Formalization Techniques for Asymptotic Reasoning in Classical Analysis

REYNALD AFFELDT
National Institute of Advanced Industrial Science and Technology, Japan
CYRIL COHEN and DAMIEN ROUHLING
Université Côte d'Azur, Inria, France

Formalizing analysis on a computer involves a lot of "epsilon-delta" reasoning, while informal reasoning may use some asymptotic hand-waving. Whether or not the arithmetic details are hidden using some abstraction like filters, a human user eventually has to break it down for the proof assistant anyway, and provide a witness for the existential variable "delta". We describe formalization techniques that take advantage of existential variables to delay the input of witnesses until a stage where the proof assistant can actually infer them. We use these techniques to prove theorems about classical analysis and to provide equational Bachmann-Landau notations. This partially restores the simplicity of informal hand-waving without compromising the proof. As expected this also reduces the size of proof scripts and the time to write them, and it also makes proofs more stable.

1. INTRODUCTION

In classical analysis, formalization problems occur when we have *local reasoning*, *i.e.* proof of facts that are only true in some neighborhood. One very early and trivial example when such reasoning occurs is to prove that the sum of two converging functions is converging. Indeed from

$$\begin{cases} \forall \varepsilon > 0.\ \exists \delta_f > 0.\ \forall x.\ |x - a| < \delta_f \Rightarrow |f(x) - l_f| < \varepsilon \\ \forall \varepsilon > 0.\ \exists \delta_g > 0.\ \forall x.\ |x - a| < \delta_g \Rightarrow |g(x) - l_g| < \varepsilon \end{cases},$$

$$\text{we get} \quad \forall \varepsilon > 0.\ \exists \delta > 0.\ \forall x.\ |x - a| < \delta \Rightarrow |f(x) + g(x) - (l_f + l_g)| < \varepsilon.$$

Formally proving this requires to show the existence of such a δ, here it may be the minimum of the two δ_f, δ_g we can get from the hypotheses applied to $\frac{\varepsilon}{2}$. Giving δ explicitly makes the proof less *stable* and less readable than it would be with a "correct" informal reasoning. By stable proof, we mean that changes in its statement, or in statements it depends on, will break only the parts of the proof where the changes actually matter. When we provide an existential witness way before using it, the distance between the place it is used (and breaks), and the place where it is introduced, makes it difficult to maintain the proof script. Indeed, the maintainer has to go back and forth in the proof script to understand how changing the existential leads to breakage.

Filters slightly improve stability and readability by hiding arithmetic reasoning, as successfully demonstrated by previous work in the Isabelle/HOL library [HIH13] and in the Coquelicot library [BLM15]. However, the explicit existential quantifiers are still replaced by forward reasoning with statements that depend on how the proof will be led. Our first contribution is to solve this problem by giving a set of tactics and lemmas to handle existential variables in a consistent way.

Another common tool in informal classical analysis is asymptotic developments using Bachmann-Landau notations, often called little-o and big-$\mathcal{O}$ notations [Bac94, Lan09]. They are used to write developments such as

$$f(x) = a_0 + a_1 x + \ldots + a_n x^n + \underset{x \to 0}{o}(x^n)$$

or in the definition of differential: it is the linear operator df_x such that

$$f(x+h) = f(x) + df_x(h) + o(h).$$

Using Bachmann-Landau notations, one performs arithmetic operations with developments and uses laws like $\underset{x \to 0}{o}(x^n) + \underset{x \to 0}{o}(x^n) = \underset{x \to 0}{o}(x^n)$. At first sight, the abuse of notation seems to make such a law impossible to represent as an equation on functions in a formal logic. Our second contribution is to provide a solution for this problem with a set of notations and lemmas which make the user believe that she is doing arithmetic with little-o and big-$\mathcal{O}$ at the same time. To our knowledge, this is the first formalization that mixes in a purely equational manner little-o and big-$\mathcal{O}$ functions with arithmetic operations. The formalizations of Bachmann-Landau notations [AD04, BCF$^+$13, BLM15, Ebe17, GCP18] we know about either handle sets of functions or do not mix little-o and big-$\mathcal{O}$.

We believe that the contributions discussed in this article are a key to make proofs in classical analysis easier in Coq [Coq18], both in the development of an analysis library and in its use. We extensively tested our tools in an on-going effort to provide Mathematical Components [G$^+$15] with analysis [ACM$^+$18a]. Both the development from this article and its numerous applications can be found in the latter repository.

Outline of this Article. In Sect. 2, we remind the reader of the concept of filter and explain how we extend the ideas from the Coquelicot library with a few structures and notations to make it look closer to mathematical practice. Then, in Sect. 3, we describe our methodology to make explicit existential quantifiers disappear from the proof flow; it can be seen as a special introduction rule. We provide a commented example of script that has been considerably shortened and made more stable using this methodology. In Sect. 4, we introduce our implementation of Bachmann-Landau notations. We discuss in particular in Sect. 4.1 how we represent laws using Bachmann-Landau notations as equations on functions in formal logic. We also give a few examples of informal reasoning that can actually be done as such with our techniques. Our formalization primarily targets classical analysis, which naturally calls for an extension of Coq's otherwise constructive logic. In Sect. 5, we explain the axioms on which our techniques rely, how we justify them, and why they are important to develop practical asymptotic reasoning in Coq. We discuss related work in Sect. 6 where we review in particular other formal proof libraries for Bachmann-Landau notations. Finally, we conclude in Sect. 7. We also provide in Appendix A a short overview of the contents of the Mathematical Components Analysis library [ACM$^+$18a], which is used both as a basis for our work and as a test base. We believe our work does not strongly depend on it, but

Standard Coq/Mathematical Components notations:

exists2 x, P x & Q x	existence of an x that satisfies both P and Q
.1, .2	first, second projection of a pair
@^~ x	application at x, *i.e.* fun f => f x
{linear U -> V}	linear functions of type U -> V

Definitions/notations from [CR17]:

set A	A -> Prop
A ‘<=‘ B	set inclusion
[set a | P a]	the set of elements a that satisfy P
F --> G	reverse set inclusion for filters $F \supseteq G$
f @ F	filter $f(F)$, see Sect. 2.3
+oo	$+\infty$

Fig. 1. Notations from previous work used in this article

a description might be useful for the reader to understand how our tools fit in the bigger picture and to help her find many examples of use.

About Notations. In this article, we introduce a number of notations that we explain as they appear (and summarize at the end of this article in Fig. 3). For the convenience of the reader, we summarize in Fig. 1 other standard Coq and Mathematical Components notations, as well as notations from previous work.

This article uses and extends SSReflect tactics. For explanations about the standard ones, the reader is referred to a tutorial introduction [GM10], as well as a precise documentation [GMT16] available online. As for the new tactics introduced in this article (namely, `near=>`, `near·=>`, `near:`, and `end_near`), they are explained in details in Sect. 3.2. They are designed as a conservative extension of SSReflect, except that they overload the tacticals `=>` and `:` for introduction and discharge of hypotheses.

Coq comes with a mechanism for implicit arguments: thanks to type constraints, some arguments in definitions can be inferred. We follow in this article Coq's syntax for implicit arguments. When giving a new definition, implicit arguments are declared using curly brackets, as in

```
Definition fct {arg1 : type1} (arg2 : type2) := ... .
```

Then, `arg1` is omitted in subsequent uses of `fct`, which all are of the form `fct arg2`.

2. ABSTRACTING ASYMPTOTIC STATEMENTS USING FILTERS

The use of filters in the Coquelicot library [BLM15] and the Isabelle/HOL library [HIH13] proved that they define a good abstraction for convergence proofs in analysis. We first recall in Sect. 2.1 the definition of filters and give a few examples. Then, we explain in Sect. 2.2 how our hierarchy of topological structures compares to the one of Coquelicot. We try to give just enough details to explain how to use our tools in practice. More details about these can be found in Appendix A. Finally, we detail in Sect. 2.3 the structures and notations we use in order to make the use of filters more natural in Coq.

2.1 Definition and Use of Filters

Let us first start with the definition of filters. A filter F on T is a set of sets of elements of T that satisfies the following three laws:

$$T \in F, \quad \forall A, B \in F.\ A \cap B \in F \quad \text{and} \quad \forall A, B.\ A \subseteq B \Rightarrow A \in F \Rightarrow B \in F.$$

Our Coq representation of filters is exactly the same as in the Coquelicot library, *i.e.* the propositional predicate `Filter F`, which states that the set of sets `F : set (set T)`, is a filter. Additionally, this predicate is a class, in order to trigger type-class inference when needed[1].

```
Class Filter {T : Type} (F : set (set T)) := {
  filterT : F setT ;
  filterI : forall P Q : set T, F P -> F Q -> F (P ‘&‘ Q) ;
  filterS : forall P Q : set T, P ‘<=‘ Q -> F P -> F Q
}.
```

We additionally provide the type `filter_on T` of filters on a type `T`, with an implicit coercion so that `F : filter_on T` may also be seen as a set of sets on `T`, satisfying the property `Filter F`.

```
Structure filter_on T := FilterType {
  filter :> (T -> Prop) -> Prop;
  filter_class : Filter filter
}.
```

This structure is used to register canonical filter instances in addition to type-class instances. Eventually, we might unify both inference mechanisms by choosing canonical structures over type-classes. The topic of structure inference is however beyond the scope of this article, and we recommend dedicated papers about it. [SO08, MT13]

The most important sort of filters used for analysis and local reasoning is the filter of neighborhoods. The set of neighborhoods of a point x indeed defines a filter, called $\text{locally}(x)$ in Coquelicot [BLM15] and in our work. In Coquelicot, the notion of neighborhood is defined using balls in a uniform space. Thus, the neighborhood filter of x is

$$\text{locally}(x) = \{A \mid \exists \varepsilon > 0.\ \text{ball}_\varepsilon(x) \subseteq A\}\,.$$

Balls can also be used to define another filter which is the set of *entourages*. An entourage is a set that is a "neighborhood" of the diagonal set $\{(x, x) \mid x \in T\}$, *i.e.* a set that contains all the pairs (x, y) such that $y \in \text{ball}_\varepsilon(x)$ for some positive ε.

An important point to notice here is the fact that the filter of entourages is defined as the set of supersets of the family of sets $(\{(x, y) \mid y \in \text{ball}_\varepsilon(x)\})_{\varepsilon>0}$.

In fact, we often use this kind of construction in proofs about filters. Hence, we define a function `filter_from` that takes a family of sets and returns its set of supersets.

[1] Unless stated otherwise, the code snippets displayed in this section can be found in [ACM+18a, file `topology.v`].

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

Here, D should be understood as the domain of indices and B defines the family. We also use notations for set comprehension and set inclusion that have been introduced in a previous work [CR17] (see Fig. 1). If the domain is not empty and if for any two indices i and j in the domain one can find a third index k in the domain such that $B_k \subseteq B_i \cap B_j$, then we say that the family defines a *filter base* and we prove that filter_from D B indeed defines a filter.

The entourage filter is then easily defined using filter_from and the family of sets described above[2].

```
Definition entourages {T : uniformType} : set (set (T * T)):=
  filter_from [set eps : R | eps > 0]
              (fun eps => [set xy | ball xy.1 eps xy.2]).
```

Since we are using balls, this definition is valid in a uniform space, denoted by uniformType in our work (note there is an implicit coercion from a uniformType to its carrier in Type). In fact, a more abstract definition of entourages, which does not rely on balls, could replace balls as primitive for the definition of the type representing uniform spaces. This would lead to an equivalent definition of uniform spaces where the pseudometric is abstracted, but we kept Coquelicot's definition for this work.

We can also use the filter_from function to define the *filter product*: if F and G are respectively filters on spaces T and U, then the filter product of F and G is a filter on the Cartesian product $T \times U$ and is defined as the set of supersets of the family $(P_1 \text{ `}*\text{` } P_2)_{P_1 \in F, P_2 \in G}$ where $A \text{ `}*\text{` } B = \{(a, b) \mid a \in A, b \in B\}$.

```
Definition filter_prod {T U : Type} (F : set (set T))
  (G : set (set U)) :=
  filter_from (fun P => F P.1 /\ G P.2) (fun P => P.1 `*` P.2).
```

This is a simplification of the filter product from the Coquelicot library, which is defined using an inductive predicate. This can easily be generalized to the n-ary filter product, allowing us in particular to build the neighborhood filter of a vector in $\mathbb{R}^n$ as the n-ary filter product of the neighborhood filters of its components.

A last construction which is of interest for analysis is the image of a filter by a function. Given a function f from T to U and a filter F on T, the image of F by f, defined by $f(F) = \left\{B \mid f^{-1}(B) \in F\right\}$, is a filter on U.

Except for filter_from, all the filters or constructions we introduced have or preserve the property of being a *proper filter*. Proper filters satisfy the extra law that they do not contain the empty set, which implies classically that any element of a proper filter is non-empty and that we can thus pick one element. The filter filter_from D B is proper if the family B does not contain the empty set, which is the case for instance in the definitions of entourages, and filter_prod F G is proper when F and G are proper filters. Most often we are interested only in proper filters, hence they are sometimes simply called "filters" (as in [GCP18]).

[2] The formalization of uniformType and the formal definition of the entourage filter can be found in [ACM+18a, file hierarchy.v] and in Appendix A.2.4

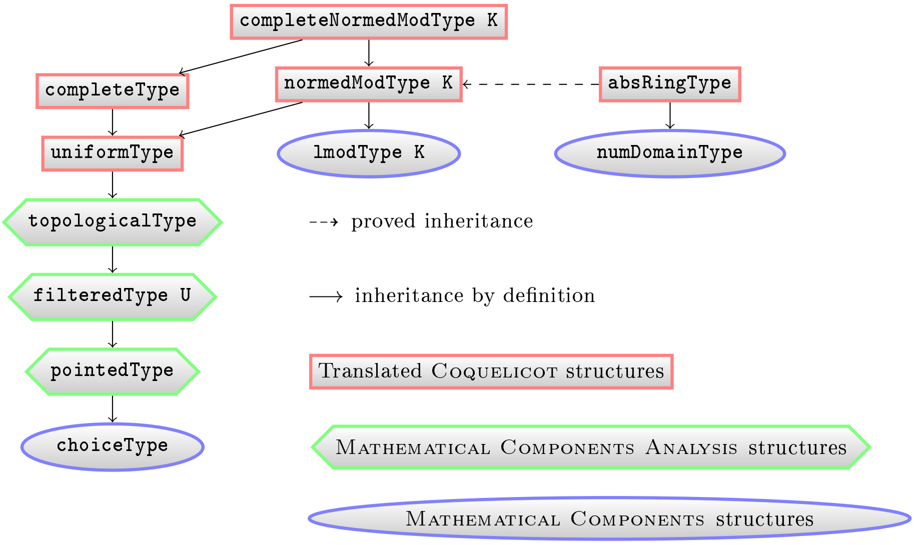

Fig. 2. Mathematical Components Analysis hierarchy

The main benefit of filters for analysis is to rephrase ε–δ phrasing into more concise statements. For instance, $f(\text{locally}(x)) \supseteq \text{locally}(y)$ stands for $\lim_x f = y$ and $((x, y) \mapsto (f(x), f(y)))(\texttt{entourages}) \supseteq \texttt{entourages}$ states that f is uniformly continuous. Preserving this abstraction also shortens the proofs.

2.2 About the Mathematical Components Analysis Hierarchy

Our library inherits several of its topological structures from the Coquelicot library (see Fig. 2). We remove the structures that can be replaced with those of the Mathematical Components library, *e.g.* the `Ring` structure from Coquelicot is replaced with Mathematical Components's `ringType`, and we reimplement the other structures. We mentioned in Sect. 2.1 the type representing uniform spaces (`uniformType`). We also reimplement the structure representing normed modules (`normedModType K`) over a ring `K` equipped with an absolute value (`absRingType`). In this work, the norm of `x` is denoted by `` `|[x]| ``. Finally, we also adapt to our context the types representing complete spaces (`completeType`), analogous to complete metric spaces but with a filter-based notion of convergence, and complete normed modules (`completeNormedModType K`). The corresponding formal definitions can be found in [ACM+18a, file `hierarchy.v`] together with their main properties, and we provide a short description of them in Appendix A.

Moreover, we extend Coquelicot's hierarchy with our own structures. Having uniform spaces at the bottom of the hierarchy as in the Coquelicot library makes some proofs harder or even impossible. In particular, Tychonoff's Theorem has a very concise proof in terms of filters where the topology induced by balls in a uniform space is not appropriate [Rou18].

Topological spaces come with their own notion of neighborhood: the set A is a neighborhood of p if A contains an open set B which contains p. Although the neighborhoods defined by balls (recall the definition of $\mathrm{locally}(x)$ in Sect. 2.1) are compatible with this notion of neighborhood for the uniform topology, some sets cannot be expressed as neighborhoods of a point in a topological space. Indeed, "neighborhoods of $+\infty$" (defined as `Rbar_locally +oo` in Sect. 2.3) are for instance subsets of $\mathbb{R}$ and $+\infty$ is not a point of $\mathbb{R}$.

In order to reconcile the different notions of neighborhoods, we put three structures below our copy of Coquelicot's hierarchy (as shown in Fig. 2). Elements of each of these structures have an implicit coercion to their carrier and can be used as if they were types.

—A structure for non-empty types, with a distinguished point, represented by the `pointedType` structure (see [ACM+18a, file `classical_sets.v`]).

—A family of types `T : filteredType U`, such that elements `t` of `T` represent sets of sets on `U`, through the filtered space operator `locally : T -> set (set U)`. This is just for sharing purposes, so we do not enforce that `locally t` is a filter yet. Moreover, having `T` different from `U` makes it possible to have `locally +oo` equal to `Rbar_locally +oo`, thanks to an instantiation of the `filteredType R` structure as the canonical filter on `R` associated to `+oo : Rbar`. We require `T` to be non-empty (see [ACM+18a, file `topology.v`] for details).

—Finally, a topological space structure `topologicalType`, for which we enforce that the `T` and `U` in the operator `locally` are the same and that `locally t` is exactly the proper filter generated by the filter base of open neighborhoods of `t` (see [ACM+18a, file `topology.v`] for details).

In the uniform space structure (copied from Coquelicot), we enforce that `locally t` also coincides with the filter generated by the filter base of uniform balls, which was not necessarily chosen the same as the basis for open sets.

These additions also give us the opportunity to provide shorter and generic notations, closer to the mathematical practice, in order to improve readability, as explained in the next section.

2.3 Notations for Limits and Convergence

In a previous work [CR17], we introduced notations in order to represent the convergence statement $\lim_x f = y$ as `f @ x --> y` in Coq. In fact, we provide the notation `f @ F` for the filter $f(F)$ and the notation `F --> G` for reverse filter inclusion ($F \supseteq G$). However, in the notation `f @ x --> y`, usually the variables `x` and `y` are not filters but points in a uniform space. Hence, we also have a mechanism based on canonical structures [MT13] to automatically infer the filter corresponding to the type of the point. For instance, if x is in a uniform space, then the neighborhood filter $\mathrm{locally}(x)$ is inferred, or if x is $+\infty$, or `+oo` in Coq using our notations, then it is Coquelicot's filter of "neighborhoods of $+\infty$" [BLM15]

$$\texttt{Rbar\_locally +oo} = \{A \mid \exists M.\,]M, +\infty[\subseteq A\}\,.$$

In the particular case of functions, dedicated canonical structures are defined to match their source type. If it is `nat`, then the function is a sequence, hence we infer the filter `u @ eventually`, where `eventually` is Coquelicot's equivalent of `Rbar_locally +oo` for sets of natural numbers, in order to be able to write `u --> y` for $\lim u = y$. If the source type is a function type, then we recognize in particular the case where `x` is a function of type `(T -> Prop) -> Prop`, hence a set of sets. The inferred filter is then `x` itself.

For pointed types (and hence every structure from our hierarchy), using our classical axioms from Sect. 5, we can define an Hilbert's epsilon function we call `get`. It takes a predicate `P` and outputs a point which satisfies `P` if there is one (and outputs a default point otherwise). This function makes it possible to define functions computing the limit of a function (see `lim_in` below), the generic `mklittleo` operator (see Sect. 4.2), or the differential of a function (see Sect. 4.5).

```
Definition lim_in {U : Type} {T : filteredType U} :=
  fun F : set (set U) => get (fun l : T => F --> l).
```

Here, the function `lim_in` takes as input a filter and outputs a limit of `F` if there is one; `T` defines canonical filters on `U`. We say then that `l` is a limit of `F` if the canonical filter associated to `l` is contained in `F`. In particular, if the filter `F` is of the form `f @ x` for some function `f` and some point `x`, then `lim_in F` is the limit of `f` at point `x`. We provide the notation `[lim F in T]` to represent the limit of filter `F` in `T : filteredType U`.

The `lim_in` function also makes it possible to express the fact that a filter or function converges without using an existential quantifier: a filter or function converges if and only if it converges to its limit.

```
Notation "[ 'cvg' F 'in' T ]" := (F --> [lim F in T]).

Lemma cvg_ex (U : Type) (T : filteredType U) (F : set (set U)) :
  [cvg F in T] <-> (exists l : T, F --> l).
```

We also provide the notation `cvg F`, which triggers the inference of `T` in order to build the term `[cvg F in T]`. The complete formalization of limits and convergence can be found in [ACM$^+$18a, file `topology.v`].

3. SMALL-SCALE FILTER ELIMINATION

Although filters are a good way to hide "epsilon-delta" in statements, in order to prove `F P` for some ultimately true proposition `P`, one might be tempted to replace the filter `F` by its definition. This may result in a breakage of abstraction and lead to longer and less stable proof scripts (*e.g.* if the filter slightly changes).

Libraries such as Coquelicot already provide tools to combine results on filters without doing any unfolding. We copy and extend the same tools in Sect. 3.1. We then show how to go one step further in the transparency of filters in Sect. 3.2. Section 3.3 explains how to phrase Cauchy filters so as to make their definition usable more easily by our tools. Finally, Sect. 3.4 illustrates our tools in action in a real proof and Sect. 3.5 details the differences between this proof and the corresponding one in Coquelicot.

3.1 Combining Filters by Hand

The defining properties of filters entail the following facts.

```
Lemma filter_app (T : Type) (F : set (set T)) : Filter F ->
  forall H G : set T, F (fun x => H x -> G x) -> F H -> F G.
```

```
Lemma filterE (T : Type) (F : set (set T)) : Filter F ->
  forall G : set T, (forall x, G x) -> F G.
```

The first lemma can be used to combine hypotheses of the form `F` $\mathtt{H}_i$ and a conclusion `F G` into `F (fun x =>` $\mathtt{H}_1$ `x -> ... ->` $\mathtt{H}_n$ `x -> G x)`, and the second lemma removes the filter so that we shall prove instead the (hopefully) simpler goal `forall x,` $\mathtt{H}_1$ `x -> ... ->` $\mathtt{H}_n$ `x -> G x`.

However this forces forward reasoning, since the user has to anticipate every fact $\mathtt{H}_i$ `x` that will be used in the proof of `G x` beforehand. This means the statements $\mathtt{H}_i$ have to be written explicitly by the user, and they often depend on the choice of splitting of epsilons in the rest of the proof, which was also the main source of instability of proof scripts without using filters. This clearly appears in the proofs of the lemmas of the double limit theorem `filterlim_switch_1` and `filterlim_switch_2` in the Coquelicot library [BLM18, file `Hierarchy.v`].

We now show a novel method which absolves the user from providing explicitly the statements $\mathtt{H}_i$.

3.2 The Tactics `near=>`, `near·=>`, `near:`, and `end_near`

The purpose of this section is to explain the new `near` tactics we provide to perform small-scale filter elimination.

The basic principle of filter elimination is to make the user believe that instead of proving `F Q` she should instead prove `Q x` directly, where `x` can be in an arbitrarily precise set belonging to `F`. The lemma `filter_near_of` describes this formally[3]: the `in_filter` record is the type of sets belonging to `F`, hence `P` represents the arbitrarily precise set containing `x`.

```
Record in_filter T (F : set (set T)) := InFilter {
  prop_in_filter_proj : T -> Prop;
  prop_in_filterP_proj : F prop_in_filter_proj
}.

Lemma filter_near_of T F (P : in_filter T F) (Q : set T) :
  Filter F -> (forall x, prop_in_filter_proj P x -> Q x) -> F Q.
```

From now on, we sometimes use the notation `\forall x \near F, G x`, which is a notation for `F (fun x => G x)`. This should be read "for all `x` which is `near F`, `G x` holds", and we will use this phrasing instead of the too specific "ultimately true" or "eventually true". We also define the notation `x \is_near F` for `prop_in_filter_proj P x` for some `P : in_filter T F`. This notation deliberately hides `P` since it is not meant to be given by the user but rather instantiated trough the use of the `near` tactics.

3.2.1 *Using `near=>`, `near:` and `end_near`*

(1) The tactic `near=> x` performs an "introduction".
On a goal of the form `\forall x \near F, G x`, it puts into the local context a variable `x` and an hypothesis `x \is_near F` and yields the goal `G x`. The

[3] We have slightly simplified the presentation; the actual statement does not use directly the projection `prop_in_filter_proj` [ACM+18a].

latter hypothesis hides an existential variable ?H for the neighborhood to which x belongs, so that the membership proof F ?H is actually delayed. This is in fact a simple application of Lemma filter_near_of.

```
Tactic Notation "near=>" ident(x) :=
  apply: filter_near_of => x ?.
```

We call the x which is now in the local context a *near-variable*. A near-variable could be defined as variables x which come with an hypothesis x \is_near F, hiding an existential variable, in the local context.

(2) The tactic near: x "discharges" the near-variable x.
On a goal of the form H$_i$ x such that the hypothesis x \is_near F is in the context, it yields the goal \forall x \near F, H$_i$ x. This partially instantiates the existential variable ?H associated with the hypothesis x \is_near F as the intersection between H$_i$ and a fresh existential variable ?H'. The user is invited to prove the goal right away.
If H$_i$ had already been added to the set hidden in the hypothesis x \is_near F through a previous use of near: x, then it immediately closes the goal without modifying ?H.

(3) Once the main goal has been proved, there remain existential variables that have not been instantiated. These correspond to the ?H' in the last calls of the near: x tactic. They can be instantiated with the total set, since it belongs to any filter. This is the purpose of the end_near tactic.

3.2.2 *Using **near F => x**, **near:** and **end_near**.* Instead of acting on the goal, the tactic near F => x introduces a variable x along with the x \is_near F hypothesis, once again hiding an existential variable. After using near F => x, one may use near: and end_near in exactly the same ways as before. The tactic near F => x requires the filter F to be proper, *i.e.* no set H in F is empty.

3.2.3 *Combining all Near Tactics.* The tactics near=> x and near F => y may be combined any number of times, and in any order. Near-variables can be discharged by using near: z provided that the statement contains only variables introduced before z was. This limitation, guaranteed by Coq type checking, is legitimate as we must not be able to introduce circular dependencies in the existential variables.

For the detailed implementation of the near tactics, we refer the reader to the formalization [ACM+18a, file topology.v].

3.3 Rephrasing Concepts

Our methodology requires that some lemmas are phrased in a particular way. For example there are several equivalent ways to define a Cauchy filter. The most (ε–δ)-ish way is

```
Definition cauchy_ex {T : uniformType} (F : set (set T)) :=
  forall eps : R, 0 < eps -> exists x, F (ball x eps).
```

However it is easier to use the following equivalent definition:

```
Definition cauchy {T : uniformType} (F : set (set T)) :=
  forall e, e > 0 -> \forall x & y \near F, ball x e y.
```

Indeed, the existential quantification is then encapsulated in the `\near F` notation and can thus be treated in a systematic way in our proofs. This rephrasing could be disturbing for users that might not immediately see these are the same concepts. Hopefully they can be convinced by simple equivalence lemmas.

```
Lemma cauchyP (T : uniformType) (F : set (set T)) :
  ProperFilter F -> cauchy F <-> cauchy_ex F.
```

Although this particular equivalence is only true for proper filters, it is sufficient since in practice we manipulate only filters that are proper.

Note that the point of view of `uniformType` in terms of entourages leads to an even more compact equivalent definition, which we use as an intermediate step in the proof of `cauchyP`.

```
Lemma cauchy_entouragesP (T : uniformType) (F : set (set T)) :
  Filter F -> cauchy F <-> (F, F) --> entourages.
```

Here, the filter associated to `(F, F)` in the `· --> ·` notation is inferred as the product filter `filter_prod F F` thanks to the canonical structures mentioned in Sect. 2.3.

In the same vein, our definition of big-$\mathcal{O}$ in Sect. 4.1, which is equivalent to standard ones, encapsulates both existential quantifiers from the mathematical definition in the `\forall \near` notation to work better with the `near` tactics.

3.4 Use-Case: a Short Completeness Proof

We detail a proof that the type of functions from an arbitrary (choice) type to a complete type is again complete. This proof is interesting for several reasons. First, it illustrates our main technical contributions: it uses all of our tactics and demonstrates our use of filters, in particular, this proof uses two filters on two different types. Second, it shortens the original proof in Coquelicot [BLM18, Lemma `complete_cauchy_fct`, file `Hierarchy.v`] from about 40 lines to 7 lines (see Sect. 3.5 for a brief explanation of the differences between the two proofs), by removing in particular the three explicit witnesses. Finally, it shows how our work leads to formal proofs that look like informal ones: arguments can be stated without being cluttered by technical constructions of witnesses (see line 5), the latter being delayed and constructed by resorting to lemma applications (see lines 5–6), which makes for shorter and more stable proof scripts.

The proof script we explain below is part of the formalization accompanying this article [ACM+18a, file `hierarchy.v`].

```
Lemma fun_complete (T : choiceType) (U : completeType)
 (F : set (set (T -> U))) {FF : ProperFilter F} : cauchy F -> cvg F.
Proof.
```

Before all, observe that the implicit type of the Cauchy filter is not `T -> U` as it may appear at first sight; it is actually inferred to be `fct_uniformType T U`, the type of functional metric spaces, which is a `uniformType`, as required by the definition of Cauchy filter (see Sect. 3.3). The mechanism at work here is (again) the one of canonical structures [MT13].

We start by proving that for all t of type T, the filter $\{\{f(\mathtt{t}) \mid f \in A\} \mid A \in \mathtt{F}\}$ is Cauchy in U. This filter can be expressed succinctly as soon as one observes that it is the image of the filter F by the function fun f => f t. More precisely, it can be written ((fun f => f t) @ F) using the infix notation @ that denotes the image of a filter (see Fig. 1). Line 1 below states that this filter is Cauchy[4]; line 2 proves this fact. The proof is very simple since it is a direct consequence of cauchy F (the local hypothesis labeled Fc introduced at the beginning of line 1).

```
1 move=> Fc; have /(_ _)/complete_cauchy Ft_cvg : cauchy (@^~_ @ F).
2   by move=> t e ?; rewrite near_simpl; apply: filterS (Fc _ _).
```

At this stage, we have to prove cvg F, knowing that cauchy F holds as well as forall t : T, cvg ((fun f => f t) @ F). This latter hypothesis is a direct consequence of the subgoal we explained just above. It results from the application performed at line 1 of Lemma complete_cauchy to the following subgoal, proved by line 2, forall t : T, cauchy ((fun f => f t) @ F). Under these hypotheses, the function fun t => lim ((fun f => f t) @ F) is the pointwise limit of the filter F. We now prove that this limit is uniform.

```
3 apply/cvg_ex; exists (fun t => lim (@^~t @ F)).
```

Under the same hypotheses as before, we now have to prove:

```
F --> (fun t : T => lim ((fun f => f t) @ F)).
```

Since the right-hand side is a point of T -> U, it is interpreted as the filter of neighborhoods of this point. So it suffices to prove, for all e such that e > 0, that we have

```
\forall f \near F, ball (fun t : T => lim ((fun f => f t) @ F)) e f.
```

This goal transformation is achieved at the beginning of line 4 by the application of the Lemma flim_ballPpos. After application of this lemma, we are in a position to use the near tactics as we will explain shortly.

```
4 apply/flim_ballPpos => e; near=> f => t; near F => g => /=.
```

The first near tactic used at line 4 has the following consequence: we are asked to prove for all f which are near F and for all t that

```
ball (lim ((fun f => f t) @ F)) e (f t)
```

holds. The proof goes by introducing a g, which is near F as well; this is also achieved at line 4 by the second near tactic.

We then split the ball around (g t) (using Lemma ball_splitl, see line 5) and are left to prove two goals:

(1) ball (lim ((fun f => f t) @ F)) (e / 2) (g t), and

(2) ball (f t) (e / 2) (g t)

[4] We display the actual proof script where fun f => f t is abbreviated @^~ t using Mathematical Components notations.

which will both be true when `g` is `near F`, so that we will use the `near: g` tactic. We claim that this reasoning is an informal one in the sense that this proof step does not need to be interleaved with technical proofs that can be handled later as mere consequences of `near` facts.

The first goal can be proved by `Ft_cvg` (the hypothesis obtained from line 1). The terseness of SSReflect tactics actually allows us to prove it on the same line of proof script as the application of Lemma `ball_splitl`:

```
5 apply: (@ball_splitl _ (g t)); first by near: g; exact/Ft_cvg/
     locally_ball.
```

The second goal can be proved for all values of `t`, when `f` is `near F`, so that we first generalize `t`, using `move: (t)`, and then discharge `g` and `f` using `near:` twice:

```
6 by move: (t); near: g; near: f; apply: nearP_dep; apply: filterS (Fc
      _ _).
```

After calling `near: f`, we have to prove

```
\forall f \near F, \forall g \near F,
  forall t, ball (f t) (e / 2) (g t).
```

This is achieved by using Lemma `nearP_dep` (see line 6). We can conclude because the filter `F` is Cauchy and `e / 2` is obviously positive (this is automatically proved by using another set of canonical structures).

The last line of the proof script is for proving automatically the remaining trivial existentials:

```
7 Grab Existential Variables. all: end_near. Qed.
```

3.5 Differences with the Proof in Coquelicot

Several differences occur between both the statements and the proofs of Lemma `fun_complete` (Sect. 3.4) and Lemma `complete_cauchy_fct` (Coquelicot library [BLM18, file `Hierarchy.v`]). Let us clarify them.

First, let us compare the statements. Both Lemma `fun_complete` and Lemma `complete_cauchy_fct` state that for any proper filter on the function space `T -> U`, if it is Cauchy, then it converges, but both the definitions of Cauchy filters and filter convergence differ. For Cauchy filters, we use the `cauchy` predicate from Sect. 3.3 while `cauchy_ex` is used in Coquelicot. This forces the authors of Coquelicot to use in their proof an additional lemma [BLM18, Lemma `cauchy_distance`, file `Hierarchy.v`], which states the equivalence between `cauchy_ex` and a predicate which is very close to `cauchy` once the `\forall \near` notation is unfolded.

For filter convergence, the authors of Coquelicot use an epsilon phrasing which amounts to:

```
\forall eps, 0 < eps -> F (ball (lim F) eps).
```

We choose instead the `cvg F` notation (recall Sect. 2.3), which is equivalent to `F --> lim F`, *i.e.* `F` contains the neighborhood filter of `lim F`. In our definition, balls are thus abstracted through the use of the topology. However, our proof of Lemma `fun_complete` still resorts to balls so we have to use an additional lemma [ACM+18a, Lemma `flim_ballPpos`, file `hierarchy.v`] in order to get back the epsilon phrasing.

These two differences taken into account, the two proofs follow the same reasoning. Apart from the impact of the `near` tactics, the remaining differences in the proofs are due to naming conventions and to slight reformulation of some lemmas.

4. MECHANIZATION OF BACHMANN-LANDAU NOTATIONS

When Donald Knuth addresses the editor of the Notices of the American Mathematical Society about teaching calculus, he insists on using the big-$\mathcal{O}$ notation such as it blends smoothly into equational reasoning [Knu98]. "[I]t significantly simplifies calculations because it allows us to be sloppy but in a satisfactorily controlled way." He goes as far as "dream[ing] of writing a calculus text entitled $\mathcal{O}$ Calculus".

This section synthesizes the key ideas that mechanize Knuth's dream in a provably correct fashion. We explain the basic intent of our mechanization in Sect. 4.1, show how we recover an equational view with little-o and big-$\mathcal{O}$ notations in Sect. 4.2, describe a few aspects of the equational theory in Sect. 4.3, and provide concrete evidence of its usefulness in Sections 4.4–4.6.

4.1 The Notations $f = o(e)$ and $f = \mathcal{O}(e)$

The little-o and big-$\mathcal{O}$ notations are traditionally defined by

$$f = o_0(e) \text{ or } f(x) = \underset{x\to 0}{o}(e(x)) \Leftrightarrow \forall\varepsilon>0.\,\exists\delta>0.\,\forall x.\,|x|<\delta \Rightarrow |f(x)| \leqslant \varepsilon|e(x)|,$$
$$f = \mathcal{O}_0(e) \text{ or } f(x) = \underset{x\to 0}{\mathcal{O}}(e(x)) \Leftrightarrow \exists k>0.\,\exists\delta>0.\,\forall x.\,|x|<\delta \Rightarrow |f(x)| \leqslant k|e(x)|.$$

For the sake of readability we gave the definitions of these notions at a neighborhood of 0, but they are generalized to any filter in our library [ACM+18a, file `landau.v`].

The "equality" in the notation $f = o(e)$ is a well-known abuse of notation. Indeed it is neither symmetric, since one cannot write $o(e) = f$, nor transitive, since $f = o(e)$ and $g = o(e)$ do not imply $f = g$ and not even $f \sim g$ (cf. Sect. 4.4).

In fact, $f = o(e)$ should be read as "f *is a* little-o of e". It is not rare to see this reading enforced by the notation "$f \in o(e)$" in undergraduate-level teaching, allegedly to prevent students' confusion (see for example in [AF88], a textbook from the eighties still popular in France). It is therefore no surprise to find $o_0(e)$ viewed as a set of functions, or equivalently a predicate on functions, even in recent formalizations [GCP18].

Our formalization builds on the set-theoretic notation using a type-theoretic variant. Indeed, we provide both a predicate `littleo_def` for functions that are little-o of other functions at some filter, and a sigma-type `littleo_type`. Similarly for big-$\mathcal{O}$, we provide a predicate `bigO_def` and a type `bigO_type`. The formal definitions of the predicates advantageously use the `\near` notation introduced in Sect. 3.2:

```
Context {T : Type} {K : absRingType} {V W : normedModType K}.

Let littleo_def (F : set (set T)) (f : T -> V) (e : T -> W) :=
  forall eps : R, 0 < eps ->
  \forall x \near F, ‘|[f x]| <= eps * ‘|[e x]|.
```

```
Let bigO_def (F : set (set T)) (f : T -> V) (e : T -> W) :=
  \forall k \near +oo, \forall x \near F, `|[f x]| <= k * `|[e x]|.
```

Regarding notational conventions in the remaining of this article, note that, like in the code snippet just above, `T` is a type, `K` is a ring equipped with an absolute value, and `V` and `W` are normed modules over `K`; they all are implicit arguments of forthcoming definitions.

The sigma-types `littleo_type` and `bigO_type` directly follow from the corresponding ternary predicates:

```
Structure littleo_type (F : set (set T)) (e : T -> W) :=
  Littleo {
    littleo_fun :> T -> V;
    littleoP : littleo_def F littleo_fun e
  }.

Structure bigO_type (F : set (set T)) (e : T -> W) :=
  BigO {
    bigO_fun :> T -> V;
    bigOP : bigO_def F bigO_fun e
  }.
```

For the sake of readability, we slightly simplified the definitions compared to the source code: we removed `Prop` to `bool` coercions and the `littleoP` and `bigOP` fields are lemmas deduced from unnamed fields in the actual code. The unaltered definitions can be found online [ACM+18a, file `landau.v`].

Let us comment more specifically on the structure `littleo_type`. It packs a function, namely the `littleo_fun` projection, with a proof that it is a little-o of `e`, providing us with the type of functions that are a little-o of another function. In particular, we can inhabit this type with the null function (and the trivial proof that it is a little-o). Let us call `littleo0` this record with the null function. The type `littleo_type F e` is furthermore equipped with the notation `{o_F e}` to improve reading.

So `littleo_type` and `bigO_type` provides a type-theoretic variant of the set-theoretic notation for little-o and big-$\mathcal{O}$, but it can be argued that such a set-theoretic notation is misplaced because it precludes the equational viewpoint that Knuth advocates [Knu98], along with formal-proof practitioners, and even anachronistic now that today's students use symbolic algebra systems like Maple and WolframAlpha where the big-$\mathcal{O}$ notation appears in power series calculations.

In this article, we make a strong case for the equational viewpoint, and we explain in the next section how to recover it.

4.2 The Notations $f = g + o(e)$ and $f = g + \mathcal{O}(e)$

Indeed it is also in the folklore to write $f = g + o(e)$ to mean $f - g = o(e)$ in the previous acceptation. Since expressions involving the little-o notation are to be considered as classes of functions, the formula $f = g + o(e)$ suggests a reading in terms of a congruence relation. It might therefore seem like a good idea to formally define the corresponding equality and let it be denoted by a ternary notation. However, doing so carelessly might preclude routine mathematical practice, first because the bound e changes a lot from one equality to another, for example, if $f(x) = g(x) + \underset{x\to 0}{o}(x)$ then $xf(x) = xg(x) + \underset{x\to 0}{o}\left(x^2\right)$. Second, mathematicians add little-o and big-$\mathcal{O}$ from various scales as in: "if $f(x) = g(x) + \underset{x\to 0}{o}(x)$ and $g(x) = \underset{x\to 0}{\mathcal{O}}\left(x^2\right)$ then $f(x) = \underset{x\to 0}{o}(x)$".

To reflect this mathematical practice, we decided to stress that $f = g + o(e)$ means "$f = g + h$ where h is a little-o of e", which is defined formally as follows.

Definition 4.2.1. We define $o(e)[h]$ to be h if h is a little-o of e, and 0 otherwise.

In particular, the statement $f = g + o(e)[h]$ means $f = g + h$ if h is little-o of e, and $f = g$ otherwise.

In Coq, to define $o(e)[h]$, we provide a function `mklittleo`[5] that builds a little-o from an arbitrary function. Given a function `h`, `mklittleo` tries to coerce it to the subtype of little-o's and, when it fails, it returns the null little-o (using the function `littleo0` mentioned in Sect. 4.1). This mechanism of partial projection into a subtype is provided by the generic operator `insubd` from the Mathematical Components library:

```
Definition mklittleo (F : filter_on T) (h : T -> V) (e : T -> W) :=
  littleo_fun (insubd (littleo0 F e) h).

Notation "[o_ x e 'of' h ]" := (mklittleo x h e)
  (at level 0, x, e at level 0, only parsing).
```

In order to avoid stating witnesses explicitly, we notice that if $f = g + h$, then $h = f - g$ hence h is a little-o of e if and only if $f - g$ is. This leads us to define the sought ternary notation to be:

```
Notation "f = g '+o_' x e" := (f = g + [o_x e of f - g]).
```

The ternary notation `f = g +o_x e` expands to `f = g + [o_x e of f - g]`. *We then deliberately hide the* `h` *in the printing of the notation so that* `[o_x e of h]` *prints back* `'o_x e`.

However, if we try to prove `f = g +o_x e` in a purely arithmetical way, we might rewrite with equations for `f` and `g` and finally get a goal of the form $o(\mathtt{e}) = o(\mathtt{e})$. In a paper-and-pencil proof, this is considered as trivial, but in a formal proof, both little-o hide functions h and h', and the statement to prove is in fact the equality $o(\mathtt{e})[h] = o(\mathtt{e})[h']$. In this situation there is very little chance that this unification succeeds, so our methodology consists in replacing h' by an existential variable $?h'$ as soon as possible. This is made possible because of the following observation:

$$f = g + o\,(e)\,[f - g] \Leftrightarrow \exists h.\ f = g + o\,(e)\,[h]\ , \tag{1}$$

which allows to replace a goal `f = g +o_x e` by a goal `f = g + [o_x e of ?h]` (printed `f = g + 'a_o_x e`) where `?h` is an existential variable.

4.3 Equational Theory

Our main concern is to preserve the benefits of the equational view of little-o and big-$\mathcal{O}$. That means developing a small theory containing the main "equations" one

[5] Here again the definition is simplified by removing `Prop` to `bool` coercions and phantom types.

may need in order to combine them easily. Once sufficiently many equations are proved, that allows the user to prove facts about little-o and big-$\mathcal{O}$ using informal reasoning, without having to go back to the definition of little-o and big-$\mathcal{O}$ and to do explicit local reasoning, except in particular cases where the theory lacks an equation (see Sect. 4.4 for examples where the filter characterization of little-o is completely abstracted from the proof).

We do not claim to have reached such a complete set of equations, but we proved a few equations that seemed important to us. Let us give examples. First, we have arithmetic rules for little-o and big-$\mathcal{O}$. For instance, little-o absorbs addition and the product of a $\mathcal{O}(h_1)$ and a $\mathcal{O}(h_2)$ is a $\mathcal{O}(h_1 \cdot h_2)$.

```
Context {F : filter_on T}.

Lemma addo (f g : T -> V) (e : T -> W) :
  [o_F e of f] + [o_F e of g] =o_F e.

Lemma mulO (h1 h2 f g : T -> R) :
  [O_F h1 of f] * [O_F h2 of g] =O_F (h1 * h2).
```

We also have a few rules combining little-o and big-$\mathcal{O}$. For example, a $o(e)$ is also a $\mathcal{O}(e)$ and a little-o of a $\mathcal{O}(g)$ is a $o(g)$.

```
Lemma littleo_eqO (e : T -> W) (f : {o_F e}) :
  (f : _ -> _) =O_F e.
```

```
Lemma littleo_bigO_eqo (g : T -> W) (f : T -> V) (h : T -> X) :
  f =O_F g -> [o_F f of h] =o_F g.
```

Of course, in order to prove this set of equations, local reasoning is necessary at some point. This is where the `near` tactics from Sect. 3.2 come into use.

For instance, let us have a look at the proof of Lemma `littleo_bigO_eqo`[6]. The function `f` is a $\mathcal{O}$(`g`) and the function `[o_F f of h]` is a o(`f`), either equal to `h` if it is a o(`f`), or to the null function (recall Sect. 4.2). Since the goal is to prove that the function `[o_F f of h]` is a o(`g`), the first step is to go back to the definition of little-o and introduce the universally quantified "epsilon". This is the application of lemma `eqoP` at line 1 that recovers the definition `littleo` seen in Sect. 4.1.

```
1  move->; apply/eqoP => _/posnumP[e]; have [k c] := bigO _ g.
```

In line 1, we also replaced `f` in `[o_F f of h]` with `[O_F g of f]`. We also abstract the latter function using a fresh function `k` and, since it was a big-$\mathcal{O}$ of `g`, we get as an hypothesis a positive constant `c` such that

```
\forall x \near F, ‘|[k x]| <= c * ‘|[g x]|.
```

At this point the goal to prove is

```
\forall x \near F, ‘|[k x]| <= c * ‘|[g x]| ->
\forall x \near F, ‘|[’o_(x \near F) (k x)]| <= eps * ‘|[g x]|.
```

[6] The proof script can be found in [ACM+18a, file `landau.v`].

We use `filter_app` to combine both statement so that we should now prove

```
\forall x \near F, `|[k x]| <= c * `|[g x]| ->
                      `|['o_(x \near F) (k x)]| <= eps * `|[g x]|.
```

Now we give ourselves an `x` which is `near F` thanks to the `near=> x` tactic.

```
2 apply: filter_app; near=> x.
```

We have to prove that

```
`|[k x]| <= c * `|[g x]| ->
  `|['o_(x \near F) (k x)]| <= eps * `|[g x]|
```

by manipulating the inequalities until we reach the goal

```
`|['o_(x \near F) (k x)]| <= eps / c * `|[k x]|
```

as the result of multiple rewritings and transitivity (see line 3).

```
3 rewrite -!ler_pdivr_mull //; apply: ler_trans; rewrite
     ler_pdivr_mull // mulrA.
```

The latter goal should be true for `x` which is `near F` since `'o_(x \near F) (k x)` is a little-o of `k` and `eps / c` is positive.

To finish the proof, we can now call the `near: x` tactic. At this point, the remaining goal is

```
\forall x \near F, `|['o_(x \near F) (k x)]| <= eps / c * `|[k x]|
```

which can be proved by using the filter characterization of little-o (at line 4).

```
4 by near: x; apply: littleoP.
```

The last line of the proof script calls the `end_near` tactic to dispose of the remaining existentials:

```
5 Grab Existential Variables. all: end_near. Qed.
```

4.4 Application: Asymptotic Equivalence

Two functions $f(x)$ and $g(x)$ are equivalent as x goes to a (notation: $f \sim_a g$) when $f = g + o_a(g)$. Thanks to the ideas explained in Sections 4.1 and 4.2 and to the equations already proved in Sect. 4.3, the fact that $\sim$ is an equivalence relation can be established by short proof scripts. For the sake of illustration, let us explain how we show that $\sim$ is symmetric and transitive (see [ACM$^+$18a, file `landau.v`] for details).

The symmetry of $\sim$ is mechanized as follows (`f ~_F g` is the Coq notation for $f \sim_F g$):

```
Context {F : filter_on T}.

Lemma equiv_sym (f g : T -> V) : f ~_F g -> g ~_F f.
Proof.
move=> fg; have /(canLR (addrK _))<- := fg.
by apply: eqaddoE; rewrite oppo (equivoRL _ fg).
Qed.
```

The first line of the proof script is made of standard tactics that change the goal to $f - o(g) \sim f$. Lemma `eqaddoE` implements the idea of (1): it introduces an existential variable $?h$ such that the goal becomes $f - o(g) = f + o(f)[?h]$. Rewriting with Lemma `oppo` turns $f - o(g)$ into $f + o(g)$ and Lemma `equivoRL` turns $o(g)$ into $o(f)$ (it uses the hypothesis $f \sim g$). The right- and left-hand sides can now be unified and the proof is completed.

The transitivity of $\sim$ is mechanized as follows:

```
Lemma equiv_trans (f g h : T -> V) : f ~_F g -> g ~_F h -> f ~_F h.
Proof.
by move=> -> ->; apply: eqaddoE; rewrite eqoaddo -addrA addo.
Qed.
```

After the application of Lemma `eqaddoE`, the goal is

$$h + o\,(h) + o\left(h + o\,(h)\right) \sim h + o\,(h)\,[?e],$$

where $?e$ is an existential variable. Lemma `eqoaddo` transforms $o(h + o(h))$ into $o(h)$ and Lemma `addo` transforms $o(h) + o(h)$ into $o(h)$. After rewriting, the goal is $h + o(h) \sim h + o(h)[?e]$, so that unification succeeds and completes the proof.

4.5 Application: Differential of a Function

We use the little-o notations, in combination with the `get` function from Sect. 2.3, in order to define the differential of a function:

```
Definition diff (F : filter_on V) (f : V -> W) :=
  (get (fun (df : {linear V -> W}) => forall x,
  f x = f (lim F) + df (x - lim F) +o_(x \near F) (x - lim F))).
```

where the `x` of `(x \near F)` is used to find the function hidden by the little-o, and, let us recall from Sect. 4.1, where `V` and `W` are normed modules over a ring `K` equipped with an absolute value (see [ACM+18a, file `derive.v`] for details).

We remarked in previous work [CR17, Rou18] that having such a function for the differential and using additional hypotheses that state which functions are differentiable makes proofs more natural and easier than using only a predicate stating that an expression is the differential of a function.

4.6 Application: Uniform Big-$\mathcal{O}$

Boldo et al. designed a notion of uniform big-$\mathcal{O}$ in their work on the numerical resolution of the wave equation [BCF+13]. They are interested in relations of the form

$$f\,(x, \Delta) = \underset{\Delta \to 0}{\mathcal{O}}\,(g\,(\Delta))\,,$$

but with a uniform definition with respect to x, *i.e.* with the following definition:

$$\exists C > 0.\ \exists \alpha > 0.\ \forall x.\ \forall \Delta.\ \ \|\Delta\| < \alpha \Rightarrow \|f\,(x, \Delta)\| < \|g\,(\Delta)\|\,.$$

We can formalize this uniform big-$\mathcal{O}$ as

```
(fun p => f p) =O_F (fun p => g p.2)
```

where `p` represents the pair (x, Δ) and `p.2` is thus Δ (recall Fig. 1), and with an appropriate choice for the filter `F`.

The filter `F` must be a filter on a Cartesian product and can be defined using the filter product function `filter_prod` (see Sect. 2.1). Its first argument is the filter corresponding to x. Since there is no constraint on x, we can choose the trivial filter containing only the total set. The second argument, corresponding to Δ, is the neighborhood filter of 0 (*i.e.* `locally 0`).

In fact, in their source code, Boldo et al. force Δ to be within a particular domain and use sigma types instead of the existential quantifier. We can still provide an equivalent definition thanks to a "restriction" operator on filters and prove that it is indeed equivalent to their definition thanks to the axioms from Sect. 5. The interested reader may refer to the source code for more details [ACM+18a, file `misc/uniform_bigO.v`].

5. COQ AXIOMS TO MAKE CLASSICAL ANALYSIS PRACTICAL

The formal tools we have explained so far have been developed with classical analysis in mind. However, the native logic of Coq (the Calculus of Inductive Constructions) is constructive. Fortunately, there exist several axioms that can be safely added. In this section, we explained the axioms on which our development relies. In short, they bring to the logic of Coq: extensionality, the law of excluded middle, and the axiom of choice. These are arguably much-needed axioms for classical analysis and it is therefore no wonder that they are important to make the Mathematical Components Analysis library practical. In the following, we explain how they are used in the tools we described in this article. For the sake of presentation, we divide our four axioms into two categories: the *extensionality* axioms and the *classical* axioms.

5.1 Extensionality Axioms

The first two axioms on which our development relies are "extensionality" axioms:

```
propext: forall (P Q : Prop), (P <-> Q) -> (P = Q).
funext: forall {T U : Type} (f g : T -> U),
       (forall x, f x = g x) -> f = g.
```

The first one is sometimes called "propositional extensionality": it expresses that two logically equivalent propositions are intentionally equal. The second one, called "functional extensionality", expresses that two functions that are pointwise (or extensionally) equal are (intentionally) equal. These "extensionality" axioms are justified by multiple models including the simplicial model of type theory. In particular, they are a consequence of the univalence axiom [Uni13] (up to replacing `Prop` in our statements by `hProp` and equality by relevant identity). We justified them using the axioms propositional_extensionality and functional_extensionality_dep from the Coq standard library.

Theoretically, many theorems proved using these axioms might in practice be established without them, at the cost of changing the notion of equality. Indeed, we could replace many uses of the generic (Leibniz) Coq equality with a weaker form of equality such as pointwise equality for functions or equivalence for propositions. However, this would make proofs considerably longer, for no particular benefits. Moreover, the main proof assistants where classical logic is primitive (*i.e.* Isabelle/HOL, HOL Light, HOL4, Mizar) also have a primitive equality which is extensional for functions and propositions, and so does Lean.

These extensionality axioms are particularly useful in stating and reasoning about equality on sets and equality on filters (which ultimately boils down to the former). Indeed, since we represent sets on a type T as functions of type T -> Prop the combination of the two axioms helps state and reason about equality of functions whose target is a proposition. This is so pervasive that we even have the following lemma (as well as several variants):

```
Lemma predeqE {T} (P Q : set T) : (P = Q) = (forall x, P x <-> Q x).
```

5.2 Classical Axioms

The other two axioms on which our development relies are “classical” axioms:

```
pselect: forall (P : Prop), {P} + {~P}.
gen_choiceMixin: forall {T : Type}, Choice.mixin_of T.
```

The axiom pselect is a strong version of the law of excluded middle, where the result can be used to give a value in Type. The axiom gen_choiceMixin states the existence of a choice function on any type, which we do not detail here as it is specific to the Mathematical Components library (see [MT16, Sect. 7.3]). These two axioms are justified by the axiom constructive_indefinite_description from the Coq standard library, which is one formulation of Hilbert’s epsilon, and by the axiom of propositional extensionality from Sect. 5.1. This is best illustrated by the proof of the axiom pselect. The latter is established in part as a consequence of the standard law of excluded middle (*i.e.* in Prop), which we prove as Diaconescu’s theorem [Dia75] using both indefinite description and propositional extensionality.

These classical axioms are used in particular in our implementation of Bachmann-Landau notations. The latter indeed relies on an operation that takes a function, finds out whether it is a little-o and outputs the proof when it is the case, and otherwise returns the null little-o. We do not know if there is a constructive alternative, like taking the minimum of two functions, in order to force it to be a little-o.

In contrast, it is likely that the near tactics still work without classical axioms, since their ancestor bigenough did work to prove facts about Cauchy reals [Coh12b].

6. RELATED WORK

Our work takes its starting point in the re-implementation of the Coquelicot library [BLM15] to make it fully compatible with the Mathematical Components library [GAA+13], in order to be able to combine algebra and analysis in the same framework. We alter Coquelicot's hierarchy by adding more structures (see Fig. 2), and with notations that make formal statements closer to the mathematical ones (see Fig. 1 and Sect. 2.3). We reformulate many definitions and theorems involving asymptotic reasoning to take advantage of the `near` tactics, which make proofs shorter. On the other hand, several parts of the Coquelicot library have not been adapted to our new context yet (mostly, sequences, integrals, and series).

The Coquelicot library also contains ternary predicates defining little-o and asymptotic equivalence of functions. Our definitions are basically the same (in particular the ternary predicate `littleo`) but their theory is not quite developed in Coquelicot. We provide a set of notations (see Fig. 3) and a more substantial equational theory on top of our definitions, which make them easier to manipulate. We also have notations and a theory for big-$\mathcal{O}$.

The Coquelicot library provides total functions to compute the limit and the derivative of a function. They are however restricted to functions from $\mathbb{R}$ to $\mathbb{R}$. We define a limit function for any function whose domain and codomain are equipped with canonical filters and a differential function for any function whose domain and codomain are normed modules. The crucial difference is that we include the existence of choice functions in our hierarchy at the cost of additional axioms, which give us these functions for free, while in the Coquelicot library they are constructed from the limited principle of omniscience.

Avigad and Donnelly's formalization in Isabelle/HOL [AD04] views big-$\mathcal{O}$ as sets. They describe inclusion and equational reasoning on big-$\mathcal{O}$ at the set level, and they manage to prove the prime number theorem using it. Thirteen years later, Eberl improves and extends their work by providing, in addition to big-$\mathcal{O}$, the little-o, Ω, ω, and Θ notations, in order to prove the complexity of "divide-and-conquer" algorithms [Ebe17]. Coupled with Isabelle/HOL's "heavy automation", his Landau symbols halve the size of his proofs [Ebe17, Sect. 3.2.2]. His Landau symbols are defined using the `eventually` construct of the standard library that applies a predicate to a filter. Formal proofs therefore enjoy the `eventually_elim` tactic that automates the application of filter-related lemmas together, and is often combined with other lemma collections (such as `field_simps`). The tactic `eventually_elim` is a simpler form of the `near` tactics, well adapted to Isabelle/HOL proof style. Indeed, when using `eventually_elim` one lists upfront a list of hypothesis that will be used by the automated proof search. Using `near`, these sets are *inferred* at the appropriate places while writing the proofs in an imperative style.

Guéneau et al. [GCP18] have developed in Coq a library to formalize the time-complexity of OCaml programs. To represent asymptotic bounds, they provide a formalization of the big-$\mathcal{O}$ notation. Similarly to us, their definition relies on filters, but only on finite filter products of the `eventually` filter (see Sect. 2.3) and its equivalent in $\mathbb{Z}$. Furthermore, they define a type for types equipped with *one filter*, while we make it possible to have *a different filter for each element of the type*.

However, in the face of the difficulties encountered to reproduce the (apparently sloppy) manipulation of the big-$\mathcal{O}$ notation, they give up on producing proofs "as simple [...] as their paper counterparts", choose to formalize the big-$\mathcal{O}$ notation as a dominance relation, and deprive themselves of Coq equational reasoning capabilities. Their library would require extension with the little-o notation and to arbitrary filters for it to "have other applications in mathematics". In comparison, our work already provides both notations, retains equational reasoning, and already fits together with a hierarchy of mathematical structures [ACM$^+$18a] designed on the model of Mathematical Components [GGMR09, MT16].

Their proofs also use delayed production of witnesses of existential quantifiers in the particular case of the computation of cost functions. They use Procrastination [Gué18], a small library of tactics similar to our `near` tactics. Both our work and the Procrastination library are a generalization of the `bigenough` library from previous work [Coh12b], which only dealt with statements that are eventually true in $\mathbb{N}$. The main difference between Procrastination and the `near` tactics is the following. To prove a given goal using Procrastination, one can introduce variables and accumulate properties about them. Once the goal is proved, the user is asked to provide an actual value for these variables which satisfies every accumulated property. Our work is more centered on filters: we do not have to provide a witness of the satisfiability of a predicate, but only to prove that the accumulated predicates belong to a given filter. The implementation of Procrastination also have more tactics, which are more complex, while we try to minimize them, following the *Small-Scale Reflection* strategy [MT16].

A particular formalization of big-$\mathcal{O}$ to be mentioned is the one by Boldo et al. [BCF$^+$13]. Their uniform big-$\mathcal{O}$ can be expressed with our definition through the choice of the appropriate filter, as described in Sect. 4.6.

Finally, filters *à la* Hölzl, Immler and Huffman [HIH13], are also used in an ongoing formalization of classical analysis in Lean [Lea17].

7. CONCLUSION

In this work, we provide a set of techniques and notations (notations are summarized in Fig. 3) in order to make asymptotic reasoning as smooth as possible in Coq. We integrate a mechanism for filter inference into a hierarchy of mathematical structures [ACM$^+$18a], together with notations and definitions that make filter manipulation easier.

We define tactics that make it possible to delay the instantiation of existential witnesses in order to allow for "rigorous asymptotic hand-waving". These tactics are more generally designed to do filter elimination while avoiding forward reasoning.

We then take advantage of our new framework to design equational Bachmann-Landau notations and to develop a small theory of little-o and big-$\mathcal{O}$ that removes all explicit local reasoning from some proofs.

Future Work. We plan to build a full classical analysis library, with convergence criteria for infinite sums or integrals based on asymptotic comparison, and also infinite sums and integrals of little-o, big-$\mathcal{O}$ and equivalences. We plan in particular to experiment with the same kind of construction as the `mklittleo` function for upper bounds, limits, derivatives and differentials.

Our strategy in this work is to provide a minimalistic set of tactics that makes it easier to build a small library in the tradition of the Mathematical Components library [G$^+$15]: our tactics are "small-scale" [MT16] (they perform elementary steps, hence proofs are more stable) and we focus on proving a collection of

Notations used in Sections 3 and 4 (see [ACM+18a, file `topology.v`] for details):

`\forall x \near F, G x`	"for all `x` near `F`, `G x` holds", *i.e.* `F G` where `F` is a filter over `T` and `G : T -> Prop` (a set over `T`)
`\forall x & y \near F, H x y`	`(filter_prod F F)(fun x => H x.1 x.2)` where `F` is a filter over `T` and `H : T -> T -> Prop`
`x \is_near F`	`x` is in a set that belongs to `F`

little-o notations used in Sect. 4 (see [ACM+18a, file `landau.v`] for details and big-$\mathcal{O}$ notations):

`{o_F e}`	the type `littleo_type F e` of little-o's of `e`
`[o_F e of f]`	a function with a canonical structure of little-o of `e`, `f` if it is indeed a little-o and the null function otherwise
`f = g +o_F e`	`f = g + [o_F e of f - g]`
`f =o_F e`	`f` is a little-o of `e` near `F`, *i.e.* `f = (mklittleo F f e)`
`[littleo of f]`	recovers the canonical structure of little-o of `f`
`'o_F e`	printing of `[o_F e of h]` (`h` is hidden)
`'a_o_F e`	printing of `[o_F e of ?h]` (`?h` is an existential variable)
`f ~_F g`	`f` and `g` asymptotic equivalence
`f x = g x +o_(x \near F)(e x)`	`f x = g x + h x` where `h` is the function hidden by the little-o of `e`

Fig. 3. Summary of the new notations introduced in this article

reusable lemmas that hides the most technical parts. Other strategies exist, see for example [Ebe17, GCP18] that we already discussed in Sect. 6.

ACKNOWLEDGMENTS

We thank Assia Mahboubi for her feedback on Bachmann-Landau notations, Assia Mahboubi and Guillaume Melquiond for their feedback on the near tactics and Assia Mahboubi and Pierre-Yves Strub for designing a library for real numbers from which we drew inspiration. We are also grateful to COQUELICOT authors Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond for their inspiring library and the many conversations we have had. We also thank Yves Bertot and Laurence Rideau for their feedback on the draft. Finally, we would like to thank anonymous reviewers who gave us many constructive comments which allowed to improve the presentation of our work.

This work was partially funded by the ANR project *FastRelax* (ANR-14-CE25-0018-01) of the French National Agency for Research and by JSPS KAKENHI Grant Number 15H02687.

References

[ACM+18a] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. Analysis library compatible with Mathematical Components. https://github.com/math-comp/analysis/releases/tag/0.1.0 (last acess: 2018/10/01), 2018.

[ACM+18b] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. Classical analysis with Coq. In *Coq Workshop 2018, Oxford, UK, July 8, 2018*, Jul 2018. 2-pages abstract available at https://staff.aist.go.jp/reynald.affeldt/documents/coqws-reals.pdf (last access: 2018/09/28); presentation slides available at http://cyrilcohen.fr/CoqWS2018.pdf (last access: 2018/09/28).

[AD04] Jeremy Avigad and Kevin Donnelly. Formalizing O notation in Isabelle/HOL. In David A. Basin and Michaël Rusinowitch, editors, *Proceedings of the Second International Joint Conference on Automated Reasoning, IJCAR 2004, Cork, Ireland, July 4–8, 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2004.

[AF88] Jean-Marie Arnaudiès and Henri Fraysse. *Cours de mathématiques*, volume 2, Analyse. Dunod, 1988.

[AM17] Mauricio Ayala-Rincón and César A. Muñoz, editors. *Proceedings of the 8th International Conference on Interactive Theorem Proving, ITP 2017, Brasília, Brazil, September 26–29, 2017*, volume 10499 of *Lecture Notes in Computer Science*. Springer, 2017.

[Bac94] Paul Bachmann. *Die Analytische Zahlentheorie*. B.G. Teubner, 1894.

[BCF$^+$13] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[Ber17] Sophie Bernard. Formalization of the Lindemann-Weierstrass theorem. In Ayala-Rincón and Muñoz [AM17], pages 65–80.

[BJMD$^+$10] Nicolas Brisebarre, Mioara Joldes, Erik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Pasca, Laurence Rideau, and Laurent Théry. CoqApprox. Available at http://tamadi.gforge.inria.fr/CoqApprox/ (last access: 2018/09/28), 2010. Version 2.0.0. Now part of the CoqInterval library (see http://coq-interval.gforge.inria.fr/).

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[BLM18] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. The Coquelicot library. Available at: http://coquelicot.saclay.inria.fr/ (last access: 2018/09/28), May 2018. Version 3.0.2.

[BPP13] Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors. *Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.

[Coh12a] Cyril Cohen. Construction of real algebraic numbers in Coq. In Lennart Beringer and Amy P. Felty, editors, *Proceedings of the Third International Conference on Interactive Theorem Proving, ITP 2012, Princeton, NJ, USA, August 13–15, 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2012.

[Coh12b] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, Nov 2012.

[Coq18] The Coq Development Team. *The Coq proof assistant reference manual*, 2018. Version 8.8.0.

[CR17] Cyril Cohen and Damien Rouhling. A Formal Proof in Coq of LaSalle's Invariance Principle. In Ayala-Rincón and Muñoz [AM17], pages 148–163.

[Dia75] Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.

[Ebe17] Manuel Eberl. Proving divide and conquer complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017.

[G$^+$15] Georges Gonthier et al. The Mathematical Components repository. https://github.com/math-comp/math-comp (last access: 2018/09/28), 2015. Full list of contributors: https://github.com/math-comp/math-comp/blob/master/AUTHORS.

[GAA$^+$13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Blazy et al. [BPP13], pages 163–179.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming on Programming Languages and Systems, ESOP 2018, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 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.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[GMT16] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016.

[Gon11] Georges Gonthier. Point-free, set-free concrete linear algebra. In *Proceeding of the 2nd International Conference on Interactive Theorem Proving, ITP 2011, Berg en Dal, The Netherlands, 22–25 August, 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2011.

[Gué18] Armaël Guéneau. Procrastination, a proof engineering technique. In *Coq Workshop 2018, Oxford, UK, July 8, 2018*, July 2018. 2-pages abstract available at http://gallium.inria.fr/~agueneau/publis/procrastination.pdf (last access 2018/09/28); source code available at https://github.com/Armael/coq-procrastination (last access: 2018/09/28).

[HIH13] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In Blazy et al. [BPP13], pages 279–294.

[Knu98] Donald E. Knuth. Teach Calculus with Big O. *Notices of the AMS*, 45(6):687–688, Jun/Jul 1998. Letter to the editor of the Notices of the American Mathematical Society. Full version available at: https://www-cs-faculty.stanford.edu/~knuth/calc.

[Lan09] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. B.G. Teubner, 1909.

[Lea17] The Lean mathematical components library developers. Lean mathematical components library. https://github.com/leanprover/mathlib (last access: 2018/09/28), 2017. Work in progress.

[MT13] Assia Mahboubi and Enrico Tassi. Canonical Structures for the Working Coq User. In Blazy et al. [BPP13], pages 19–34.

[MT16] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: https://math-comp.github.io/mcb/ (last access: 2018/09/28), 2016. With contributions by Yves Bertot and Georges Gonthier. Version of 2018/08/11.

[Rou18] Damien Rouhling. A Formal Proof in Coq of a Control Function for the Inverted Pendulum. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 28–41. ACM, 2018.

[Sch11a] Daniel Schepler. coq-topology: Topology library for Coq. Available at https://github.com/coq-contribs/topology (last access: 2018/09/28), 2011.

[Sch11b] Daniel Schepler. coq-zorns-lemma: Naive set theory library for Coq. Available at https://github.com/coq-contribs/zorns-lemma (last access: 2018/09/28), 2011.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Proceedings of the 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 278–293. Springer, 2008.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book (last access: 2018/09/28), Institute for Advanced Study, 2013.

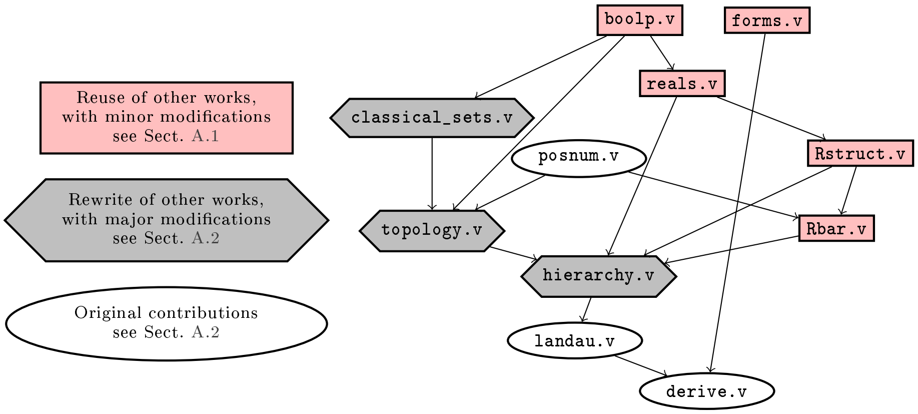

Fig. 4. Mathematical Components Analysis files in release 0.1.0

A. THE MATHEMATICAL COMPONENTS ANALYSIS LIBRARY

The Mathematical Components Analysis library [ACM+18a] is work-in-progress [ACM+18b]. It serves both as context to build the techniques developed in the present article and as a way to apply them to numerous examples. We believe the techniques of this article are valid, regardless of the methodology for packaging canonical structures or the inference techniques at use, hence the contents of this Appendix are not needed *stricto sensu*. However, we make here our context more precise, for the user to understand exactly which structures and mechanisms we use in our own implementation.

Figure 4 provides an overview of the files of Mathematical Components Analysis. The contributions from this article are located in the file `topology.v` for filters and `near` tactics (see Sect. 3), and in the file `landau.v` for little-o and big-$\mathcal{O}$ notations (see Sect. 4). One can find basic results and applications in these same two files. Numerous more elaborated examples of use can be found in files `hierarchy.v` and `derive.v`, including the examples we give in this article about Cauchy completeness and differentials.

Section A.1 is about the first part of the library (`boolp.v`, `reals.v`, `Rstruct.v`, `Rbar.v`, `forms.v`). This part is based on previous work or work-in-progress . We do not enter into details and mostly refer to available resources when they exist (*e.g.* the documentation embedded in the Coq files, publications by their authors, "historical information"), or provide explanations otherwise.

Section A.2 depicts the second part of the library (`posnum.v`, `classical_sets.v`, `topology.v`, `hierarchy.v`, `landau.v`, and `derive.v`). This part was introduced by the authors of the current article to have a ground to develop and test the asymptotic tools. We provide more details for that part. In particular, `topology.v` and `hierarchy.v` are based on previous work but required substantial adjustments which are described in Sections 2.1 and 2.2; we provide here more technical details at the risk of being redundant with existing literature.

A.1 Library Files Based on Previous Work

A.1.1 `boolp.v`, `reals.v`, *and* `altreals/*.v`.

All these files were originally written by Assia Mahboubi and Pierre-Yves Strub, and are now part of Mathematical Components Analysis.

The file `boolp.v` introduces the axioms explained in Sect. 5 and was edited from the original version to support our work, essentially by adding the axiom `gen_choiceMixin`, along with other minor contributions.

This file `reals.v` provides a classical axiomatization of real numbers, as a discrete real archimedean field with a least upper bound operator, which we use in a non-critical way as an abstraction of the reals from the Coq standard library. This axiomatic has been indeed instantiated in file `Rstruct.v` (see below), but there are some plans to provide a model solely based on axioms from `boolp.v`.

We did not modify the files `altreals/*.v`. They were written before the existence of the Mathematical Components Analysis library, and are thus independent from all of the files below. They are waiting to be ported to the new framework, and we expect their rewriting will benefit greatly from the present work.

A.1.2 `Rbar.v` *and* `Rstruct.v`.

Both files deal with the reals of the Coq standard library.

The file `Rbar.v` originally comes from the Coquelicot library [BLM18] and has been mostly rewritten using `posnum.v`.

To the best of our knowledge, the file `Rstruct.v` has been first released in the CoqApprox library [BJMD+10]. Its original version established that the type of reals `R` of the standard library of Coq is a field type from the Mathematical Components library. It has then been extended by Sophie Bernard in her formalization of the Lindemann-Weierstrass Theorem [Ber17] to provide `R` with the `rcfType` structure [Coh12a, Coh12b] from the Mathematical Components library. We have extended it again to `R` with the `realType` structure from `reals.v`.

A.1.3 `forms.v`.

This file is a work-in-progress that predates Mathematical Components Analysis. It was written by Cyril Cohen and Laurence Rideau and is in an active development branch of the Mathematical Components library. It will disappear from Mathematical Components Analysis once merged into Mathematical Components. It is only used to state generic differentiation of bilinear applications, thus encompassing derivation rules for the scalar product on `R` and for dot products on finite dimensional vector spaces.

A.2 Library Files Introduced with this Article

The files introduced with this article provide in particular several interfaces corresponding to mathematical structures. These interfaces come as extensions of existing Mathematical Components interfaces. The methodology for extension is the one of *packed classes* [GGMR09]. Below, when it comes to interfaces, we present only the *mixin* and the *class*.

A.2.1 `posnum.v`.

This file provides small-scale automation to rule out proofs of positivity for ε's. Its use is illustrated in Line 7 of the proof script of Lemma `fun_complete` in Sect. 3.4. We use canonical structures as an inference mechanism [MT13].

A.2.2 `classical_sets.v`. This file develops a theory of sets represented by a predicate over a type (the type `set`) with Mathematical Components-like notations (indeed, Mathematical Components only provides finite sets). Some of these notations can be seen in Fig. 1. This is inspired by previous work by Cyril Cohen and Damien Rouhling [CR17].

This file has been extended with *pointed types*. Pointed types are represented by the type `pointedType` that extends the type `choiceType` [MT16, Sect. 7.3] of Mathematical Components with a canonical inhabitant:

```
Definition point_of (T : Type) := T.
Record class_of (T : Type) := Class {
  base : Choice.class_of T;
  mixin : point_of T
}.
```

Pointed types sit at the bottom of the Mathematical Components Analysis hierarchy, as described in Sect. 2.2.

The formalization of Zorn's lemma by Daniel Schepler [Sch11b] has been ported to this setting.

A.2.3 `topology.v`. The role of this file is explained in Sect. 2.2. It provides, among other structures and results, the following types.

Types with Canonical Filters. The type `filteredType U` extends pointed types with an operator `locally` such that for any `t` of type `filteredType U`, the object `locally t` is a set of sets around `t` over the type `U`. In subsequent structures, this will be identified with the filter of neighborhoods of `t`.

```
Definition locally_of U T := T -> set (set U).

Record class_of U T := Class{
  base : Pointed.class_of T;
  locally : locally_of U T
}.
```

Topological Spaces. They are represented by the type `topologicalType` that extends `filteredType` with an operator `open` and three axioms:

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

Record class_of (T : Type) := Class {
  base : Filtered.class_of T T;
  mixin : mixin_of (Filtered.locally_op base)
}.
```

Given a `T` of type `topologicalType`, for any `t` of type `T`, the set of sets `locally t` is now the proper filter of neighborhoods of `t` (see Sect. 2.1). Indeed, it is defined using the filter base of the `open`'s of `t`.

The proof of Tychonoff's Theorem by Daniel Schepler [Sch11a] has been ported to this setting.

The file `topology.v` also contains the `near` notations and tactics explained in Sect. 3.

A.2.4 `hierarchy.v`. The structures described in this file are all translated from Coquelicot [BLM15, BLM18], and rebased onto structures from Mathematical Components, as explained in Sect. 2.2.

Uniform Spaces. They are represented by the type `uniformType` that extends topological spaces with a notion of `ball` and four axioms:

```
Definition locally_ {T T’} (ball : T -> R -> set T’) (x : T) :=
   @filter_from R _ [set x | 0 < x] (ball x).
Record mixin_of (M : Type) (locally : M -> set (set M)) := Mixin {
  ball : M -> R -> M -> Prop ;
  ax1 : forall x (e : R), 0 < e -> ball x e x ;
  ax2 : forall x y (e : R), ball x e y -> ball y e x ;
  ax3 : forall x y z e1 e2, ball x e1 y -> ball y e2 z ->
        ball x (e1 + e2) z;
  ax4 : locally = locally_ ball
}.

Record class_of (M : Type) := Class {
  base : Topological.class_of M;
  mixin : mixin_of (Filtered.locally_op base)
}.
```

The operator `filter_from` is explained in Sect. 2.1.

Complete Spaces. They are represented by the type `completeType`, which extends uniform spaces with the axiom that proper filters that satisfy the Cauchy property converge:

```
Definition axiom (T : uniformType) :=
  forall (F : set (set T)), ProperFilter F ->
    cauchy F -> F --> lim F.

Record class_of (T : Type) := Class {
  base : Uniform.class_of T ;
  mixin : axiom (Uniform.Pack base T)
}.
```

The property `cauchy` on a filter is defined in Sect. 3.3, the arrow `-->` is explained in Sect. 2.3 and originally comes from [CR17], and `lim` is explained in Sect. 2.3.

Rings with Absolute Value in $\mathbb{R}$. They are represented by the type `absRingType` which now extends Mathematical Components's `numDomainType` [Coh12b, Chapter 4] with an absolute value in `R`:

```
Record mixin_of (D : ringType) := Mixin {
  abs : D -> R;
  ax1 : abs 0 = 0 ;
  ax2 : abs (- 1) = 1 ;
  ax3 : forall x y : D, abs (x + y) <= abs x + abs y ;
  ax4 : forall x y : D, abs (x * y) = abs x * abs y ;
  ax5 : forall x : D, abs x = 0 -> x = 0
}.

Record class_of (K : Type) := Class {
  base : Num.NumDomain.class_of K ;
  mixin : mixin_of (Num.NumDomain.Pack base K)
}.
```

This structure was modified from Coquelicot by replacing the inequality in axiom `ax4` by an equality. Moreover this type should disappear in future work, and simply be replaced by `numDomainType`.

Normed Modules. They are represented by the type `normedModType`, which now extends Mathematical Components's `lmodType` [G+15, file `ssralg.v`] with the mixins for pointed, filtered, topological, and uniform types:

```
Record mixin_of (K : absRingType) (V : lmodType K) loc
    (m : @Uniform.mixin_of V loc) := Mixin {
  norm : V -> R ;
  ax1 : forall (x y : V), norm (x + y) <= norm x + norm y ;
  ax2 : forall (l : K) (x : V), norm (l *: x) = abs l * norm x;
  ax3 : Uniform.ball m = ball_ norm;
  ax4 : forall x : V, norm x = 0 -> x = 0
}.

Record class_of (T : Type) := Class {
  base : GRing.Lmodule.class_of K T ;
  pointed_mixin : Pointed.point_of T ;
  locally_mixin : Filtered.locally_of T T ;
  topological_mixin : @Topological.mixin_of T locally_mixin ;
  uniform_mixin : @Uniform.mixin_of T locally_mixin;
  mixin : @mixin_of _ (@GRing.Lmodule.Pack K (Phant K) T base T) _
      uniform_mixin
}.
```

The type `lmodType K` is a structure with an addition operation and a scaling operation with coefficients in `K` [Gon11].

This structure was modified from Coquelicot by replacing the inequality in axiom `ax2` by an equality.

Complete Normed Modules. Finally, the structure `completeNormedModType` combines `normedModType` with `completeType`:

```
Record class_of (T : Type) := Class {
  base : NormedModule.class_of K T ;
  mixin : Complete.axiom (Uniform.Pack base T)
}.
```

A.2.5 `landau.v` *and* `derive.v`. These two files are original developments that strongly rely on the tools we described in this article. They contain numerous results about limits, continuity, and differentiation presented using Bachmann-Landau notations and with proofs made shorter and more robust thanks to our contribution. For example, they include the following results:

—`eqolimP` (in `landau.v`): a function f converges to a limit l if and only if $f = l + o(1)$,

—`linear_for_continuous` (in `landau.v`): locally bounded linear functions are continuous,

—`differentiable_continuous` (in `derive.v`): differentiable functions are continuous,

—`is_diff_comp` (in `derive.v`): the differential of a composition is the composition of the differentials,

—`Rolle` and `MVT` (in `derive.v`): Rolle's Theorem and the Mean Value Theorem.