Formalizing functional analysis structures in dependent type theory

Reynald Affeldt[1], Cyril Cohen[2], Marie Kerjean[2], Assia Mahboubi[2], Damien Rouhling[2], and Kazuhiko Sakaguchi[3]

[1] National Institute of Advanced Industrial Science and Technology (AIST), Japan
[2] Inria, France
[3] University of Tsukuba, Japan

Abstract. This paper discusses the design of a hierarchy of structures which combine linear algebra with concepts related to limits, like topology and norms, in dependent type theory. This hierarchy is the backbone of a new library of formalized classical analysis, for the Coq proof assistant. It extends the Mathematical Components library, geared towards algebra, with topics in analysis. This marriage arouses issues of a more general nature, related to the inheritance of poorer structures from richer ones. We present and discuss a solution, coined forgetful inheritance, based on packed classes and unification hints.

Keywords: formalization of mathematics · dependent type theory · packed classes · Coq

1 Introduction

Mathematical structures are the backbone of the axiomatic method advocated by Bourbaki [8, 9] to spell out mathematically relevant abstractions and establish the corresponding vocabulary and notations. They are instrumental in making the mathematical literature more precise, concise, and intelligible. Modern libraries of formalized mathematics also rely on hierarchies of mathematical structures to achieve modularity, akin to interfaces in generic programming. By analogy, we call *instance* a witness of a mathematical structure on a given carrier. Mathematical structures, as interfaces, are essential to factor out the shared *vocabulary* attached to their instances. This vocabulary comes in the form of formal definitions and generic theorems, but also parsable and printable notations, and sometimes delimited automation. Some mathematical structures are *richer* than others in the sense that they entail them. Like in generic programming, rich structures *inherit* the vocabulary attached to poorer structures. Working out the precise meaning of symbols of this shared vocabulary is usually performed by enhanced type inference, which is implemented using type classes [5, 23, 24] or unification hints [4, 13, 16, 20]. In particular, these mechanisms must automatically identify inheritance relations between structures.

This paper discusses the design of a hierarchy of mathematical structures supporting a Coq [26] formal library for functional analysis, i.e., the study of

spaces of functions, and of structure-preserving transformations on them. The algebraic vocabulary of linear algebra is complemented with a suitable notion of "closeness" (e.g., topology, distance, norm), so as to formalize convergence, limits, size, etc. This hierarchy is based on the *packed classes* methodology [13, 16], which represents structures using dependent records. The library strives to provide notations and theories that are as generic as they would be on paper. It is an extension of the "Mathematical Components" library [17] (hereafter MathComp), which is geared towards algebra. This extension is inspired by the Coquelicot real analysis library [7], which has its own hierarchy.

Fusing these two hierarchies, respectively from the MathComp and Coquelicot libraries, happens to trigger several interesting issues, related to inheritance relations. Indeed, when several constructions compete to infer an instance of the poorer structure, the proof assistant displays indiscernible notations, or keywords, for constructions that are actually different. This issue is not at all specific to the formalization of functional analysis: actually, the literature reports examples of this exact problem but in different contexts, e.g., in Lean's mathlib [11,27]. It is however more likely to happen when organizing different flavors of mathematics in a same coherent corpus, as this favors the presence of problematic competing constructions. Up to our knowledge, the problem of competing inheritance paths in hierarchies of dependent records was never discussed per se, beyond some isolated reports of failure, and of ad hoc solutions. We thus present and discuss a general methodology to overcome this issue, coined *forgetful inheritance*, based on packed classes and unification hints.

The paper is organized as follows: in Sect. 2, we recall the packed classes methodology, using a running example. Sect. 3 provides two concrete examples of the typical issues raised by the presence of competing inheritance paths, before describing the general issue, drawing its solution, and comparing with other typeclass-like mechanisms. Finally, Sect. 4 describes the design of our hierarchy of structures for functional analysis, and its features, before Sect. 5 concludes.

2 Structures, inheritance, packed classes

We recall some background on the representation of mathematical structures in dependent type theory, and on the construction of hierarchies using packed classes. For that purpose, we use a toy running example (see the accompanying file `packed_classes.v` [1]), loosely based on the case study presented in Sect. 4.

2.1 Dependent records

In essence, a mathematical structure attaches to a carrier set some data (e.g., operators of the structure, collections of subsets of the carrier) and prescribed properties about these data, called the *axioms* of the structure. The Calculus of Inductive Constructions [12], as implemented, e.g., by Coq [26], Agda [25], or Lean [18], provides a *dependent record* construct, which allows to represent a given mathematical structure as a type, and its instances as terms of that

type. A dependent record is an inductive type with a single constructor, which generalizes dependent pairs to dependent tuples. The elements of such a tuple are the arguments of the single constructor. They form a *telescope* [10], i.e., collection of terms, whose types can depend on the previous items in the tuple.

For example, the `flatNormSpace` record type formalizes a structure with two operators, `fminus` and `fnorm`, and one axiom `fnormP`. Its single constructor is named `FlatNormSpace`, and it has four projections (also called fields) `carrier`, `fminus`, `fnorm`, and `fnormP`, onto the respective components of the tuple:

```
Structure flatNormSpace := FlatNormSpace {
  carrier : Type ;
  fminus : carrier → carrier → carrier;
  fnorm : carrier → nat;
  fnormP : ∀ x : carrier, fnorm (fminus x x) = 0 }.
```

Fields have a dependent type, parameterized by the one of the structure:

```
fminus : ∀ f : flatNormSpace, carrier f → carrier f → carrier f
fnormP : ∀ (f : flatNormSpace) (x : carrier f), fnorm f (fminus f x x) = 0.
```

In this case, declaring an instance of this structure amounts to defining a term of type `flatNormSpace`, which packages the corresponding instances of carrier, data, and proofs. For example, here is an instance on carrier `Z` (using the `Z.sub` and `Z.abs_nat` from the standard library resp. as the `fminus` and the `fnorm` operators):

```
Lemma Z_normP (n : Z) : Z.abs_nat (Z.sub n n) = 0. Proof. ... Qed.
Definition Z_flatNormSpace := FlatNormSpace Z Z.sub Z.abs_nat Z_normP.
```

2.2 Inference of mathematical structures

Hierarchies of mathematical structures are formalized by nesting dependent records but naive approaches quickly incur scalability issues. *Packed classes* [13, 16] provide a robust and systematic approach to the organization of structures into hierarchies. In this approach, a *structure* is a two-field record, which associates a carrier with a *class*. A class encodes the inheritance relations of the structure and packages various *mixins*. Mixins in turn provide the data, and their properties. In Coq, `Record` and `Structure` are synonym, but we reserve the latter for record types that represent actual structures. Let us explain the basics of inference of mathematical structures with packed classes by replacing the structure of Sect. 2.1 with two structures represented as packed classes. The first one provides just a binary operator:

```
1 Record isModule T := IsModule { minus_op : T → T → T }.
2 Structure module := Module {
3   module_carrier  : Type;
4   module_isModule : isModule module_carrier }.
```

Since the `module` structure is expected to be the bottom of the hierarchy, we are in the special class where the class is the same as the mixin (here, the class would be *equal* to `isModule`). To endow the operator `minus_op` with a generic

infix notation, we introduce a definition `minus`, parameterized by an instance of `module`. In the definition of the corresponding notation, the wildcard `_` is a placeholder for the instance of `module` to be inferred from the context.

```
Definition minus (M : module) :
    module_carrier M → module_carrier M → module_carrier M :=
  minus_op _ (module_isModule M).
Notation "x - y" := (minus _ x y).
```

We can build an instance of the `module` structure with the type of integers as the carrier and the subtraction of integers for the operator:

```
Definition Z_isModule : isModule Z := IsModule Z Z.sub.
Definition Z_module                := Module   Z Z_isModule.
```

But defining an instance is not enough to make the `_ - _` notation available:

```
Fail Check ∀ x y : Z, x - y = x - y.
```

To type-check the expression just above, Coq needs to fill the wildcard in the `_ - _` notation, which amounts to solving the equation `module_carrier ?M ≡ Z`, where `_ ≡ _` is the definitional equality. One can indicate that the instance `Z_module` is a *canonical* solution by declaring it as a *canonical instance*:

```
Canonical Z_module.
Check ∀ x y : Z, x - y = x - y.
```

This way, Coq fills the wildcard in `minus _ x y` with `Z_module` and retrieves as expected the subtraction for integers.

2.3 Inheritance and packed classes

We introduce a second structure class to illustrate how inheritance is implemented. This structure extends the `module` structure of Sect. 2.2 with a norm operator and a property (the fact that `x - x` is 0 for any `x`):

```
1  Record naiveNormMixin (T : module) := NaiveNormMixin {
2    naive_norm_op  : T → nat ;
3    naive_norm_opP : ∀ x : T, naive_norm_op (x - x) = 0 }.
4  Record isNaiveNormSpace (T : Type) := IsNaiveNormSpace {
5    nbase : isModule T ;
6    nmix  : naiveNormMixin (Module _ nbase) }.
7  Structure naiveNormSpace := NaiveNormSpace {
8    naive_norm_carrier          :> Type;
9    naive_normSpace_isNormSpace : isNaiveNormSpace naive_norm_carrier }.
10 Definition naive_norm (N : naiveNormSpace) :=
11   naive_norm_op _ (nmix _ (naive_normSpace_isNormSpace N)).
12 Notation "| x |" := (naive_norm _ x).
```

The new mixin for the norm appears at line 1 (it takes a `module` structure as parameter), the new class appears at line 4, and the structure[4] at line 7. It is the

[4] The notation `:>` in the structure declares the carrier as a *coercion*, which means that Coq has the possibility to use the function `naive_norm_carrier` to fix type-mismatches, transparently for the user.

class that defines the inheritance relation between `module` and `naiveNormSpace` (at line 6 precisely). The definitions above are however not enough to achieve proper inheritance. For example, `naiveNormSpace` does not yet enjoy the `_ - _` notation coming with the `module` structure:

```
Fail Check ∀ (N : naiveNormSpace) (x y : N), x - y = x - y.
```

Here, Coq tries to solve the following equation[5]:

```
module_carrier ?M ≡ naive_norm_carrier N
```

The solution consists in declaring a canonical way to build a `module` structure out of a `naiveNormSpace` structure in the form of a function that Coq can use to solve the equation above (using `naiveNorm_isModule N` in this particular case):

```
Canonical naiveNorm_isModule (N : naiveNormSpace) :=
  Module N (nbase _ (naive_normSpace_isNormSpace N)).
Check ∀ (N : naiveNormSpace) (x y : N), x - y = x - y.
```

3 Inheritance with packed classes and type classes

When several inheritance paths compete to establish that one structure entails the other, the proof assistant may display misleading information to the user and prevent proofs.

3.1 Competing inheritance paths

We extend the running example of Sect. 2. with a third and last structure. The `ballSpace` structure below provides a binary relation (line 2) together with a property of reflexivity (line 3):

```
1 Record isBallSpace T   := IsBallSpace {
2   ball_op  : T → T → Prop ;
3   ball_opP : ∀ x : T, ball_op x x }.
4 Structure ballSpace    := BallSpace {
5   ball_carrier          :> Type;
6   ballSpace_isBallSpace : isBallSpace ball_carrier }.
7 Definition ball {N : ballSpace} := ball_op _ (ballSpace_isBallSpace N).
8 Notation "x ∼∼ y" := (ball x y).
```

For the sake of the example, we furthermore declare a canonical way of building a `ballSpace` structure out of a `naiveNormSpace` structure:

```
Variable (N : naiveNormSpace).
Definition nnorm_ball (x : N) := fun y : N => |x - y| ≤ 1.
(* details about naiveNormSpace_isBallSpace omitted *)
Canonical nnorm_ballSpace := BallSpace N naiveNormSpace_isBallSpace
```

[5] The application of `naive_norm_carrier` is not necessary in our case thanks to the coercion explained in footnote 4.

We first illustrate the issue using a construction (here the Cartesian product) that preserves structures, and that is used to build canonical instances. First, we define the product of `module` structures, and tag it as canonical:

```
Variables (M M' : module).
Definition prod_minus (x y : M * M') := (fst x - fst y, snd x - snd y).
Definition prod_isModule             := IsModule (M * M') prod_minus.
Canonical  prod_Module               := Module   (M * M') prod_isModule.
```

Similarly, we define canonical products of `ballSpace` and `naiveNormSpace`:

```
Variables (B B' : ballSpace) (N N' : naiveNormSpace).
Definition prod_ball (x y : B * B') := fst x ∼∼ fst y ∧ snd x ∼∼ snd y.
(* definition of prod_isBallSpace omitted from the paper *)
Canonical prod_ballSpace := BallSpace (B * B') prod_isBallSpace.

Definition prod_nnorm (x : N * N') := max (|fst x|) (|snd x|).
(* definition of prod_isNNSpace omitted from the paper *)
Canonical prod_naiveNormSpace := NaiveNormSpace (N * N') prod_isNNSpace.
```

The problem is that our setting leads Coq's type-checker to fail in unexpected ways, as illustrated by the following example:

```
Variable P : ∀ {T}, (T → Prop) → Prop.
Example failure (Pball : ∀ V : naiveNormSpace, ∀ v : V, P (ball v))
  (W : naiveNormSpace) (w : W * W): P (ball w).
Proof. Fail apply Pball. Abort.
```

The hypothesis `Pball` applies to any goal `P (ball v)` where the type of `v` is of type `naiveNormSpace`, so that one may be led to think that it should also apply in the case of a product of `naiveNormSpace`s, since there is a canonical way to build one. What happens is that the type-checker is looking for an instance of a normed space that satisfies the following equation:

`nnorm_ballSpace ?N ≡ prod_ballSpace (nnorm_ballSpace W) (nnorm_ballSpace W)`

while the canonical instance Coq infers is `?N := prod_naiveNormSpace W W`, which does not satisfy the equation. In particular, `(ball_op x y)` is definitionally equal to `|x - y| ≤ 1` on the left-hand side and `(fst x ∼∼ fst y ∧ snd x ∼∼ snd y)` on the right-hand side: the two are not definitionally equal. One can describe the problem as the fact that the diagram in Fig. 1 *does not* commute definitionally.

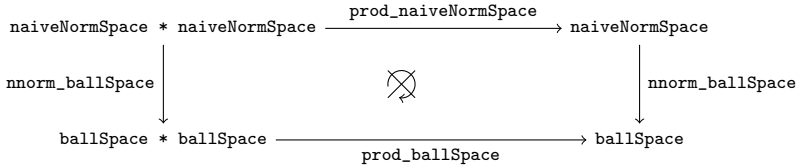

Fig. 1. Diagrammatic explanation for the failure of the first example of Sect. 3.1

This is of course not specific to Cartesian products and similar problems would also occur when lifting dependent products, free algebras, closure, completions, etc., on metric spaces, topological groups, etc. as well as in simpler settings without generic constructions as illustrated by our last example.

As a consequence of the definition of `nnorm_ballSpace`, the following lemma about balls is always true for any `naiveNormSpace`:

```
Lemma ball_is_nball (N : naiveNormSpace) (x y : N) : x ∼∼ y ↔ |x - y| ≤ 1.
Proof. reflexivity. Qed.
```

For the sake of the example, we define canonical instances of the `ballSpace` and `naiveNormSpace` structures with integers:

```
Definition Z_ball (m n : Z) := (m = n ∨ m = n + 1 ∨ m = n - 1)%Z.
(* definition of Z_isBallSpace omitted *)
Canonical Z_ballSpace := BallSpace Z Z_isBallSpace.

Definition Z_naiveNormMixin := NaiveNormMixin Z_module Z.abs_nat Z_normP.
Canonical Z_naiveNormSpace :=
  NaiveNormSpace Z (IsNaiveNormSpace Z_naiveNormMixin).
```

Since the generic lemma `ball_is_nball` holds, the user might expect to use it to prove a version specialized to integers. This is however not the case as the following script shows:

```
Example failure (x y : Z) : x ∼∼ y ↔ |x - y| ≤ 1.
rewrite -ball_is_nball. (* the goal is: x ∼∼ y ↔ x ∼∼ y *)
Fail reflexivity. (* !!! *)
```

The problem is that on the left-hand side Coq infers the instance `Z_ballSpace` with the `Z_ball` relation, while on the right-hand side it infers the instance `nnorm_ballSpace Z_naiveNormSpace` whose `ball x y` is definitionally equal to `|x - y| ≤ 1`, which is not definitionally equal to the newly defined `Z_ball`. In other words, the problem is the multiple ways to construct a "canonical instance" of `ballSpace` with carrier `Z`, as in Fig. 2.

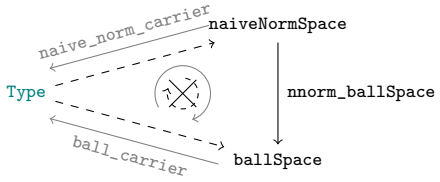

Fig. 2. Diagrammatic explanation for the type-checking failure of the second example of Sect. 3.1: the dashed arrows represent the inference of an instance from the carrier type; the outer diagrams commutes, while the inner one does not

The solution to the problems explained in this section is to ensure definitional equality by including poorer structures into richer ones; this way, "deducing" one structure from the other always amounts to erasure of data, and this guarantees there is a unique and canonical way of getting it. We call this technique *forgetful inheritance*, as it is reminiscent of forgetful functors in category theory.

3.2 Forgetful inheritance with packed classes

When applied to the first problem exposed in Sect. 3.1, forgetful inheritance makes the diagram of Fig. 1 commute *definitionally*. Indeed, the only way to

achieve commutation is to have `nnorm_ballSpace` be a mere erasure. This means that one needs to include inside each instance of `normSpace` a canonical `ballSpace` (line 7 below). Furthermore the `normMixin` must record the compatibility between the operators `ball_op` and `norm_op` (line 4 below):

```
1 Record normMixin (T : module) (m : isBallSpace T) := NormMixin {
2   norm_op       : T → nat;
3   norm_opP      : ∀ x,   norm_op (x - x) = 0;
4   norm_ball_opP : ∀ x y, ball_op _ m x y ↔ norm_op (x - y) ≤ 1 }.
5 Record isNormSpace (T : Type) := IsNormSpace {
6   base : isModule T;
7   bmix : isBallSpace T;
8   mix  : normMixin (Module _ base) bmix }.
9 Structure normSpace := NormSpace {
10  norm_carrier          :> Type;
11  normSpace_isNormSpace : isNormSpace norm_carrier }.
12 Definition norm (N : normSpace) :=
13  norm_op _ _ (mix _ (normSpace_isNormSpace N)).
```

Since every `normSpace` includes a canonical `ballSpace`, the construction of the canonical `ballSpace` given a `normSpace` is exactly a projection:

```
Canonical norm_ballSpace (N : normSpace) :=
  BallSpace N (bmix _ (normSpace_isNormSpace N)).
```

As a consequence, the equation

```
norm_ballSpace ?N ≡ prod_ballSpace (norm_ballSpace W) (norm_ballSpace W)
```

holds with `prod_normSpace W W` and the diagram in Fig. 1 (properly updated with the new `normSpace` structure) commutes definitionally, and so does the diagram in Fig. 2, for the same reasons.

Factories Because of the compatibility axioms required by forgetful inheritance, the formal definition of a structure can depart from the expected presentation. In fact, with forgetful inheritance, the very definition of a mathematical structure should be read in *factories*, i.e., functions that construct the mixins starting from only the expected axioms. And `Structure` records are rather interfaces, in a software engineering acceptation. Note that just like there can be several equivalent presentations of a same mathematical stuctures, several mixins can be associated with a same target `Structure`.

In our running example, one can actually derive, from the previously defined `naiveNormMixin`, two mixins for both `ballSpace`:

```
Variable (T : module) (m : naiveNormMixin T).
Definition fact_ball (x y : T) := naive_norm_op T m (x - y) ≤ 1.
Lemma fact_ballP (x : T) : fact_ball x x. Proof. (* omitted *). Qed.
Definition nNormMixin_isBallSpace := IsBallSpace T fact_ball fact_ballP.
```

(where the `ball` relation is the one induced by the norm, by construction) and `normSpace`:

```
(* details for fact_normP and fact_norm_ballP omitted from the paper *)
Definition nNormMixin_normMixin :=
  NormMixin T nNormMixin_isBallSpace (naive_norm_op T m)
            fact_normP fact_norm_ballP.
```

These two mixins make `naiveNormMixin` the source of two factories we mark as coercions, in order to help building two structures:

```
Coercion nNormMixin_isBallSpace : naiveNormMixin >→ isBallSpace.
Coercion nNormMixin_normMixin   : naiveNormMixin >→ normMixin.

Canonical alt_Z_ballSpace := BallSpace Z Z_naiveNormMixin.
Canonical alt_Z_normSpace := NormSpace Z (IsNormSpace Z_naiveNormMixin).
```

The second part of this paper provides concrete examples of factories for our hierarchy for functional analysis.

3.3 Forgetful inheritance with type classes

Type class mechanisms [5,23,24] propose an alternative implementation of hierarchies. Inference relations are coded using parameters rather than projections, and proof search happens by enhancing the resolution of implicit arguments. But the issue of competing inheritance paths does not pertain to the inference mechanism at stake, nor to the prover which implements them. Its essence rather lies in the non definitionally commutative diagrams as in Fig. 1 and Fig. 2.

We illustrate this with a type classes version of our examples, in both Coq and Lean, using a semi-bundled approach (see the accompanying files `type_classes.v` and `type_classes.lean` [1]). Compared to the packed class approach, hierarchies implemented using type classes remove the *structure* layer, which packages the carrier and the *class*. Hence our example keeps only the records whose name starts with `is`, declares them as type classes, and substitutes `Canonical` declarations with appropriate `Instance` declarations.

The choice on the level of bundling in the resulting classes, i.e., what are parameters of the record, and what are its fields, is not unique. For example, one may choose to formalize rings as groups extended with additional operations and axioms, or as a class on a type which is also a group.

```
Class isGroup T := IsGroup { ... };
Class isRing_choice1 T := IsRing { ring_isGroup : isGroup T; ... }.
Class isRing_choice2 T `{isGroup T} := IsRing { ... }.
```

By contrast, a structure in the packed class approach must always package with its carrier every mixins and classes that characterize the structure.

In the current implementations of type classes in both Coq and Lean, the choice on bundling may have dramatic consequences on resolution performances. Although tabling [22] may solve such performance issues by exploring the graph of dependencies in a cleverer way, it cannot make non definitionally commutative diagrams commute definitionally. The impact of forgetful inheritance on proof search using type classes is difficult to predict. But it has no performance cost with the packed class approach.

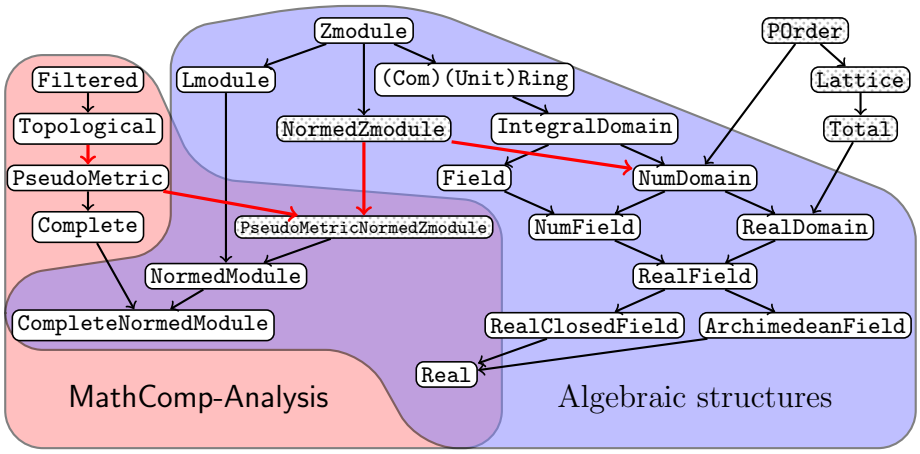

Fig. 3. Excerpt of MathComp and MathComp-Analysis hierarchies. Each rounded box corresponds to a mathematical structure except for `(Com)(Unit)Ring` that corresponds to several structures [13]. Dotted boxes correspond to mathematical structures introduced in Sect. 4.2 and Sect. 4.3. Thick, red arrows correspond to forgetful inheritance.

4 The Mathematical Components Analysis library

The Coquelicot library comes with its own hierarchy of mathematical structures and the intent of the MathComp-Analysis library is to improve it with the algebraic constructs of the MathComp library, for the analysis of multivariate functions for example. This section explains three applications of forgetful inheritance that solve three design issues of a different nature raised by merging MathComp and Coquelicot, as highlighted in Fig. 3.

We begin by an overview of the mathematical notions we deal with in this section. A *topological space* is a set endowed with a topology, i.e., a total collection of open sets stable by finite intersection and arbitrary unions. Equivalently, a topology can be described by the *neighborhood filter* of each point. A *neighborhood* of a point x is a set containing an open set around x; the neighborhood filter of a point x is the set of all neighborhoods of x. In MathComp-Analysis, neighborhood filters are the primary component of topological spaces. *Pseudo metric spaces* are intermediate between topological and metric spaces. They were introduced as the minimal setting to handle Cauchy sequences. In Coquelicot, pseudo metric spaces are called "uniform spaces" and are formalized as spaces endowed with a suitable `ball` predicate. This is the topic of Sect. 4.1. Coquelicot also provides *normed spaces*, i.e., $\mathbb{K}$-vector spaces E endowed with a suitable norm. On the other hand, in MathComp, the minimal structure with a norm operator corresponds to *numerical domains*, i.e., integral domains with order and absolute value. This situation led to a generalization of MathComp described in Sect. 4.2. Finally, in Sect. 4.3, we explain how to do forgetful inheritance across the two distinct libraries MathComp and MathComp-Analysis.

4.1 Forgetful inheritance from pseudo metric to topological spaces

When formalizing topology, we run into a problem akin to Sect. 3.1 because we face several competing notions of neighborhoods; we solve this issue with forgetful inheritance as explained in Sect. 3.

A neighborhood of a point p can be defined at the level of topological spaces using the notion of open as a set A that contains an open containing p:

$$\exists B.\ B \text{ is open, } p \in B \text{ and } B \subseteq A. \tag{1}$$

or at the level of pseudo metric spaces as a set A that contains a ball containing p:

$$\exists \varepsilon > 0.\ B_\varepsilon(p) \subseteq A. \tag{2}$$

We ensure these two definitions of neighborhoods coincide by adding to mixins compatibility axioms that constrain a shared function. The function in question maps a point to a set of neighborhoods (hereafter `locally`), it is shared between the mixins for topological and pseudo metric spaces, and constrained by the definitions of opens and balls as in Equations (1) and (2). More precisely, the mixin for topological spaces introduces the set of opens (see line 3 below) and defines neighborhoods as in Equation (1) (at line 5). We complete the definition by specifying with a specific axiom (not explained in detail here) that neighborhoods are proper filters (line 4) and with an alternative characterization of opens (namely that an open is a neighborhood of all of its points, line 6).

```
1 (* Module Topological. *)
2 Record mixin_of (T : Type) (locally : T → set (set T)) := Mixin {
3   open : set (set T) ;
4   ax1 : ∀ p : T, ProperFilter (locally p) ;
5   ax2 : ∀ p : T, locally p = [set A | ∃ B, open B ∧ B p ∧ B ⊆ A] ;
6   ax3 : open = [set A : set T | A ⊆ (fun x => locally x A) ] }.
```

The mixin for pseudo metric spaces introduces the notion of balls (line 10) and defines neighborhoods as in Equation (2) (at line 12, `locally_ ball` corresponds to the set of supersets of balls). The rest of the definition (line 11) are axioms about `ball` which are omitted for lack of space.

```
7 (* Module PseudoMetric. *)
8 Record mixin_of (R : numDomainType) (M : Type)
9     (locally : M → set (set M)) := Mixin {
10  ball : M → R → M → Prop ;
11  ax1 : ... ; ax2 : ... ; ax3 : ... ;
12  ax4 : locally = locally_ ball }.
```

Here, our definition of topological space departs from the standard definition as a space endowed with a family of subsets containing the full set and the empty set and stable by union and by finite intersection. However, the latter definition can be recovered from the former. Factories (see Sect. 3.2) are provided for users who want to give only `open` and to infer `locally` (using [3, file `topology.v`, definition `topologyOfOpenMixin`]), or the other way around.

4.2 Forgetful inheritance from numerical domain to normed Abelian group

The second problem we faced when developing the MathComp-Analysis library is the competing formal definitions of norms and absolute values. The setting is more complicated than Sect. 4.1 as it involves amending the hierarchy of mathematical structures of the MathComp library.

While the codomain of a norm is always the set of (non-negative) reals, an absolute value on a `numDomainType` is always an endofunction `norm` of type `∀ (R : numDomainType), R → R`. Thanks to this design choice, the absolute value preserves some information about its input, e.g., the integrality of an integer. On the other hand, the Coquelicot library also had several notions of norms: the absolute value of the real numbers (from the Coq standard library), the absolute value of a structure for rings equipped with an absolute value, and the norm operator of normed modules (the latter two are Coquelicot-specific).

We hence generalize the norm provided by the MathComp library to encompass both absolute values on numerical domains and norms on vector spaces, and share notation and lemmas. This is done by introducing a new structure in MathComp called `normedZmodType`, for normed Abelian groups, since Abelian groups are called $\mathbb{Z}$-modules in MathComp. This structure is now the poorest structure with a norm, which every subsequent normed type will inherit from.

The definition of the `normedZmodType` structure requires to solve a mutual dependency problem. Indeed, to state the fundamental properties of norms, such as the triangle inequality, the codomain of the norm function should be at least an ordered and normed Abelian group, requiring `normedZmodType` to be parameterized by such a structure. However the codomain should also inherit from `normedZmodType` to share the notations for norm and absolute value.

Our solution is to dispatch the order and the norm originally contained in `numDomainType` between normed Abelian groups `normedZmodType` and partially ordered types `porderType` as depicted in Fig. 3. We define the two following mixins for `normedZmodType` and `numDomainType`.

```
Record normed_mixin_of (R T : zmodType) (Rorder : lePOrderMixin R) :=
 NormedMixin { norm_op : T → R ; (* properties of the norm omitted *) }.

Record num_mixin_of (R : ringType) (Rorder : lePOrderMixin R)
  (normed : @normed_mixin_of R R Rorder) := Mixin { (* omitted *) }.
```

Now we define `numDomainType` (which is an abbreviation for `NumDomain.type`) using these two mixins but *without* declaring inheritance from `normedZmodType` (yet to be defined). More precisely, the class of `numDomainType` includes the mixin for `normedZmodType` (at line 5 below), which will allow for forgetful inheritance:

```
1 (* Module NumDomain. *)
2 Record class_of T := Class {
3   base : GRing.IntegralDomain.class_of T ;
4   order_mixin : lePOrderMixin (ring_for T base) ;
5   normed_mixin : normed_mixin_of (ring_for T base) order_mixin ;
6   num_mixin : num_mixin_of normed_mixin }.
```

```
7 Structure type := Pack { sort :> Type; class : class_of sort }.
```

Finally, we define the class of `normedZmodType`, parameterized by a `numDomainType`:

```
(* Module NormedZmodule. *)
Record class_of (R : numDomainType) (T : Type) := Class {
  base : GRing.Zmodule.class_of T ;
  normed_mixin : @normed_mixin_of R (@GRing.Zmodule.Pack T base)
          (NumDomain.class R) }.
```

It is only then that we declare inheritance from the `normedZmodType` structure to `numDomainType`, effectively implementing forgetful inheritance. We finally end up with a norm of the general type

```
norm : ∀ (R : numDomainType) (V : normedZmodType R), V → R.
```

Illustration: sharing of norm notation and lemmas As an example, we explain the construction of two norms and show how they share notation and lemmas. In MathComp, the type of matrices is `'M[K]_(m,n)` where `K` is the type of coefficients. The norm `mx_norm` takes the maximum of the absolute values of the coefficients:

```
Variables (K : numDomainType) (m n : nat).
Definition mx_norm (x : 'M[K]_(m, n)) : K := \big[maxr/0]_i `|x i.1 i.2|.
```

This definition uses the generic big operator [6] to define a "big max" operation out of the binary operation `maxr`. Similarly, we define a norm for pairs of elements by taking the maximum of the absolute value of the two projections[6]:

```
Variables (R : numDomainType) (U V : normedZmodType R).
Definition pair_norm (x : U * V) : R := maxr `|x.1| `|x.2|.
```

We then go on proving that these definitions satisfy the properties of the norm and declare canonical instances of `normedZmodType` for matrices and pairs (see [3] for details). All this setting is of course carried out in advance and the user only sees one notation and one set of lemmas (for example `ler_norm_add` for the triangle inequality), so that (s)he can mix various norms transparently in the same development, as in the following two examples:

```
Variable (K : numDomainType).
Example mx_triangle m n (M N : 'M[K]_(m, n)) : `|M + N| ≤ `|M| + `|N|.
Proof. apply ler_norm_add. Qed.
Example pair_triangle (x y : K * K) : `|x + y| ≤ `|x| + `|y|.
Proof. apply ler_norm_add. Qed.
```

One could fear that the newly introduced structures make the library harder to use since that, to declare a canonical `numDomainType` instance, a user also needs now to declare canonical `porderType` and `normedZmodType` instances of the same type. Here, the idea of factory (Sect. 3.2) comes in handy for the original `numDomainType` mixin has been re-designed as a factory producing `porderType`, `normedZmodType`, and `numDomainType` mixins in order to facilitate their declaration.

[6] The actual code of `mx_norm` and `pair_norm` is slightly more complicated because it uses a type for non-negative numeric values, see [3, file `normedtype.v`].

4.3 Forgetful inheritance from normed modules to pseudo metric spaces

The combination of the MathComp library with topological structures ultimately materializes as a mathematical structure for *normed modules*. It is made possible by introducing an intermediate structure that combines norm (from algebra) with pseudo metric (from topology) into normed Abelian groups. The precise justification for this first step is as follows.

Since normed Abelian groups have topological and pseudo metric space structures induced by the norm, `NormedZmodType` should inherit from `PseudoMetricType`. To do so, we can (1) insert a new structure above `NormedZmodType`, or (2) create a common extension of `PseudoMetricType` and `NormedZmodType`. We choose (2) to avoid amending the MathComp library. This makes both `NormedZmodType` and its extension `PseudoMetricNormedZmodType` normed Abelian groups, where the former is inadequate for topological purposes. The only axiom of this extension is the compatibility between the pseudo metric and the norm, as expressed line 5 below, where `PseudoMetric.ball` has been seen in Sect. 4.1 and the right-hand side represents all the ternary relations $\lambda x, \varepsilon, y.\, |x - y| < \varepsilon$:

```
1 (* Module PseudoMetricNormedZmodule. *)
2 Variable R : numDomainType.
3 Record mixin_of (T : normedZmodType R) (locally : T → set (set T))
4     (m : PseudoMetric.mixin_of R locally) :=
5   Mixin { _ : PseudoMetric.ball m = ball_ (fun x => `|x|) }.
```

The extension is effectively performed by using this mixin in the following class definition at line 12 (see also Fig. 3):

```
6 Record class_of (T : Type) := Class {
7   base : Num.NormedZmodule.class_of R T ; ... ;
8   locally_mixin : Filtered.locally_of T T ;
9   topological_mixin : @Topological.mixin_of T locally_mixin ;
10  pseudometric_mixin : @PseudoMetric.mixin_of R T locally_mixin ;
11  mixin :
12    @mixin_of (Num.NormedZmodule.Pack _ base) _ pseudometric_mixin }.
```

Finally, the bridge between algebraic and topological structures is completed by a common extension of a normed Abelian group (`PseudoMetricNormedZmodType`) with a left-module (`lmodType` from the MathComp library, which provides scalar multiplication), extended with the axiom of linearity of the norm for the scalar product (line 5 below).

```
1 (* Module NormedModule. *)
2 Variable (K : numDomainType).
3 Record mixin_of
4     (V : pseudoMetricNormedZmodType K) (scale : K → V → V) :=
5   Mixin { _ : ∀ (l : K) (x : V), `|scale l x| = `|l| * `|x| }.
```

One can again observe here the overloaded notation for norms explained in Sect. 4.2. The accompanying file `scalar_notations.v` [1] provides an overview of MathComp-Analysis operations regarding norms and scalar notations.

We ensured that the structure of normed modules indeed serves its intended purpose of enabling multivariate functional analysis by generalizing existing theories of Bachmann-Landau notations and of differentiation [2, Sect. 4].

5 Conclusion and related work

This paper has two main contributions: forgetful inheritance using packed classes, and the hierarchy of the MathComp-Analysis library. The latter library is still in its infancy and covers far less real and complex analysis than the libraries available in HOL Light and Isabelle/HOL [15,19]. However, differences in foundations matter here, and the use of dependent types in type-class-like mechanisms is instrumental in the genericity of notations illustrated in this paper. Up to our knowledge, no other existing formal library in analysis has comparable sharing features.

The methodology presented in this paper to tame competing inheritance paths in hierarchies of dependent records is actually not new. The original description of packed classes [13, end of Sect. 3.1] already mentions that a choice operator (in fact, a mixin) should be included in the definition of a structure for countable types, even if choice operators can be defined for countable types in Coq without axiom. Yet, although the MathComp library uses forgetful inheritance at several places in its hierarchy, this solution was never described in earlier publications, nor was the issue precisely described. Proper descriptions, as well as the comparison with other inference techniques, are contributions of the present paper. As explained in Sect. 3.3, type classes also allow for a variant of forgetful inheritance, although with a less robust proof search. This was also observed before, e.g., by Buzzard et al. [11, Sect. 3]. Interestingly, it seems that the authors have not identified that another issue they raise, about completing groups and rings [11, Sect. 5.3], pertains to the same problem, and can be solved using forgetful inheritance as well.

Packed classes, and forgetful inheritance, already proved robust and efficient enough to formalize and populate large hierarchies [14], where "large" applies both to the number of structures and to the number of instances. Arguably, this approach also has drawbacks: defining deep hierarchies becomes quite verbose, and inserting new structures is tedious and error-prone. We argue that, compared to their obvious benefits in control and efficiency of the proof search, this is not a fundamental issue. As packed classes are governed by systematic patterns and invariants, this rather calls for more inspection and generation tooling, which is work in progress [21].

Acknowledgments The authors are grateful to Georges Gonthier for the many fruitful discussions that helped rewriting parts of MathComp and MathComp-Analysis library. We also thank Arthur Charguéraud and anonymous reviewers for their comments on earlier versions of this work.

References

1. Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Sakaguchi, K.: Formalizing functional analysis structures in dependent type theory (accompanying material). `https://staff.aist.go.jp/reynald.affeldt/fipc/` (2020), contains the files `packed_classes.v`, `packed_classes.v`, `type_classes.lean`, and `scalar_notations.v`, and a stand-alone archive of the development version of the Mathematical Components library [17] with a snapshot of [3]
2. Affeldt, R., Cohen, C., Rouhling, D.: Formalization Techniques for Asymptotic Reasoning in Classical Analysis. Journal of Formalized Reasoning **11**, 43–76 (2018)
3. analysis Team: Mathematical components compliant analysis library. `https://github.com/math-comp/analysis` (2017), branch `analysis_270`. Last stable version 0.2.3 (2019) available on the same website
4. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: Hints in unification. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009. Lecture Notes in Computer Science, vol. 5674, pp. 84–98. Springer (2009)
5. Bauer, A., Gross, J., Lumsdaine, P.L., Shulman, M., Sozeau, M., Spitters, B.: The HoTT library: a formalization of homotopy type theory in Coq. In: 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017), Paris, France, January 16–17, 2017. pp. 164–172. ACM (2017)
6. Bertot, Y., Gonthier, G., Biha, S.O., Pasca, I.: Canonical big operators. In: 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008), Montreal, Canada, August 18–21, 2008. Lecture Notes in Computer Science, vol. 5170, pp. 86–101. Springer (2008)
7. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for Coq. Mathematics in Computer Science **9**(1), 41–62 (2015)
8. Bourbaki, N.: The architecture of mathematics. The American Mathematical Monthly **57**(4), 221–232 (1950), `http://www.jstor.org/stable/2305937`
9. Bourbaki, N.: Théorie des ensembles. Éléments de mathématique, Springer (2006), original Edition published by Hermann, Paris, 1970
10. de Bruijn, N.G.: Telescopic mappings in typed lambda calculus. Information and Computation **91**(2), 189–204 (1991)
11. Buzzard, K., Commelin, J., Massot, P.: Formalising perfectoid spaces. In: 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020), New Orleans, LA, USA, January 20–21, 2020. pp. 299–312. ACM (2020)
12. Coquand, T., Paulin, C.: Inductively defined types. In: International Conference on Computer Logic (COLOG-88), Tallinn, USSR, December 1988. Lecture Notes in Computer Science, vol. 417, pp. 50–66. Springer (1990)
13. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Theorem Proving in Higher-Order Logics (TPHOL 2009). Lecture Notes in Computer Science, vol. 5674, pp. 327–342. Springer (2009)
14. 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., Théry, L.: 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. Lecture Notes in Computer Science, vol. 7998, pp. 163–179. Springer (2013)
15. Harrison, J.: The HOL Light System REFERENCE (2017), available at `https://www.cl.cam.ac.uk/~jrh13/hol-light/index.html`.

16. Mahboubi, A., Tassi, E.: Canonical structures for the working Coq user. In: 4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, pp. 19–34. Springer (2013)
17. Mathematical Components Team: Mathematical Components library. `https://github.com/math-comp/math-comp` (2007), development version. Last stable version 1.10 (2019) available on the same website
18. Microsoft Reasearch: L∃∀N THEOREM PROVER. `https://leanprover.github.io` (2020)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag (2019), availble at `https://isabelle.in.tum.de/doc/tutorial.pdf`
20. Saïbi, A.: Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory). Ph.D. thesis, Pierre and Marie Curie University, Paris, France (1999)
21. Sakaguchi, K.: Validating mathematical structures. In: The Coq Workshop 2019, Portland, OR, USA, September 8, 2019 (2019), abstract. 2 pages
22. Selsam, D., Ullrich, S., de Moura, L.: Tabled typeclass resolution (2020), available at `https://arxiv.org/abs/2001.04301`
23. Sozeau, M., Oury, N.: First-Class Type Classes. In: 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008), Montréal, Québec, Canada, August 18–21, 2008. Lecture Notes in Computer Science, vol. 5170, pp. 278–293. Springer (2008)
24. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science **21**(4), 795–825 (2011)
25. The Agda Team: The Agda User Manual (2020), available at `https://agda.readthedocs.io/en/v2.6.0.1`. Version 2.6.0.1
26. The Coq Development Team: The Coq Proof Assistant Reference Manual. Inria (2019), available at `https://coq.inria.fr`. Version 8.10.2
27. The mathlib Community: The Lean mathematical library. In: 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020), New Orleans, LA, USA, January 20–21, 2020. pp. 367–381. ACM (2020)