Formalization Techniques for Asymptotic Reasoning in Classical Analysis

Reynald Affeldt, Cyril Cohen, Damien Rouhling

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

Abstract

Formalizing analysis on a computer involves a lot of "epsilon-delta" reasoning, while informal reasoning may use some asymptotical 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.

Using filters (see Sect. 2) slightly improves the situation by hiding the arithmetic, but the explicit existential quantifiers are replaced by forward reasoning with statements that depend on how the proof will be led. We solve this problem (in Sect. 3) by giving a set of tactics and lemmas to handle existential variables in a consistent way.

Another common tool in informal classical analysis is asymptotical developments written using Bachmann-Landau notations, also known as 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 $f(x+h) = f(x) + df_x(h) + o(h)$. One performs arithmetic operations with developments written with Bachmann-Landau notations, 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 in a formal logic, as an equation on functions (this issue is discussed in more details in Sect. 4.1). We actually manage 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 big-$\mathcal{O}$ and little-o functions with arithmetic operations. The other formal proof libraries we know about either handle sets of functions or do not mix big-$\mathcal{O}$ and little-o (see Sect. 6 for related work).

We believe that the development discussed in this paper is 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+18] with analysis [ACM+18]. Both the development from this paper and its numerous applications—which we cannot all detail here—can be found in the latter repository.

Outline of this Paper We explain in Sect. 2 the concept of filter, successfully used in the Coquelicot library [BLM15] and the Isabelle/HOL library [HIH13], and we explain how we extend their ideas 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 method to delay proofs. 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 Bachmann-Landau notations and give a few examples of informal reasoning that can actually be done as such with them. In Sect. 5, we explain the axioms of the Coq standard library on which our development relies and why they are important. Finally, we discuss related work in Sect. 6 and conclude in Sect. 7.

About Notations In this paper, we introduce a number of notations that we explain as they appear (and summarize at the end of this paper 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 [CR17].

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. 2.1
`+oo`	$+\infty$

Figure 1: Notations from previous work used in this paper

This paper 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 paper (namely, `near=>`, `near:`, `end_near`, and `near have`), 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), and the `have` tactic (for forward reasoning).

Coq comes with a mechanism for implicit arguments: thanks to type constraints, some arguments in definitions can be inferred. We follow in this paper Coq's syntax for implicit arguments.

When giving a new definition, implicit arguments are declared using curly brackets, as in

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

Then, `arg1` is omitted in subsequent uses of `foo`, which all are of the form `foo 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. 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 typeclass 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
}.
```

The most important sort of filters used for analysis and local reasoning is the notion of neighborhood filter. 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 of pairs that is a "neighborhood" of the diagonal $\Delta = \{(x, x) \mid x \in T\}$, *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.

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

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

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.

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 $T * U$ and is defined as the set of supersets of the family $(P_1 `*` P_2)_{P_1 \in F, P_2 \in G}$ where $A `*` B = \{(a, b) \mid a \in A, b \in B\}$.

```
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]).

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

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

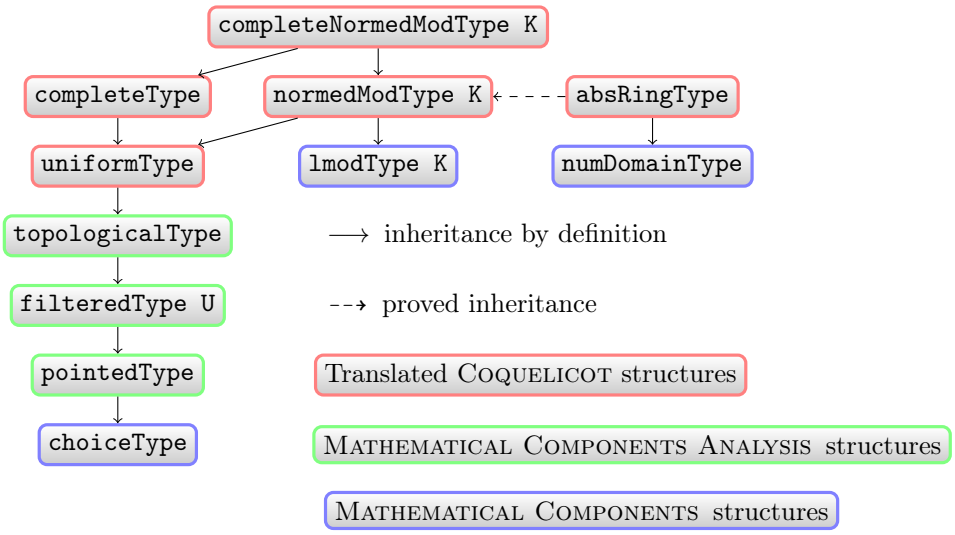

Figure 2: Mathematical Components Analysis hierarchy

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+18, file `hierarchy.v`].

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 adapted [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 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 nonempty types, with a distinguished point, represented by the `pointedType` structure (see [ACM+18, file `set.v`]).
- A family of types `T : filteredType U`, such that elements `t` of `T` represent sets of sets on the type `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 it to be non empty (see [ACM+18, 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+18, 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 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 differential of a function (see Sect. 4.4) or the generic `mklittleo` operator (see Sect. 4.3).

```
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+18, 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 changes slightly).

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.

3.1 Combining Filters by Hand

The axioms 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` H_i and a conclusion `F G` into `F (fun x =>` H_1 `x -> ... ->` H_n `x -> G x)`, and the second lemma removes the filter so that we shall prove `forall x,` H_1 `x -> ... ->` H_n `x -> G x` instead.

However this forces forward reasoning, since the user has to anticipate every fact H_i `x` that will be used in the proof of `G x` beforehand. This means the statements 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 [BLM17, file `Hierarchy.v`].

We now show a novel method which absolves the user from providing explicitly the statements H_i.

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

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 G` she should instead prove `G x` directly, where `x` can be in an arbitrarily precise set of `F`. The lemma `filterP` describes this formally:

```
Lemma filterP (T : Type) (F : set (set T)) (FF : Filter F)
  (G : set T) :
  (exists2 H : set T, F H & forall x : T, H x -> G x) <-> F G.
```

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".

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

1. The tactic `near=> x` starts by applying `filterP`, then provides an existential witness `H`, delays the membership `F H` for later, and tags the property `H x` with the variable `x`, to remember that it is `H` that should be progressively instantiated when we say that `x` is `near F`.

   ```
   Tactic Notation "near=>" ident(x) :=
     (apply/filterP; eexists=> [|x /(tag_nearI x) ?]; last first).
   ```

2. Now the user thinks she is proving `G x` but may enrich the constraints on `x` as she goes. Indeed every time she encounters a goal of the shape $\texttt{H}_i\,\texttt{x}$, she can now call `near: x`. This adds $\texttt{H}_i$ to the existential variable `H` by intersection, and closes the current goal: this goal has now been delayed in its "filter" form: `\forall x \near F,` $\texttt{H}_i$ `x` must be proved in the third phase.

3. Finally, when every main subgoal has been proved, the user is left to prove that an intersection of properties is in the filter: $\bigcap_i \texttt{H}_i \in \texttt{F}$, and the tactic `end_near` can be called to get many subgoals of the form

   `\forall x \near F,` $\texttt{H}_i$ `x.`

   Ideally, each one should be trivial: an hypothesis or an element from the filter base of `F`. Sometimes, however, one may rephrase the subgoal in terms of another filter, before solving it directly, or calling `near=> x` again.

Using `near F have x`, `near:` and `end_near`

Instead of acting on the goal, the tactic `near F have x` introduces a variable `x`, that will be `near F`. This means that, we may assume $\texttt{H}_i$ `x` is true for any $\texttt{H}_i$ in `F`. After using `near F have x`, one may use `near:` and `end_near` in exactly the same ways as before. The tactic `near F have x` requires the filter `F` to be proper, *i.e.* no set `H` in `F` is empty.

Combining all Near Tactics

The tactics `near=> x` and `near F have y` may be combined any number of times, and in any order. Goals can be delayed 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+18, 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 $(\varepsilon{-}\delta)$-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.

Note that the point of view of `uniformType` in terms of entourages leads to an even more compact equivalent definition.

```
Definition cauchy_entourage {T : uniformType} (F : set (set T)) :=
  (F, F) --> entourages.
```

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 (`complete_cauchy_fct`) from about 40 lines to 8 lines, 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 6), the latter being delayed and constructed by resorting to lemma applications (see lines 7–8), which makes for shorter and more stable proof scripts.

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

```
Lemma fun_complete (T : choiceType) (U : completeType)
  (F : set (set (T -> U))) : 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 $\{\{f(\texttt{t}) \mid f \in A\} \mid A \in \texttt{F}\}$ 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`. The latter function can be written `@^~ t` and `@` is an infix notation for the image of a filter (see Fig. 1), so that the filter in question can be abbreviated as `(@^~ t @ F)`. Line 1 states that this filter is Cauchy and line 2 proves this fact. The proof is very simple since it is a direct consequence of `cauchy F` (the local hypothesis labeled `Fc` in the proof script).

```
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 (@^~ t @ F)` (a direct consequence of the application of Lemma `complete_cauchy` performed at line 1 to the subgoal `cauchy (@^~_ @ F)` we explained just above). Under these hypotheses, the function `fun t => lim (@^~ 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 (@^~ 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 (@^~ t @ F)) e f.
```

This goal transformation is achieved by Lemma `flim_ballP` at line 4. We are now in a position to use the `near=>` tactic.

```
4 apply/flim_ballP => _ /posnumP[e]; near=> f => [t|].
```

As a consequence of the application of the `near=>` tactic at line 4, we are asked to prove for all `t` and for all `f` which are `near F` that

```
ball (lim (@^~ t @ F)) e (f t)
```

holds. Now the proof goes by introducing a `g`, which is `near F` as well, using the following tactic:

```
5  near F have g => /=.
```

We then split the ball around `g` (using Lemma `ball_splitl`, see line 6) and are left to prove two goals:

1. `ball (lim (@^~ t @ F)) (e / 2) (g t)` for all `t`, and
2. `ball f (e / 2) g`

which will both be true when `g` is `near F`, so we call the `near: g` tactic, in order to delay their proof to later. The reasoning we just described is arguably an informal one, in the sense that we did not need to interleave it with technical proofs of mere consequences of `near` facts. Still, this step boils down to the following single line of proof script:

```
6    by apply: (@ball_splitl _ (g t)); last move: (t); near: g.
```

Of course, we now have to deal with the two following "delayed" facts:

1. `\forall g \near F, ball (lim (@^~ t @ F)) (e / 2) (g t)`, and
2. `\forall g \near F, ball f (e / 2) g`.

The first one can be proved by `Ft_cvg` (from line 1) and the second one can be proved when `f` is `near F`, so we call `near: f`:

```
7  by end_near; [exact/Ft_cvg/locally_ball|near: f].
```

Finally we have to prove

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

This goal can be reduced to

```
\forall x & y \near F, forall t : T, ball (x t) (e / 2) (y t)
```

using Lemma `nearP_dep` (see line 8). We can conclude because `F` is Cauchy and `e / 2` is obvious positive (this is automatically ruled out by using another set of canonical structures):

```
8  by end_near; apply: nearP_dep; apply: filterS (Fc _ _).
   Qed.
```

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 modulo-like 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 Sect. 4.4.

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

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

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

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+18, 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 still builds on the set-theoretic notation, using a type-theoretic variant, since we provide both a ternary predicate `littleo` for functions that are little-o of other functions at some filter (`bigO` for big-$\mathcal{O}$), and a sigma-type `littleo_type` (and similarly for big-$\mathcal{O}$). In the remaining of this paper, `T` is a type, `K` is a ring equipped with an absolute value (`absRingType`) and `V` and `W` are normed modules over `K` (*i.e.* of type `normedModType K`). They all are implicit arguments for the remaining definitions.

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

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

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

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

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

This structure `littleo_type` 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.

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 paper, 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. This can be naturally seen as an equality modulo and it might seem like a good idea to formally define this equality modulo and denote it 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 $xf(x) = xg(x) + \underset{x\to 0}{o}(x^2)$. 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 $g(x) = \underset{x\to 0}{\mathcal{O}}(x^2)$ then $f(x) = \underset{x\to 0}{o}(x)$".

In order 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`[3] 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]`. **Then, we 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 $o(\mathtt{e})[h] = o(\mathtt{e})[h']$. 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]` where `?h` is an existential variable (printed `f = g + 'a_o_x e`).

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 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

[3]For the sake of readability, we slightly simplified the definitions compared to the source code: we removed phantom types and `Prop` to `bool` coercions. The unaltered definitions can be found online [ACM+18, file `landau.v`].

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 $\mathcal{O}(h_1 * h_2)$.

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

Lemma mulO (F : filter_on T) (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 (F : filter_on T) (e : T -> W)
  (f : littleo_type F e) : f =O_F e.

Lemma littleo_bigO_eqo (F : filter_on T) (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`[4]. 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[eps] /=.
```

In line 1, we also replaced `f` in `[o_F f of h]` with `[O_F g of f]`. We will call this function `k` and, since it is by definition a $\mathcal{O}$(`g`), unfolding the definition of big-$\mathcal{O}$ we get as an hypothesis a constant `c` such that

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

```
2 set k := 'O g; have [/= _/posnumP[c]] := bigOP [bigO of k].
```

At this point the goal is to prove

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

In fact, if `x` is `near F`, the assumption on `c` is valid for `x` and is sufficient to prove this goal. So we give ourselves an `x` which is `near F` thanks to the `near=> x` tactic (see line 3), and we 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]|
```

which should be true for `x` which is `near F` since `'o_(x \near F)(k x)` is a o(`k`) and `eps / c` is positive. As a consequence, we call the `near: x` tactic (at line 5).

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

```
3  apply: filter_app; near=> x.
4    rewrite -!ler_pdivr_mull //; apply: ler_trans.
5    by rewrite ler_pdivr_mull // mulrA; near: x.
```

Since the main subgoal has been proved, we can call the `end_near` tactic and prove the remaining delayed goal

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

by using the filter characterization of little-o (line 6 below).

```
6  by end_near; rewrite /= !near_simpl; apply: littleoP.
   Qed.
```

4.4 Applications

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+18, file `landau.v`] for details).

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

```
Lemma equiv_sym (F : filter_on T) (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 : filter_on T) (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 $h + o(h) + o(h + o(h)) \sim h + o(h)[?e]$, 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.

Differential of a Function

We use these 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+18, 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.

5 Axioms

Our development relies on four axioms, which can be divided into two categories: the *extensionality* axioms and the *classical* axioms. These axioms are both useful in the development of the Mathematical Components Analysis library and the tools we described in this paper. Let us detail only their use in the tools.

5.1 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 two are "extensionality" axioms. Indeed, the first one is sometimes called "propositional extensionality", it expresses that two logically equivalent propositions are intentionally equal, and 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 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 Leibniz (generic) Coq equality with a weaker form of equality such as pointwise equality for functions or equivalence for propositions. However, this would make proof 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 and several variants.

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

5.2 Classical axioms

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

These other two are "classical" axioms. 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 [G+18]). These two axioms are justified by the axiom `constructive_indefinite_description` from Coq standard library, which is one formulation of Hilbert's epsilon.

Our implementation of Bachmann-Landau notations 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.

However, it is likely that the `near` tactics still work without classical axioms, since their ancestor `bigenough` did work to prove facts about Cauchy reals [Coh12].

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 [G+18], 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 one (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 sets 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*. Their proofs also use delayed production of witnesses of existential quantifiers in the particular case of the computation of cost functions. In their source code, they use a simplified `bigenough` tactic [Coh12], although they do not make explicitly reference to it in their paper. They also have more tactics, which are more complex, while we try to minimize them, following the *Small-Scale Reflection* strategy [MT16].

However, in the face of the difficulties encountered to reproduce the (apparently sloppy) ma-

Notations used in Sections 3 and 4 (see [ACM+18, 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`

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

`[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`

Figure 3: Summary of the new notations introduced in this paper

nipulation 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+18] designed on the model of Mathematical Components [GGMR09, MT16].

Finally, filters *à la* Hölzl, Immler and Huffman [HIH13], are also used in an ongoing formalization of classical analysis in Lean [lea18].

7 Conclusion

In this work, we provide a set of techniques and notations (notations are summarized in Fig. 3) in order to make asymptotical reasoning as smooth as possible in Coq. We integrate a mechanism for filter inference into a hierarchy of mathematical structures [ACM+18], 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 asymptotical hand-waving". This is actually a generalization of the `bigenough` library from previous work [Coh12], which only dealt with statements that are eventually true in $\mathbb{N}$.

We then take advantage of our new framework to design equational Bachman-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.

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+18]: our tactics are "small-scale" [MT16] (they perform elementary steps, hence proofs are more stable) and we focus on proving a collection of 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, Pierre-Yves Strub and Assia Mahboubi 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+18] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. Analysis library compatible with Mathematical Components. https://github.com/math-comp/analysis/tree/asymptotic_reasoning_paper, 2018. Work in progress.

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

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

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

[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.

[BLM17] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. The Coquelicot library. Available at: http://coquelicot.saclay.inria.fr/, Sep 2017. Version 3.0.1.

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

[Coh12] 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 Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2017.

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

[G+18] Georges Gonthier et al. The Mathematical Components repository. https://github.com/math-comp/math-comp, 2018. Full list of contributors: https://github.com/math-comp/math-comp/blob/master/etc/AUTHORS. Project started in 2006.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 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, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, 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.

[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. Letter to the editor of the Notices of the American Mathematical Society. https://www-cs-faculty.stanford.edu/~knuth/calc, Mar 1998.

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

[lea18] Lean mathematical components library. https://github.com/leanprover/mathlib, 2018. 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/, 2016. With contributions by Yves Bertot and Georges Gonthier.

[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.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.