCOMP's parlance)

and thus that they can be combined accordingly (see `Section comring` in [42, file `lebesgue_integral.v`]).

Among all the simple functions, indicator functions `indic A` (notation `\1_A`, where `A` is a set) are of particular interest because they are used pervasively in the theory of integration:

```
Definition indic {T} {R : ringType} (A : set T) (x : T) : R :=
  (x \in A)%:R.
```

(`%:R` embeds boolean numbers and natural numbers into a ring.) In particular, any function with a finite image (and thus any simple function) is a linear combination of indicator functions:

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

This fact is instrumental in proofs using the monotone convergence theorem, such as Fubini's theorem (Sect. 6.5).

6.3 The Integral of Simple Functions

The integral of a simple function is the sum of its images multiplied by the measure of the associated preimage. In textbooks, the corresponding formula can be written in two ways. One can make explicit the finite image of the simple function and sum w.r.t. the indices, i.e., as $\sum_{k=0}^{n-1} a_k \mu(A_k)$ using the notations from the previous section and some measure μ. Since the image of a simple function is finite, one can alternatively use sums over finite supports (Sect. 2.4) and write: $\sum_{x \in \mathbb{R}} x\, \mu(f^{-1}\{x\})$. From the viewpoint of formalization, the former reveals implementation details while the latter is more compact and allows for the following simple definition of the integral of simple functions:

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

See Fig. 1 for `numDomainType`. The development of the properties of the integral of simple functions goes on by establishing the properties of the integral of non-negative simple functions such as semi-linearity, monotonicity, etc. Among them, the fact that the integral of the sum of simple functions is the sum of the integrals is the most technical result. Yet, it can be proved within 23 lines of script using generic properties of sums over finite supports (see `sintegralD`[4] [42, file `lebesgue_integral.v`]).

6.4 Integral of Measurable Functions

The integral of a measurable function is defined as the difference between its non-negative part and its non-positive part, both considered as non-negative

[4] `https://github.com/math-comp/analysis/blob/7d4ed9cf0e32f6be5b50c092cc8d93a21ec4dee3/theories/lebesgue_integral.v#L668`

functions. We therefore first temporarily define the integral of a non-negative measurable function, as the supremum of the integrals of smaller non-negative simple functions:

```
Let nnintegral f := ereal_sup [set sintegral mu h |
  h in [set h : {nnsfun T ↦ R} | ∀ x, (h x)%:E ≤ f x]].
```

Regarding the definition of the integral of a measurable function, we make the design choice to have it parameterized with the domain of integration. For that purpose, we introduce the notation `f \_ D` for the function that behaves as `f` over the set `D` and 0 elsewhere. The definition of the integral follows (notation `\int[mu]_(x in D) f x`):

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

In the code just above, the notation `f ^+` is for $\lambda x.\, \max(\mathtt{f}(x), 0)$ and the notation `f ^-` is for $\lambda x.\, \max(-\mathtt{f}(x), 0)$.

See [42, file `lebesgue_measure.v`] for the development of the theory of integration as presented in [32], and the next section for two illustrative examples.

6.5 Dominated Convergence and Fubini's Theorem

The dominated convergence theorem establishes the convergence of a sequence of integrals of functions f_n given an hypothesis of pointwise convergence of the functions f_n and an hypothesis of domination by an integrable function; these two hypotheses are true "almost everywhere". The standard presentation (e.g., [32, Sect. IV.2]) is to first prove the theorem when the hypotheses are unconditionally true, in which case the proof is essentially a consequence of Fatou's lemma and of the linearity properties of the integral. As for the generalization to hypotheses that are true "almost everywhere", it is almost always only sketched in textbooks. The complete statement of the dominated convergence theorem follows. The notation `{ae mu, ∀ x, P x}` means that `P` holds almost everywhere for the measure `mu`, i.e., that the complement of the set defined by `P` is negligible as defined in Sect. 4.2.

```
Variables (D : set T) (mD : measurable D).
Variables (f_ : (T → \bar R)^nat) (f g : T → \bar R).
Hypothesis mf_ : ∀ n, measurable_fun D (f_ n).
Hypothesis mf : measurable_fun D f.
Hypothesis f_f : {ae mu, ∀ x, D x → f_ ^~ x ⇝ f x}.
Hypothesis ig : mu.-integrable D g.
Hypothesis f_g : {ae mu, ∀ x n, D x → `|f_ n x| ≤ g x}.
Let g_ n x := `|f_ n x - f x|.

Theorem dominated_convergence : [∧ mu.-integrable D f,
  [sequence \int[mu]_(x in D) (g_ n x)]_n ⇝ 0 &
  [sequence \int[mu]_(x in D) (f_ n x)]_n ⇝ \int[mu]_(x in D) (f x) ].
```

Note that in this version of the dominated convergence theorem we assume that `f` is measurable; this hypothesis is not needed when `mu` is complete.

Fubini's theorem is a commutation result about integration. It is a good testbed for a combined formalization of measure and integration theory because, on the one hand, it requires the construction of the *product measure*, and, on the other hand, its proof relies on several lemmas about integration. Given two measures `m1` and `m2` respectively over two measurable types `T1` and `T2`, `m2` being σ-*finite*, the product measure `m1 \x m2` is defined as `\int[m1]_x (m2 \o xsection A) x` where `xsection A x` is the set of pairs `(x, y)` in `A`. In virtue of the uniqueness of measures (Sect. 4.3), inverting the role of `m1` and `m2` actually gives rise to the *same* measure. For the proof of Fubini's theorem, we follow the presentation by Li [32, Sect. V.3], which is standard. The first step is to prove Fubini-Tonelli's theorem, which is essentially Fubini's theorem for non-negative functions. The decomposition of functions with a finite image into a linear combination of indicator functions (Sect. 6.2) comes in handy to prove Fubini-Tonelli's theorem because the latter is first established for indicator functions, then for simple functions, and finally for measurable functions. The second main ingredient is the monotone convergence theorem [42, file `lebesgue_integral.v`]. Fubini's theorem is then essentially an application of Fubini-Tonelli's theorem:

```
Context d2 d2 (T1 : measurableType d1) (T2 : measurableType d2)
  (R : realType).
Variables (m1 : {sigma_finite_measure set T1 → \bar R})
          (m2 : {sigma_finite_measure set T2 → \bar R}).
Variable f : T1 * T2 → \bar R.
Hypothesis imf : (m1 \x m2).-integrable setT f.

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

7 Related Work

About Measure and Integration Theory in Proof Assistants based on Dependent Type Theory We are not aware of any formalization of the measure extension theorem *for general semirings of sets* in a proof assistant based on dependent type theory (neither Coq nor Lean).

There is a formalization in Coq, based on the Coquelicot library, of the Lebesgue integral of non-negative functions [14]. This development is driven by detailed pencil-and-paper proofs written for the purpose of formalization [16]. The theory of Lebesgue integration has been limited to non-negative functions and stops at Tonelli's theorem [13] but it has recently been extended with a formalization of the Bochner integral [15]. The authors have communicated to us that there is work in progress on the Lebesgue measure but that it is not a modular construction like ours.

The difference between the work by Boldo et al. and our work lies more in the sustaining infrastructure than in the gallery of theorems. First, we cannot reuse their framework because of many diverging choices of conventions, one of them is assuming that $\infty - \infty = 0$, which results in the addition of the extended

real numbers being non-associative, which prevents the use of iterated operators *à la* MathComp [14, Sect. 3.2]. We also insist on developing abstractions and components developed along MathComp-Analysis so as to find the best encodings. For example, Boldo et al. use a very concrete encoding of simple functions whose ranges are represented by sorted lists [14, Sect. 6.3]. Notwithstanding the fact that sorting is not essential to develop integration theory, it appears that this makes for longer proofs. For example, we already discussed the benefits of the infrastructure of iterated operators over finite supports (Sect. 2.4) regarding the proof that the integral of the sum of simple functions is the sum of the integrals (Sect. 6.3). The approach by Boldo et al. seems to make for a five times longer script (118 vs. 23 lines of codes, see `LInt_SFp_plus`[5] [30, file `simple_fun.v`]). Another example having a `sigma_algebra` predicate or a `measurableType` structure while Boldo et al. use the fact that a class of sets is a *σ-algebra* if and only if it is equal to the smallest *σ-algebra* generated by its elements. We found this characterization impractical in the presence of the hierarchy of classes of sets, for which we need inheritance to work in order to share theorem across structures. With an inductive characterization, theorems defined on a larger class of sets (e.g., semiring of sets) could not be applied to a *σ-algebra*. On a related note, our definition of generated *σ-algebra* in Sect. 3.3 generalizes the one by Boldo et al. by defining the complement with respect to an arbitrary set instead of the full set. This is very useful in practice to develop the theory of measurable partial functions and in fine define the Lebesgue integral as parameterized by a domain (Sect. 6.4).

The C-Corn library also deals with the formalization of integration in Coq as it has a formalization of the fundamental theorem of calculus [21] but this is about the Riemann integral and it is in a constructive setting.

The coq-proba library [38] provides a formalization of the Lebesgue measure and integral but limited to real-valued functions and closed intervals.

Lean has an extensive formalization of measure and integration theory. The main source of documentation is the code of mathlib [43]. To our understanding, measures are defined as a special case of outer measures [45], following the idea than any non-negative function can generate an outer measure which in turn can generate the *σ-algebra* of its Carathéodory measurable sets. Hence mathlib does not have a hierarchy of classes of sets reflecting the literature, as we did (Sect. 3.2), even though we believe that they naturally occur inside the proofs. mathlib provides the Lebesgue integral and its standard lemmas up to Fubini's theorem and the Radon-Nikodým theorem (which we have also recently proved using our framework [28]), and is actually further generalized to the Bochner integral. It has also supported the formalization of the Haar measure [45], which generalizes the Lebesgue measure.

We are not aware of a formalization of the Lebesgue measure or integral in Nuprl [20] or Agda [40] which are also proof assistants based on dependent type theory.

[5] `https://depot.lipn.univ-paris13.fr/mayero/coq-num-analysis/-/blob/d76dc70b06f70e2f1e99fd2ba3b22bba6ea78c91/Lebesgue/simple_fun.v#L809`

About Measure and Integration Theory in Proof Assistants of the HOL Family The HOL family of proof assistants has several formalizations of measure and integration theory. It can be traced back to a formalization of measure theory in HOL in 2002 [27, Sect. 2.2.2] (work actually inspired by earlier work in Mizar [11]). It was generalized in HOL 4 2010 [17, Sect. 2.3] and used to formalize Lebesgue integration [17, 35]. Work in HOL4 triggered a port in Isabelle/HOL that was eventually reworked in 2011 [26, Sect. 4.2]. The Lebesgue measure is defined in Isabelle/HOL using the gauge integral that was already available in Isabelle/HOL, i.e., it is not built as an extension of a premeasure [26, Sect. 4.6]. This approach results from a port from HOL-Light [25].

Measure and Integration Theory in Other Proof Assistants Proof assistants we have discussed so far are based on the LCF approach which consists in having a small kernel to ensure the soundness of proof checking. Other proof assistants based on an augmented trusted base providing more automation have also been used to formalize measure and integration theory.

The Mizar Mathematical Library (MML) [9] provides a formalization of measure theory that can be traced back to 1992 [11]. The Lebesgue measure in MML has recently been reconstructed [22] using an approach by extension from a semialgebra of intervals to fix an earlier formalization [12]. This is of course in a very different setting compared to our work in Coq since the Mizar proof assistant relies on the Tarski–Grothendieck set theory instead of dependent type theory.

NASALib [44] also provides a construction of the Lebesgue measure by extension but where extension is carried out from an algebra of sets [44, file `hahn_kolmogorov.pvs`] instead of a semiring of sets as we do. NASALib is written in PVS [36], an interactive and automated prover based on higher-order logic that provides predicate subtyping and dependent types [37]. The formalization of measure theory and Lebesgue integration has been initiated in 2007 [31].

To the best of our understanding, in Metamath [34], the Lebesgue measure is not defined by extension[6].

We did not find a formalization of the Lebesgue measure or integration in other mainstream theorem provers such as ACL2 or ProofPower, which seems to be confirmed by the "Formalizing 100 Theorems" list [46].

8 Conclusion

This paper introduced a Coq formalization of measure theory and Lebesgue integration that is compatible with MathComp and that extends MathComp-Analysis. This includes an original formalization of mathematical structures for measure theory (Sect. 3.2), an original formalization of the construction of measures using the Measure Extension theorem (Sect. 4.3), whose application to a measure over a semiring of intervals yields the Lebesgue measure (Sect. 5).

[6] `https://us.metamath.org/mpeuni/df-vol.html`

This also allows for the construction of the Lebesgue integral and the formalization of its theory up to Fubini's theorem (Sect. 6).

We argued about technical aspects of this formalization that we believe improve on related work (Sect. 7). At the beginning of this experiment, much work was dedicated to the formalization of structures for measure theory and to enrich the foundations (in particular, extended real numbers). Our development now provides new reusable libraries of general interest, in particular for extended real numbers and their sequences (Sect. 2), sums over finite supports (Sect. 2.4) and over general sets (Sect. 2.5). As concrete applications that illustrate the reusability of our formalization, we can mention the Lebesgue-Stieltjes measure, which could be formalized using the same approach we used for the Lebesgue measure in Sect. 5 [7], more standard results about measure theory such as the Radon-Nikodým theorem [8], and the formalization of the semantics of a probabilistic programming language [4].

Current and Future Work The Coq community now has several formalizations of integration, that rely on different grounds. We have been exchanging with the members of the MILC project [30] to look for ways to share the development effort. As a next step of our formalization, we plan to formalize the fundamental theorem of calculus for the Lebesgue integral to connect with the theory of derivatives of MathComp-Analysis. We have also started formalizing probability theory and in particular discrete random variables to generalize existing work on the formalization of discrete probabilities on top of MathComp (e.g., [6]) and to apply it to the formalization of equational reasoning for probabilistic programming languages (e.g., to extend [5] to continuous probabilities).

References

[1] C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Hritcu, K. Maillard, and B. Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *34th IEEE Computer Security Foundations Symposium (CSF 2021), Dubrovnik, Croatia, June 21–25, 2021*, pages 1–15. IEEE, 2021.

[2] R. Affeldt, C. Cohen, M. Kerjean, A. Mahboubi, D. Rouhling, and K. Sakaguchi. Competing inheritance paths in dependent type theory: a case study in functional analysis. In *In 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June 29–July 6*, volume 12167(2) of *Lecture Notes in Artifical Intelligence*, pages 3–20. Springer, Jul 2020.

[3] R. Affeldt, C. Cohen, and D. Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.*, 11(1):43–76, 2018.

[4] R. Affeldt, C. Cohen, and A. Saito. Semantics of probabilistic programs using s-finite kernels in coq. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023) Boston, MA, USA, January 16–17, 2023*, pages 3–16. ACM, 2023.

[5] R. Affeldt, J. Garrigue, D. Nowak, and T. Saikawa. A trustful monad for axiomatic reasoning with probability and nondeterminism. *J. Funct. Program.*, 31(E17), 2021.

[6] R. Affeldt, J. Garrigue, and T. Saikawa. Reasoning with conditional probabilities and joint distributions in Coq. *Comput. Softw.*, 37(3):79–95, 2020.

[7] R. Affeldt and Y. Ishiguro. Formalization of the Lebesgue-Stieltjes measure in MathComp-Analysis. `https://github.com/math-comp/analysis/pull/677`, 2023. Pull request to [42]. Completed in 2022.

[8] R. Affeldt and Y. Ishiguro. Formalization of the Radon-Nikodým theorem in MathComp-Analysis. `https://github.com/math-comp/analysis/pull/818`, 2023. Pull request to [42]. Completed in 2022.

[9] G. Bancerek, C. Bylinski, A. Grabowski, A. Kornilowicz, R. Matuszewski, A. Naumowicz, and K. Pak. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *J. Autom. Reason.*, 61(1-4):9–32, 2018.

[10] Y. Bertot, G. Gonthier, S. O. Biha, and I. Pasca. Canonical big operators. In *21st 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, 2008.

[11] J. Bialas. Properties of Caratheodor's measure. Technical report, 1992. Formalized Mathematics 4.

[12] J. Bialas. The one-dimensional Lebesgue measure. Technical report, 1995. Formalized Mathematics 7.

[13] S. Boldo, F. Clément, V. Martin, M. Mayero, and H. Mouhcine. A Coq formalization of Lebesgue induction principle and Tonelli's theorem. In *25th International Symposium on Formal Methods (FM 2023), Lübeck, Germany, March 6–10, 2023*, volume 14000 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2023.

[14] S. Boldo, F. Clément, F. Faissole, V. Martin, and M. Mayero. A Coq formalization of Lebesgue Integration of nonnegative functions. *J. Autom. Reason.*, 66(2):175–213, 2021.

[15] S. Boldo, F. Clément, and L. Leclerc. A Coq formalization of the Bochner integral, 2022. arXiv cs.LO 2201.03242.

[16] F. Clément and V. Martin. Lebesgue integration, detailed proofs to be formalized in Coq, 2021. arXiv cs.LO 2101.05678.

[17] A. R. Coble. *Anonymity, information, and machine-assisted proof.* PhD thesis, University of Cambridge, King's College, Jul 2010. TR UCAM-CL-TR-785.

[18] C. Cohen. *Formalized algebraic numbers: construction and first-order theory.* PhD thesis, École Doctorale de l'École Polytechnique, Laboratoire d'Informatique de l'École Polytechnique, Nov 2012.

[19] C. Cohen, K. Sakaguchi, and E. Tassi. 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, 2020.

[20] R. L. Constable, S. F. Allen, M. Bromley, R. Cleaveland, J. F. Cremer, R. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.

[21] L. Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In *Selected Papers of the Second International Workshop on Types for Proofs and Programs (TYPES 2002), Berg en Dal, The Netherlands, April 24–28, 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2002.

[22] N. Endou. Reconstruction of the one-dimensional Lebesgue measure. Technical report, National Institute of Technology, Gifu College, 2020. Formalized Mathematics 28(1):93–104.

[23] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. 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, 2009.

[24] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. 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, 2013.

[25] J. Harrison. The HOL light theory of euclidean space. *J. Autom. Reason.*, 50(2):173–190, 2013.

[26] J. Hölzl and A. Heller. Three chapters of measure theory in Isabelle/HOL. In *Second International Conference on Interactive Theorem Proving (ITP 2011), Berg en Dal, The Netherlands, August 22–25, 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2011.

[27] J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002. UCAM-CL-TR-566.

[28] Y. Ishiguro and R. Affeldt. A progress report on formalization of measure theory with MathComp-Analysis. In *25th Workshop on Programming and Programming Languages (PPL2023), Nagoya University, March 6–8, 2023*. Japan Society for Software Science and Technology, Mar 2023.

[29] A. Klenke. *Probability Theory: A Comprehensive Course*. Springer, 2014. 2nd edition.

[30] Le projet MILC. Numerical analysis in Coq. `https://depot.lipn.univ-paris13.fr/mayero/coq-num-analysis`, 2023. Since 2018. See also `https://lipn.univ-paris13.fr/MILC`.

[31] D. R. Lester. Topology in PVS: Continuous mathematics with applications. In *2nd Workshop on Automated Formal Methods (AFM 2007)*, pages 11–20. Association for Computing Machinery, 2007.

[32] D. Li. *Intégration et applications—Cours et exercices corrigés*. Eyrolles, 2016.

[33] Mathematical Components Team. Mathematical Components library. `https://github.com/math-comp/math-comp`, 2007. Last stable version: 2.0 (2023).

[34] N. Megill. *Metamath: A Computer Language for Mathematical Proofs*. 2019. Available at `https://us.metamath.org/downloads/metamath.pdf`. With extensive revisions by David A. Wheeler.

[35] T. Mhamdi, O. Hasan, and S. Tahar. On the formalization of the Lebesgue integration theory in HOL. In *First International Conference on Interactive Theorem Proving (ITP 2010), Edinburgh, UK, July 11–14, 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2010.

[36] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, NY, USA, June 15–18, 1992*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[37] J. M. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24(9):709–720, 1998.

[38] J. Tassarotti, J. Tristan, and K. Palmskog. coq-proba: A probability theory library for the Coq theorem prover. `https://github.com/jtassarotti/coq-proba`, 2023. Since 2019.

[39] J. Tassarotti, K. Vajjha, A. Banerjee, and J. Tristan. A formal proof of PAC learnability for decision stumps. In *10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021), Virtual Event, Denmark, January 17–19, 2021*, pages 5–17. ACM, 2021.

[40] The Agda Team. *Agda's documentation v2.6.3*, 2023. Available at `https://agda.readthedocs.io/en/v2.6.3`.

[41] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2023. Available at `https://coq.inria.fr/refman/`. Version 8.17.0.

[42] The MathComp-Analysis Team. MathComp-Analysis: Mathematical components compliant analysis library. `https://github.com/math-comp/analysis`, 2023. Since 2017. Last stable version: 0.6.2. This paper refers to the branch `hierarchy-builder`.

[43] The mathlib community. Lean mathematical components library. `https://github.com/leanprover-community/mathlib`, 2023. Since 2017.

[44] The NASALib development team. NASA PVS library of formal developments. Current version: 7.1.1. Available at `https://github.com/nasa/pvslib`., 2023.

[45] F. van Doorn. Formalized Haar measure. In *12th International Conference on Interactive Theorem Proving (ITP 2021) June 29–July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[46] F. Wiedijk. Formalizing 100 theorems. `http://www.cs.ru.nl/~freek/100`, 2023.