Formalization Techniques for Asymptotic Reasoning in Classical Analysis

Reynald Affeldt, Cyril Cohen, Damien Rouhling

Formalization Techniques for Asymptotic Reasoning in Classical Analysis

Reynald Affeldt[1], Cyril Cohen[2], Damien Rouhling[2]

[1] National Institute of Advanced Industrial Science and Technology, Japan.
reynald.affeldt@aist.go.jp
[2] Université Côte d'Azur, Inria, France.
cyril.cohen@inria.fr, damien.rouhling@inria.fr

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 restores partially 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.

Keywords: Formal proofs. Coq. Classical Analysis. Bachmann-Landau Notations

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 provide a δ, here the minimum of the two δ_f, δ_g we can get from the hypotheses applied to $\frac{\varepsilon}{2}$. Giving explicitly δ 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. 1) improves slightly 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 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 [5,16]. Indeed, one often writes $f(x) = a_0 + a_1 x + \ldots + a_n x^n + \underset{x \to 0}{o}(x^n)$ and does arithmetic operations with such developments, and then uses laws like $\underset{x \to 0}{o}(x^n) + \underset{x \to 0}{o}(x^n) = \underset{x \to 0}{o}(x^n)$, which, at first sight, seem to be impossible to represent in a formal logic. Another common example is the definition of differential: it is the linear operator df_x such that $f(x+h) = f(x) + df_x(h) + o(h)$.

We provide 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 big-$\mathcal{O}$ and little-o, and allows to handle them in a purely equational manner.

We explain in Sect. 1 the concept of filter, successfully used in the Coquelicot library [7] and the Isabelle/HOL library [14], 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. 2, 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 give a few examples of scripts that have been considerably shortened and made more stable using this methodology. Finally, in Sect. 3, we introduce our Bachmann-Landau notations and give a few examples of informal reasoning that can actually be done as such with them.

The development discussed in this paper is available online as part of an on-going effort to provide Mathematical Components [12] with analysis [2].

1 Abstracting Asymptotic Statements using Filters

The use of filters in the Coquelicot library [7] and the Isabelle/HOL library [14] proved that they define a good abstraction for convergence proofs in analysis. We first recall in Sect. 1.1 the definition of filters and give a few examples. Then, in Sect. 1.2 we detail the structures and notations we use in order to make the use of filters more natural in Coq [20].

1.1 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:

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

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 locally(x) in the Coquelicot library [7] 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 a given 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 (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 [9]. 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.

```
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. 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 product of F and G is a filter on $T * U$ 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 (F : set (set T)) (P : 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 product, allowing us in particular to build the neighborhood filter of a vector in $\mathbb{R}^n$ as the n-ary 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 $f(F) = \{B \mid f^{-1}(B) \in F\}$, is a filter on U.

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. Most often we are interested only in proper filters, hence they are sometimes simply called filters (as in [13]).

The main benefit of filters for analysis is to rephrase $\varepsilon - \delta$ 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. Keeping this abstraction also shortens the proofs.

1.2 Improving Coquelicot's Hierarchy to get more Generic Notations

In a previous work [9], we introduced notations in order to represent the convergence statement $\lim_{x} f = y$ as `f @ x --> y` in Coq [20]. 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 [17] 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 $+\infty$" [7]

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

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 the present work, we use these canonical structures and notations in a slightly different way. Indeed, we extend Coquelicot's hierarchy with a few structures, among which is one for types which define canonical filters on another type. 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 [19].

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. 1.1) 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 $+\infty$" 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 two structures at the bottom of our copy of Coquelicot's hierarchy: one for topological spaces (`topologicalType`) and, below it, a family of structures indexed by an arbitrary type `U` (`filteredType U`). A type `T : filteredType U` is 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`.

In a topological space structure `T : topologicalType`, 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`.

Finally, in a uniform space `uniformType`, 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.

We also require that filtered, topological and uniform spaces are non empty. In combination with the classical axioms on top of which we work (the excluded middle and an extensional choice function), we can define a function `get` which 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` below) or the differential of a function (see Sect. 3.4). We remarked in previous work [9,19] 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.

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

Here, the function `lim` takes as input a filter and outputs a limit of `F` if there is one and `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 F` is the limit of `f` at point `x`. The `lim` 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).
```

The notation `[lim F in T]` allows to give explicitly the type defining the canonical filters on `U`. We also provide the notation `cvg F`, which triggers the inference of `T` in order to build the term `[cvg F in T]`.

2 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. 2.1. We then show how to go one step further in the transparency of filters in Sect. 2.2. Section 2.3 explains how to phrase Cauchy filters so as to make their definition usable more easily by our tools. Finally Sect. 2.4 illustrates our tools in action in a real proof.

2.1 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 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.

We now show a novel method which absolves the user from providing explicitly the statements `H`$_i$.

2.2 The Tactics near=>, near:, end_near and near have

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 (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 $\mathtt{H}_i$ x, she can now call near: x. This adds $\mathtt{H}_i$ 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, $\mathtt{H}_i$ 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: $\bigcap_i \mathtt{H}_i \in \mathtt{F}$, and the tactic end_near can be called to get many subgoals of the form:

   \forall x \near F, $\mathtt{H}_i$ 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 direclty, 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 $\mathtt{H}_i$ x is true for any $\mathtt{H}_i$ 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.

2.3 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. 3.1, 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.

2.4 Use-Case: a Short Completeness Proof

We detail a proof that the type of functions from an arbitrary type to a complete type is again complete. This proof is particularly interesting, first because it uses all of our tactics and relies on two filters on two different types. Second, it shortens the original proof in Coquelicot (`complete_cauchy_fct`) from about 40 lines to 8 lines and removes the three explicit witnesses. Finally, it makes the proof look like an informal one.

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

We start by proving that for all `t : T`, the filter `Ft` $= \{\{f(t) | f \in A\} | A \in F\}$ is Cauchy in U and hence converges for each `t`. We made the statement shorter by noticing that `Ft` is in fact the image filter by the functional that applies a function to `t`: `Ft = (fun f => f t) @ F`, and using Mathematical Components, this is abbreviated to `(@^~ t @ F)`. The proof is very simple since it is a direct consequence of `Fc : cauchy F`.

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

At this stage, we have to prove `cvg F`, knowing that `Fc : cauchy F` and `Ft_cvg : forall t : T, cvg (@^~ t @ F)`. Hence the function `fun t => lim (@^~ t @ F)` is the pointwise limit of the filter `F`. So we now try to prove that this limit is uniform.

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

So under the same hypotheses as before, we now have to prove that

`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` we have `\forall f \near F, ball (fun t : T => lim (@^~ t @ F)) e f`. Now we use the `near=> f` tactic.

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

and we are asked to prove for all `t` that `ball (lim (@^~ t @ F)) e (f t)` for all `f` which are `near F`. Now the informal proof goes by introducing a `g`, which is `near F` as well, using the `near F have g` tactic.

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

We then split the ball around `g` and are left to prove two goals:

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

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

We are now left to prove the two following "delayed" facts,

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

The first one can be proved by `Ft_cvg` and the second one can be proved when `f` is `near F`, so we call `near: f`.

```
  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`

which can be reduced to `\forall f & g \near F, ball f (e / 2) g` and we can conclude because `F` is Cauchy and `e / 2` is obviously positive.

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

3 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 [15]. “[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. 3.1, show how we recover an equational view with modulo-like little-o and big-$\mathcal{O}$ notations in Sect. 3.2, describe a few aspects of the equational theory in Sect. 3.3, and provide concrete evidence of its usefulness in Sect. 3.4.

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

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

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

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.

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. 3.4).

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 [3], 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 even in recent formalizations [13].

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-$\mathcal{O}$).

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

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

This structure packs a function (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.

However, it can be argued that the set-theoretic notation is misplaced because today's students use symbolic algebra systems like Maple and WolframAlpha where the big-$\mathcal{O}$ notation appears in power series calculations, and because it precludes the equational viewpoint that Knuth advocates [15], along with formal-proof practitioners.

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

3.2 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 $f(x) = g(x) + o(x)$ then $xf(x) = xg(x) + o(x^2)$. Second, mathematicians add little-o and big-$\mathcal{O}$ from various scales as in: "if $f = g + o(x)$ and $g = \mathcal{O}(x^2)$ then $f = o(x)$".

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

Definition 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] to build a little-o from an arbitrary function h. When h is not a little-o, `mklittleo` returns the null little-o `littleo0` (see Sect. 3.1):

```
Definition mklittleo (F : filter_on T) f h :=
  littleo_fun (insubd (littleo0 F h) f).

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:

[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.

```
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]` (printed `f = g + ’a_o_x e`) where `?h` is an existential variable.

3.3 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 and big-$\mathcal{O}$ and to do explicit local reasoning, except in particular cases where the theory lacks an equation (see Sect. 3.4 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 :
  [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 (e : T -> W) (f : littleo_type F e) : f =O_F e.

Lemma littleo_bigO_eqo F (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. 2.2 come into use. For instance, let us have a look at the proof of Lemma `littleo_bigO_eqo`.

The function `f` is a $\mathcal{O}(\texttt{g})$ and the function `[o_F f of h]` is a $o(\texttt{f})$, either equal to `h` if it is a $o(\texttt{f})$, or to the null function (recall Sect. 3.2). Since the goal is to prove that the function `[o_F f of h]` is a $o(\texttt{g})$, the first step is to go back to the definition of little-o and introduce the universally quantified “epsilon”.

```
move->; apply/eqoP => _/posnumP[eps] /=.
```

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 $\mathcal{O}(\texttt{g})$, unfolding the definition of big-$\mathcal{O}$ we get a constant `c` such that

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

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

At this point the goal is to prove

```
\forall x \near F, ‘|[o_F k of h] 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, and we prove that `‘|[k x]| <= c * ‘|[g x]|` implies `‘|[o_F k of h] x]| <= eps * ‘|[g x]|` by manipulating the inequalities until we reach the goal `‘|[[o_F k of h] x]| <= eps / c * ‘|[k x]|`, which should be true for `x` which is `near F` since `[o_F k of h]` is a $o(\texttt{k})$ and `eps / c` is positive. As a consequence, we call the `near: x` tactic.

```
apply: filter_app; near=> x.
  rewrite -!ler_pdivr_mull //; apply: ler_trans.
  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_F k of h] x]| <= eps / c * ‘|[k x]|
```

by using the filter characterization of little-o.

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

3.4 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 3.1 and 3.2 and to the equations already proved as mentioned in Sect. 3.3, 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.

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

```
Lemma equiv_sym F (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 (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. 1.2, 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.

Conclusion

In this work, we provide a set of techniques and notations 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 [2], 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 [8], 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.

Our development strongly relies on an alternative formulation of Hilbert's epsilon. Indeed, our implementation of Bachmann-Landau notations relies on a function 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 [8].

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 [12]: our tactics are "small scale" [18] (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 [10,13] in the Related Work section.

Related Work

Our work takes its starting point in the re-implementation of the Coquelicot library [7] to make it fully compatible with the Mathematical Components library [12], in order to be able to combine algebra and analysis in the same framework. We extend Coquelicot's hierarchy with structures for topological spaces and types that define canonical filters for another type and with notations that make formal statements closer to the mathematical ones. We reformulate many definitions and theorems involving asymptotic reasoning using the `near` tactics, which makes proofs shorter. On the other hand, several parts of the Coquelicot library have not been adapted to this new context yet (*e.g.* integrals and series, theorems about derivatives and differentials).

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 developped in Coquelicot. We provide a set of notations 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 [4] 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 [10]. Coupled with ISABELLE/HOL's "heavy automation", his Landau symbols halve the size of his proofs [10, Sect. 3.2.2]. However they rely on a decision procedure specialized for the logarithms one typically runs into when dealing with complexity. 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 combined application of filter-related lemmas together with other lemma collections (such as `field_simps`). However, in practice, intermediate goals whose proof is automated need to be explicitly stated, which lengthens proof scripts.

Guéneau et al. [13] 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 products of the `eventually` filter (see Sect. 1.2) 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` [8], 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 [18].

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 [2] designed on the model of MATHEMATICAL COMPONENTS [11,18].

Finally, filters *à la* Hölzl, Immler and Huffman [14], are also used in an ongoing formalization of classical analysis in LEAN [1].

Acknowledgements

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. Finally, we thank Yves Bertot and Laurence Rideau for their feedback on the draft.

References

1. Lean mathematical components library. https://github.com/leanprover/mathlib (2017), work in progress
2. Affeldt, R., Cohen, C., Mahboubi, A., Rouhling, D., Strub, P.Y.: Analysis library compatible with Mathematical Components. https://github.com/math-comp/analysis (2017), work in progress
3. Arnaudiès, J.M., Fraysse, H.: Cours de mathématique, vol. 2, Analyse. Dunod (1988)
4. Avigad, J., Donnelly, K.: Formalizing O notation in Isabelle/HOL. In: Basin, D., Rusinowitch, M. (eds.) Automated Reasoning. pp. 357–371. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
5. Bachmann, P.: Die Analytische Zahlentheorie. B.G. Teubner (1894)
6. Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.): Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7998. Springer (2013), http://dx.doi.org/10.1007/978-3-642-39634-2
7. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A User-Friendly Library of Real Analysis for Coq. Mathematics in Computer Science 9(1), 41–62 (2015), http://dx.doi.org/10.1007/s11786-014-0181-1
8. Cohen, C.: Formalized algebraic numbers: construction and first-order theory. Ph.D. thesis, École polytechnique (Nov 2012)
9. Cohen, C., Rouhling, D.: A Formal Proof in Coq of LaSalle's Invariance Principle. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 148–163. Springer (2017), https://doi.org/10.1007/978-3-319-66107-0_10
10. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. J. Autom. Reasoning 58(4), 483–508 (2017), https://doi.org/10.1007/s10817-016-9378-0
11. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 327–342. Springer (2009), https://doi.org/10.1007/978-3-642-03359-9_23
12. Georges Gonthier et al.: The Mathematical Components repository. https://github.com/math-comp/math-comp (2017), full list of contributors: https://github.com/math-comp/math-comp/blob/master/etc/AUTHORS
13. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: 27th European Symposium on Programming (ESOP 2018) (Apr 2018), to appear
14. Hölzl, J., Immler, F., Huffman, B.: Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In: Blazy et al. [6], pp. 279–294, http://dx.doi.org/10.1007/978-3-642-39634-2_21
15. Knuth, D.: Letter to the editor of the Notices of the American Mathematical Society. https://www-cs-faculty.stanford.edu/~knuth/calc (Mar 1998)
16. Landau, E.: Handbuch der Lehre von der Verteilung der Primzahlen. B.G. Teubner (1909)
17. Mahboubi, A., Tassi, E.: Canonical Structures for the Working Coq User. In: Blazy et al. [6], pp. 19–34, http://dx.doi.org/10.1007/978-3-642-39634-2_5

18. Mahboubi, A., Tassi, E.: Mathematical Components. Available at: https://math-comp.github.io/mcb/ (2016), with contributions by Yves Bertot and Georges Gonthier.
19. Rouhling, D.: A Formal Proof in Coq of a Control Function for the Inverted Pendulum. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 28–41. ACM (2018), http://doi.acm.org/10.1145/3167101
20. The Coq Development Team: The Coq proof assistant reference manual (2017), http://coq.inria.fr, version 8.7.1