Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi

Cyril Cohen, Kazuhiko Sakaguchi, Enrico Tassi

Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi

Cyril Cohen
Inria, Université Côte d'Azur, France
Cyril.Cohen@inria.fr

Kazuhiko Sakaguchi
University of Tsukuba, Japan
sakaguchi@logic.cs.tsukuba.ac.jp

Enrico Tassi
Inria, Université Côte d'Azur, France
Enrico.Tassi@inria.fr

Abstract

It is nowadays customary to organize libraries of machine checked proofs around hierarchies of algebraic structures [2, 6, 8, 16, 18, 23, 27]. One influential example is the Mathematical Components library on top of which the long and intricate proof of the Odd Order Theorem could be fully formalized [14].

Still, building algebraic hierarchies in a proof assistant such as Coq [9] requires a lot of manual labor and often a deep expertise in the internals of the prover [13, 17]. Moreover, according to our experience [26], making a hierarchy evolve without causing breakage in client code is equally tricky: even a simple refactoring such as splitting a structure into two simpler ones is hard to get right.

In this paper we describe $\mathcal{HB}$, a high level language to *build* hierarchies of algebraic structures and to make these hierarchies *evolve* without breaking user code. The key concepts are the ones of *factory*, *builder* and *abbreviation* that let the hierarchy developer describe an actual interface for their library. Behind that interface the developer can provide appropriate code to ensure backward compatibility. We implement the $\mathcal{HB}$ language in the `hierarchy-builder` addon for the Coq system using the Elpi [11, 28] extension language.

2012 ACM Subject Classification General and reference → General literature; General and reference

Keywords and phrases Algebraic Hierarchy, Packed Classes, Coq, Elpi, Metaprogramming, λProlog

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Modern libraries of machine checked proofs are organized around hierarchies of algebraic structures [2, 6, 8, 16, 18, 23, 27]. For example the Mathematical Components library for the Coq system [9] provides a very rich, ever growing, hierarchy of structures such as group, ring, module, algebra, field, partial order, order, lattice... The hierarchy does not only serve the purpose of organizing knowledge, but also to make it easy to exploit it. Indeed the interactive prover can take advantage of the structure of the library and the relation between its concepts to infer part of information usually left implicit by the user, a capability that turned out to be key to tame the complexity and size of the formal proof of the Odd Order Theorem [14].

The hierarchy of the Mathematical Components library is implemented following the discipline of Packed Classes initially introduced in [13] and later also adopted in [2] and [6]. We call Packed Classes a discipline, and not a language, because, in spite of its many virtues, it is unwieldy to use. In particular it leaks to the user many of the technical details of the Coq system. As a result, one needs to be a Coq expert in order to build or modify a hierarchy, and even experts make mistakes as shown in [26]. Another inconvenience of the

Packed Classes discipline is that even simple changes to the hierarchy, such as splitting a structure into two simpler ones, break user code.

In this paper we describe $\mathcal{HB}$, a high level language to *build* hierarchies of algebraic structures and to make these hierarchies *evolve* without breaking user code. The key concepts are the ones of *factory* and *abbreviation* that let the hierarchy developer describe an actual interface for their library. Behind that interface the developer can provide appropriate code to ensure backward compatibility. We implement the $\mathcal{HB}$ language by compiling it to a variant of the Packed Classes discipline, that we call *flat*, in the `hierarchy-builder` addon for Coq. We write this addon using the Elpi [11, 28] extension language.

To sum up, the main contributions of the paper are:

- the design of the $\mathcal{HB}$ language,
- the compilation of $\mathcal{HB}$ to the (flat) Packed Classes discipline, and
- the implementation of $\mathcal{HB}$ in the `hierarchy-builder` addon for Coq.

The paper is organized as follows. Via an example we introduce $\mathcal{HB}$ and its key ideas. We then describe the discipline of Packed Classes and we show how $\mathcal{HB}$ can be compiled to it. We then discuss the implementation of the Coq addon via the Elpi extension language and we position $\mathcal{HB}$ in the literature.

2 $\mathcal{HB}$ by examples: building and evolving a hierarchy

The first version of our hierarchy (that we name V1) features only two structures: `Monoid` and `Ring`.

```
1  HB.mixin Record Monoid_of_Type A := {
2    zero : A;
3    add : A -> A -> A;
4    addrA : associative add;             (* `add` is associative.                  *)
5    add0r : left_id zero add;            (* `zero` is the left and right neutral   *)
6    addr0 : right_id zero add;           (*   element with respect to `add`.       *)
7  }.
8  HB.structure Definition Monoid := { A of Monoid_of_Type A }.
9
10 HB.mixin Record Ring_of_Monoid A of Monoid A := {
11   one : A;
12   opp : A -> A;
13   mul : A -> A -> A;
14   addNr : left_inverse zero opp add;   (* `opp x` is the left and right additive  *)
15   addrN : right_inverse zero opp add;  (*   inverse of `x`.                      *)
16   mulrA : associative mul;             (* `mul` is associative.                  *)
17   mul1r : left_id one mul;             (* `one` is the left and right neutral    *)
18   mulr1 : right_id one mul;            (*   element with respect to `mul`.       *)
19   mulrDl : left_distributive mul add;  (* `mul` is left and right distributive   *)
20   mulrDr : right_distributive mul add; (*   over `add`.                          *)
21 }.
22 HB.structure Definition Ring := { A of Monoid A & Ring_of_Monoid A }.
```

In order to build a structure we need to declare some factories and later assemble them. One kind of factory supported by $\mathcal{HB}$, the simplest one, is called *mixin* and is embodied by a record, by convention named `axioms`, that gathers operations and properties.

Mixins are declared via the `HB.mixin` command that takes a record declaration with a type variable name and a possibly empty list of factories for that type. The code between lines 1 and 7 declares a mixin that can turn a naked type `A` into a monoid, hence we chose the name to be `Monoid_of_Type`.

The `HB.structure` command takes in input the definition S for a type `A` together with a list of factories for `A`. It registers a structure S in the hierarchy placing any definition specific to that structure inside a Coq module named `S`. Line 8 hence forges the `Monoid` structure.

Line 10 declares a second mixin collecting the operations and properties that are needed in order to enrich a monoid to a ring, hence the name `Ring_of_Monoid`. Indeed this time the type variable `A` is followed by `Monoid` that enriches `A` with the operations and properties of monoids. As a consequence `add` and `zero` can be used to express the new properties.

The last line declares the `Ring` structure to hold all the axioms declared so far. We can now inspect the contents of the hierarchy and then proceed to build a theory about abstract rings, register examples (instances) of ring structures and finally use the abstract theory on these examples.

```
1  Print Monoid.type. (* Monoid.type  :=  { sort : Type;  ... }                    *)
2  Check @add.        (* add           :   forall M : Monoid.type, M -> M -> M     *)
3  Check @addNr.      (* addNr         :   forall R : Ring.type, left_inverse zero opp add *)
4
5  Lemma addrC : forall R : Ring.type, commutative (@add R).
6  Proof. (* Proof by Hankel 1867, omitted for brevity *) Qed.
7
8  Definition Z_Monoid_axioms : Monoid_of_Type Z :=
9    Monoid_of_Type.Build Z 0 Z.add Z.add_assoc Z.add_0_l Z.add_0_r.
10
11 HB.instance Z Z_Monoid_axioms.
12
13 Definition Z_Ring_axioms : Ring_of_Monoid Z _ :=
14   Ring_of_Monoid.Build Z 1 Z.opp Z.mul
15     Z.add_opp_diag_l Z.add_opp_diag_r Z.mul_assoc Z.mul_1_l Z.mul_1_r
16     Z.mul_add_distr_r Z.mul_add_distr_l.
17
18 HB.instance Z Z_Ring_axioms.
19
20 Lemma exercise (m n : Z) : (n + m) - n * 1 = m.
21 Proof. by rewrite mulr1 (addrC n) -(addrA m) addrN addr0. Qed.
```

We can print the type for monoids as forged by $\mathcal{HB}$ (line 1). It packs a carrier, called `sort`, and the collection of operation and properties that we omit for brevity. We can also look at the type of two constants synthesized by $\mathcal{HB}$ out of the hierarchy declaration. Remark that while the names of the constants come from the names of mixin fields, their types differ. In particular they are quantified over a `Monoid.type` or `Ring.type`, and not a simple type `A` as in the mixins. Moreover we evince that the `sort` projection is declared as an implicit coercion [24] and is automatically inserted in order to make `M -> M -> M` a meaningful type for binary operations on the carrier of `M`. Last, we see that properties are quantified on (hence apply to) the structure they belong to but use, in their statements, operations belonging to simpler structures. For example `addNr` is a property of a ring but its statement mentions `add`, the operation of the underlying monoid.

We then follow the proof of Hankel [5] to show that ring axioms imply the commutativity of the underlying monoid (line 5). This simple example shows we can populate the theory of abstract rings with new results.

We use the `Monoid_of_Type.Build` *abbreviation* (line 8) in order to build an instance of the `Monoid` structure for binary integers `Z`. We then register that monoid instance as the canonical one on `Z` (line 11) via the command `hb.canonical`. We can similarly declare that `Z` forms a ring by using the `Ring_of_Monoid.Build` abbreviation (lines 13 and 18). Note that the `Ring_of_Monoid.Build` abbreviation is not a plain record constructor for `Ring_of_Monoid`, since that would require more arguments, namely the monoid ones (see the `_` at line 13).

The abbreviation synthesized by $\mathcal{HB}$ infers them automatically (as in [17, Section 7]) thanks to the `hb.canonical` declaration given just above.

From now on the axioms as well as the abstract theory of rings apply to integers, as shown in lemma `exercise`. The details of the proof do not matter here, what is worth pointing out is that in a single statement we mix monoid (e.g. `+`) and ring (e.g. `-`) operations and in the proof we use monoid axioms (e.g. `addrA`), ring axioms (e.g. `addrN`) and ring lemmas (e.g. `addrC`), all seamlessly.

2.1 Evolution of the hierarchy

We proceed by accommodating the intermediate structure of Abelian groups.

```
HB.mixin Record Monoid_of_Type A := { ... (* unchanged *) ... }.
HB.structure Definition Monoid := { A of Monoid_of_Type A }.

HB.mixin Record AbelianGroup_of_Monoid A of Monoid A := {
  opp : A -> A;
  addrC : commutative (add : A -> A -> A);
  addNr : left_inverse zero opp add;
}.
HB.structure Definition AbelianGroup := { A of Monoid A & AbelianGroup_of_Monoid A }.

HB.mixin Record Ring_of_AbelianGroup A of AbelianGroup A := {
  one : A;
  mul : A -> A -> A;
  mulrA : associative mul;
  mul1r : left_id one mul;              mulr1 : right_id one mul;
  mulrDl : left_distributive mul add;   mulrDr : right_distributive mul add;
}.
HB.structure Definition Ring := { A of AbelianGroup A & Ring_of_AbelianGroup A }.

Lemma addrN {R : AbelianGroup.type} : right_inverse zero opp add.
Proof. by move=>x; rewrite addrC addNr. Qed.
```

Some operations and properties were moved from the old mixin for rings into a newborn mixin `AbelianGroup_of_Monoid` that gathers the axioms needed to turn a monoid into an Abelian group. Consequently the mixin for rings is now called `Ring_of_AbelianGroup` (instead of `Ring_of_Monoid`) since it expects the type `A` to be already an Abelian group and hence gathers fewer axioms.

While operations moved from one structure to another, some properties undergo a deep change in their status. The lemma `addrC` part of the abstract theory of rings is now an axiom of Abelian groups, while `addrN` is no more an axiom of rings, but rather a theorem of the abstract theory of Abelian groups.

With this new version of the hierarchy, that we name V2, code written for version V1 breaks. For example the declaration of the canonical ring over the integers fails, if only because we do not have a `Ring_of_Monoid.Build` abbreviation anymore.

Our objective is to obtain a version of the hierarchy, that we name V3, that does not only feature Abelian groups but that is also backward compatible with V1.

2.2 The missing puzzle piece

The key to make a hierarchy evolve without breaking user code is the full fledged notion of *factory* (the mixins seen so far are degenerate, trivial, factories). Factories, like mixins, are packages for operations and properties but are not directly used in the definition of structures. Instead a factory is equipped with *builders*: user provided pieces of code that extract from the factory the contents of mixins, so that existing abbreviations can be used.

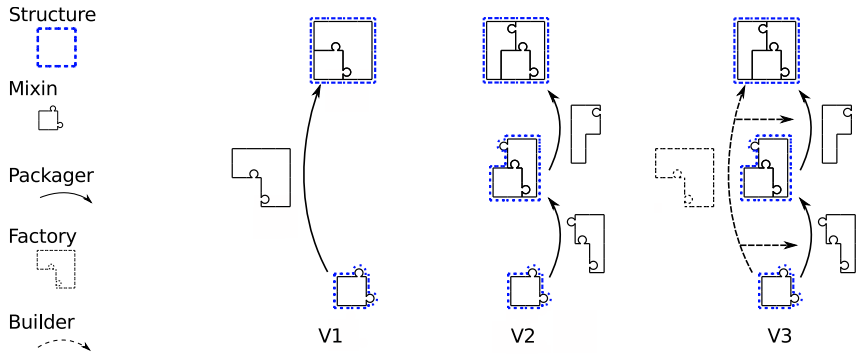

Figure 1 The evolution of the hierarchy. V3 is backward compatible with V1, while V2 is not.

As depicted in figure 1 we change again the hierarchy by declaring a `Ring_of_Monoid` factory, that, from the user point of view, will look indistinguishable from the old `Ring_of_Monoid` mixin and hence grant backward compatibility between version V3 and version V1.

```
1  HB.factory Record Ring_of_Monoid A of Monoid A := {
2    ... (* unchanged, see page 2 line 13 *) ...
3  }.
4  HB.builders Context A (f : Ring_of_Monoid A).
5
6    Lemma addrC : commutative add. Proof. (* The same proof as before *) Qed.
7
8    Definition to_AbelianGroup_of_Monoid :=
9      AbelianGroup_of_Monoid.Build A (opp f) addrC (addNr f). (* addrN unused *)
10   HB.instance A to_AbelianGroup_of_Monoid.
11
12   Definition to_Ring_of_AbelianGroup :=
13     Ring_of_AbelianGroup.Build A (one f) (mul f)
14       (mulrA f) (mul1r f) (mulr1 f) (mulrDl f) (mulrDr f).
15   HB.instance A to_Ring_of_AbelianGroup.
16
17 HB.end.
```

The record `Ring_of_Monoid` is the same we declared as a mixin in version V1. In order to make a factory out of it we equip it with two definitions that embody the *builders*. The first is `to_AbelianGroup_of_Monoid` and explains how to build an `AbelianGroup` structure out of the factory axioms (named `f`, line 4). This construction is also registered as canonical for `A`, so that the next construction `to_Ring_of_AbelianGroup` can call the `Ring_of_AbelianGroup.Build` abbreviation that requires `A` to be an `AbelianGroup`. It is worth pointing out that the proof of `addrC` we had in V1 is now required in order to write the builder for Abelian groups, while the `addrN` field is not used (the same statement is already part of the theory of Abelian groups, see line 20 of the previous code snippet).

Thanks to this factory we can now declare `Z` to be an instance of a ring using the `Ring_of_Monoid.Build` abbreviation. The associated builders generate, behind the scenes, instances of the `Ring_of_AbelianGroup` and `AbelianGroup_of_Monoid` mixins that in turn are used to build instances of the `AbelianGroup` and `Ring` structures. Indeed, when used in the context of the hierarchy version V3, the command `HB.instance Z Z_Ring_axioms` makes `Z` an instance of *both* structures, and not just the `Ring` one as in version V1. Thanks to that the proof of `example` can use the theory of both structures, for example `addrC` holds on Abelian groups, `addrA` holds on monoids, while `addr1` holds on rings. As a result the very same proof works on both version V1 and V3.

Last, it is worth pointing out that the new factory makes the following two lines equivalent (the former declares rings in V1) since they both describe the same set of mixins.

```
HB.structure Definition Ring := { A of Monoid A & Ring_of_Monoid A }.
HB.structure Definition Ring := { A of AbelianGroup A & Ring_of_AbelianGroup A }.
```

This is another example of client code that would not break: the client of the hierarchy is allowed to declare new structures on top of existing ones.

2.3 $\mathcal{HB}$ in a nutshell

By using $\mathcal{HB}$ the hierarchy designer has the following freedoms and advantages.

- Operations and properties (axioms) are made available to the user as soon as they are used in a structure. The hierarchy designer is free to move them from one to another and replace an axiom by a lemma and vice versa.
- Structures cannot disappear but the way they are built may change. The hierarchy designer is free to split structures into smaller ones in order to better factor and reuse parts of the hierarchy and the library that follows it.
- Mixins cannot disappear but can change considerably in their implementation. A mixin can become a full fledged factory equipped with builders to ensure backward compatibility.
- $\mathcal{HB}$ high level commands compile to the discipline of Packed Classes (Coq modules, records, coercions, implicit arguments, canonical structure instances, notations, abbreviations). This process lifts a considerable burden from the shoulders of the hierarchy designer and the final user who are no longer required to master the details of Packed Classes.

3 $\mathcal{HB}$ language

The Coq terms handled by $\mathcal{HB}$ are subdivided in five categories, the mixins $\mathcal{M}$, the factories $\mathcal{F}$, builders $\mathcal{B}$, the classes $\mathcal{C}$ and the structures $\mathcal{S}$. Mixins, factories and instances are tagged by the user, through the commands `HB.mixin`, `HB.factory` and `HB.instance`, and the user may rely on their implementation. However structures and classes are generated with the command `HB.structure` and builders are generated when using `HB.instance` while declaring a factory, and the user may only refer to structure types, but should never rely on their implementation, neither should they rely on explicit builders.

3.1 $\mathcal{HB}$ mixins, factories and instances

▶ **Definition 1** ($\mathcal{M}$, mixins). *A* mixin $m \in \mathcal{M}$ *is a Coq record with one or more parameters. The first parameter must be a* `(T : Type)`*, while the other parameters are mixins applied to* `T` *and possibly* n *other distinct mixins. I.e.*

```
Record m (T : Type) (p : m T p_σ)^n : Type := { .. }.
```

where $\overrightarrow{(p : m\ T\ p_\sigma)}^n = \overrightarrow{(p_i : m_i\ T\ p_{\sigma(i,1)}\ \cdots\ p_{\sigma(i,n_i)})}_{i\in\{1,\ldots,n\}}$. *For all* $i \in \{1,\ldots,n\}$ *we have* $m_i \in \mathcal{M}$ *and its arguments consist in* n_i *of the previously quantified mixin parameters* p_k*, i.e. for all* $k \in \{1,\ldots,n_i\}$*, we have* $\sigma(i,k) \in \{1,\ldots,i-1\}$.

▶ **Definition 2** (*dep*, mixin dependencies). *Given* $m \in \mathcal{M}$*, we define* $dep(m) \in \wp(\mathcal{M})$ *as the set of all mixins that occur as parameters of* m*, i.e.* $dep(m) = \{m_1,\ldots,m_n\}$.

▶ Remark 3. For all $i \in \{1,\ldots,n\}$, we have $dep(m_i) = \{m_{\sigma(i,1)},\ldots,m_{\sigma(i,m_i)}\}$.

▶ **Definition 4.** *If $f : \mathcal{X} \to \wp(\mathcal{Y})$, then $f^\star : \wp(\mathcal{X}) \to \wp(\mathcal{Y})$ is defined as $f^\star(X) = \bigcup_{x \in X} f(x)$.*

▶ **Proposition 5.** *dep is transitively closed since records are well typed in the empty context and describes a DAG since Coq does not admit circular definitions: $dep^\star(dep^\star(M)) \subseteq dep^\star(M)$*

▶ **Definition 6** (factories $\mathcal{F}$, builders $\mathcal{B}$ and from)**.** *A* builder $\mu \in \mathcal{B}$ *is any function which parameters are the carrier type* `(T: Type)`*, an arbitrary number of distinct mixins $\{m_1, \ldots, m_n\}$ and an additional argument f, which we call a* factory*, and which depends on the mixins. Its return type is a mixin $m_{n+1} \in \mathcal{M}$ i.e.*

```
Definition f (T : Type) (p : m T p_σ)^n : Type := ...
Definition μ (T : Type) (p : m T p_σ)^n : f T p_1 ... p_n -> m_{n+1} T p_{σ(n+1,1)} ... p_{σ(n+1,n_i)} := ...
```

We define $\mathsf{from}(f, m_{n+1}) = \mu$ *to be the (unique) builder for m_{n+1} from the factory f.*

▶ **Definition 7** (*provides, requires*)**.** *We define the following functions*

- *$requires(f) = \{m_1, \ldots, m_n\} \in \wp(\mathcal{M})$ is the set of mixins a factory $f \in \mathcal{F}$ depends on. As a consequence all the builders of a given factory have the same set of dependencies.*
- *$provides(f) = \{m | \mathsf{from}(f, m)$ is defined$\} \in \wp(\mathcal{M})$ the set of mixins that a factory $f \in \mathcal{F}$ provides, through its builders.*

Moreover the following properties hold for all $f \in \mathcal{F}$:

1. *$dep^\star(requires(f)) \subseteq requires(f)$*
2. *$requires(f) \cap provides(f) = \emptyset$*
3. *$\forall m \in provides(f)$, $\mathsf{from}(f, m)$ is defined*

Mixins are declared by the user as the fundamental building blocks of a hierarchy. As the next proposition shows they shall not be distinguished from regular factories, since they are a special cases.

▶ **Proposition 8** ($\mathcal{M} \subseteq \mathcal{F}$)**.** *For all $m \in \mathcal{M}$ we have $requires(m) = dep(m)$, $provides(m) = \{m\}$ and $\mathsf{from}(m, m) = \left(\texttt{fun T } (p : m\ T\ p_\sigma)^n \texttt{ (x : m T } p_1 \ldots p_n \texttt{) => x}\right) \in \mathcal{B}$.*

▶ **Coq command** to declare a new mixin:

```
HB.mixin Record M T of f_1 ... f_n := { .. }.
```

Creates a module `M` with a record `M`, which depends on the mixins $requires(\texttt{M}) = provides^\star(\{f_1, \ldots, f_n\})$, and registers this new record both as a mixin and a factory. It also creates a abbreviation `M.Build`.

▶ **Coq command** to declare an instance: `HB.instance` $X\ b_1 \ldots b_k$, synthesizes terms corresponding to all the mixins that can be built from the b_i. Indeed if $b_i : f_i\ T\ \ldots$, then this command creates elements of type $provides^\star(\{f_1, \ldots, f_k\})$. This command also generates unification hints as described in Section 4.

▶ **Coq command** to declare a new factory and generate new builders:

```
HB.factory Record M T of f_1 ... f_n := { .. }.
HB.builders Context T (a : M A).

  Definition b_{n+1} : f_{n+1} T .. := ...
  HB.instance T b_{n+1}.
  ..
  Definition b_{n+k} : f_{n+k} T .. := ...
  HB.instance T b_{n+k}.
HB.end.
```

Creates a module `M` with a record `M`, which depends on the mixins $\mathit{requires}(\mathtt{M}) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and registers this new record as a factory. We then use the factory instances $b_{n+1}, \ldots, b_{n+k}$ provided by the user and derive builders from them, so that $\mathit{provides}(\mathtt{M}) = \mathit{provides}^\star(\{f_{n+1}, \ldots, f_{n+k}\})$.

It is thus necessary that $\mathit{requires}^\star(\{f_{n+1}, \ldots, f_{n+k}\}) \subseteq \mathit{provides}^\star(\{f_1, \ldots, f_n\})$.

Note that the b_i are not builders since their return types are not necessarily mixins, but could be a factory. However, since all factories provide mixin through builders, we can obtain builders out of each b_i by function composition.

3.2 $\mathcal{HB}$ classes and structures

▶ **Definition 9** ($\mathcal{C}$, class). *A class* $\mathtt{c} \in \mathcal{C}$ *is a Coq record with one parameter* `(T : Type)`*. The type of each field is a mixin in* $\mathcal{M}$ *applied to* `T` *and, if needed, any number of other fields:*

```
Record c (T : Type) := { p_1 : m_1 T; .. ; p_i : m_i T p_σ(i,1) ... p_σ(i,n_i); ..}.
```

▶ **Definition 10** (*def*, class definition). *We call* $\mathit{def}(c) \in \wp(\mathcal{M})$ *the set of mixins mentioned in the fields of the class, i.e.* $\{m_1, \ldots, m_n\}$*. Given that class records are well typed in the empty context the set of mixin records is closed transitively. The implementation enforces that no two class records contains the same set of mixins (disregarding the order of the fields), i.e. def is injective.*

▶ **Programming invariant for** *def* For all $f \in \mathcal{F}$:

1. $\exists c \in \mathcal{C},\ \mathit{def}(c) = \mathit{requires}^\star(f) \cup \mathit{provides}^\star(f)$,
2. $\exists C \subseteq \mathcal{C},\ \mathit{def}^\star(C) = \mathit{requires}^\star(f)$.

Classes could also be seen as factories.

▶ **Proposition 11** ($\mathcal{C} \subseteq \mathcal{F}$). *forall* $m \in \mathit{def}(c)$ *we have* $\mathsf{from}(c, m_i) = p_i$*, and it follows that* $\mathit{requires}(c) = \emptyset$*,* $\mathit{provides}(c) = \mathit{def}(c)$.

However since classes are generated, their implementation may change, thus users should not rely on constructors of classes. Hence the only way users may refer to a class is as an argument of `HB.mixin`, `HB.factory` or `HB.structure`.

▶ **Definition 12** ($\mathcal{S}$ Structure). *A structure* $\mathtt{s} \in \mathcal{S}$ *is a dependent pair: a Coq record where the value of the first field occurs in the type of the second. The first field is* `(sort : Type)` *and the second field is* `(class : c sort)` *for some* $\mathtt{c} \in \mathcal{C}$*. As a consequence structures are in bijection with classes.*

▶ **Coq command** to declare a class and structure: `HB.structure Definition M := { A of` $f_1 \ldots f_n$ `}.` crafts a class `M.class_of` $\in \mathcal{C}$ where $\mathit{def}(c) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and the corresponding structure `M.type` $\in \mathcal{S}$, together with unification hints as described in Section 4.

▶ **Definition 13** ($\leq \in \mathcal{C} \times \mathcal{C}$, subclass). $c1 \leq c2$ *iff* $\mathit{def}(c2) \subseteq \mathit{def}(c1)$

3.3 Automatic inference of mixins

Since mixins may change but factories stay the same, $\mathcal{HB}$ commands must never rely on a particular set of mixins as arguments, and factory arguments must never be given explicitly by the user. As described in Sections 3.1 and 3.2, $\mathcal{HB}$ commands take a list of factories as arguments, which they expand into lists of mixins behind the scene. However factory types and constructors have mixin arguments that must be inferred automatically when used. To this end, the commands `HB.mixin` and `HB.factory` generate abbreviations for the user to

replace uses of constructors of factories. These commands first create a record f_{aux} with a constructor F_{aux} and then create abbreviations f and F that automatically fill the mixin arguments of f_{aux} and F_{aux} respectively. See [17, Section 7] for a detailed description of how to implement these abbreviations in Coq.

```
Record f_aux T (p : m T p_σ)^n := F_aux { ... }.
Notation f T := (f_aux T ...(* p_1 ... p_n inferred when T is known *)).
Notation F T  x_1 ... x_k := (F_aux T ...(* p_1 ... p_n inferred when T is known *) x_1 ... x_k).
Definition b : f T := F T x_1 ... x_n
```

4 The target language: Coq with Packed, flat, Classes

The language of packed classes were introduced in [13] and used directly to describe the algebraic hierarchy of the Mathematical Components library. It is based on a disciplined use of records and projections and on the possibility of extending the elaborator of Coq via the declaration of Canonical Structures instances. In this section we describe the *flat* variant of Packed Classes, the target language of $\mathcal{HB}$.

4.1 Describing structures with records and projections

We describe mathematical structures with three kinds of dependent records: mixins, classes, and structures. As shown in section 2, a mixin gathers operators and axioms newly introduced by a structure. As in [13, section 2.4][26, section 2], a class record is parametrized by the carrier type (`T : Type`) and gathers all the operators and axioms of a structure by assembling mixins, and a `Structure type` (a record) bundles a carrier and its class instance, as follows.

```
Module Monoid.
Record axioms (T : Type) : Type :=
  Class { Monoid_of_Type_mixin : Monoid_of_Type T; }.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End Monoid.
```

The `Monoid` module plays the role of a name space and forces us to write qualified names such as `Monoid.type`; as a consequence we can reuse the same unqualified names for other structures, i.e., class and structure record can always be named as `axioms` and `type` respectively.

Mixins and classes are internal definitions to structures; in contrast, `Monoid.type` is part of the interface of the monoid structure. As seen in section 2, we declare `Monoid.sort` as an implicit coercion and lift monoid operators and axioms from projections for the monoid mixin to definitions and lemmas for `Monoid.type` as follows.

```
Coercion Monoid.sort : Monoid.type >-> Sortclass.

Definition zero {T : Monoid.type} : T :=
  Monoid_of_Type.zero T (Monoid.Monoid_of_Type_mixin T (Monoid.class T)).
Definition add {T : Monoid.type} : T -> T -> T :=
  Monoid_of_Type.add T (Monoid.Monoid_of_Type_mixin T (Monoid.class T)).
Lemma addrA {T : Monoid.type} : associative (@add T).
(* Two monoid axioms `add0r` and `addr0` are omitted. *)
```

Next we define Abelian groups. Since the monoid structure is the bottom of this hierarchy its class record `Monoid` consists just one mixin. In contrast the class record of Abelian groups consists of two mixins where the second one depends on the former (since Abelian groups inherit from monoids).

```
Module AbelianGroup.
Record axioms (T : Type) : Type := Class {
```

```
  Monoid_of_Type_mixin : Monoid_of_Type T;
  AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid T Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort }.
End AbelianGroup.
```

As in the monoid structure, we declare `AbelianGroup.sort` as an implicit coercion and lift additive inverse `opp` and Abelian group axioms. Since Abelian groups inherit from monoids, we also declare an implicit coercion from Abelian groups to monoids. A coercion between structures can be defined in two steps: first a coercion between the class record of the superclass to the class record of the subclass (line 2); and a coercion between structure records (line 4) relying on the first one.

```
Coercion AbelianGroup.sort : AbelianGroup.type >-> Sortclass.
Coercion AbelianGroup_class_to_Monoid_class (T : Type) (c : AbelianGroup T) :
  Monoid T := Monoid.Class T (AbelianGroup.Monoid_of_Type_mixin T c).
Coercion AbelianGroup_to_Monoid (T : AbelianGroup.type) : Monoid.type :=
  Monoid.Pack (AbelianGroup.sort T) (AbelianGroup.class T).

Definition opp {T : AbelianGroup.type} : T -> T := ... .
(* Two Abelian group axioms `addrC` and `addNr` are omitted. *)
```

Generally, a coercion from a structure A to another structure B can (and should) have the following form, thanks to the corresponding coercion between classes `A_class_to_B_class`.

```
Coercion A_to_B (T : A.type) : B.type :=
  B.Pack (A.sort T) ((* A_class_to_B_class _ *) (A.class T)).
```

4.2 Multiple inheritance

In order to introduce multiple inheritance, we extend the hierarchy described in section 2.2 with the structure of semiringsas depicted in figure 2. Semirings introduce the binary multiplication operator `mul` and its identity element `one`.

```
HB.mixin Record SemiRing_of_Monoid A
    of Monoid A := {
  one : A;
  mul : A -> A -> A;
  (* 7 axioms are omitted. *)
}.

HB.factory Record Ring_of_AbelianGroup A
    of AbelianGroup A := {
  (* 2 operators and 5 axioms are omitted. *)
}.
HB.builders Context A (a : Ring_of_AbelianGroup A).
..
HB.end.
```

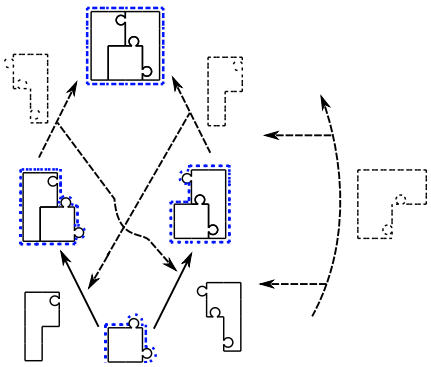

Figure 2 V4 introduces multiple inheritance. For brevity we omit the factories/builders for the upper arrows.

Since semirings and Abelian groups do not inherit from each other, the definition of semirings and Abelian groups are quite similar:

```
Module SemiRing.
Record axioms (T : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type T;
  SemiRing_of_Monoid_mixin : SemiRing_of_Monoid T Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End SemiRing.
```

We define implicit coercions from the semiring structure to the carrier and the monoid structure, and we lift semiring operators and axioms as follows:

```
Coercion SemiRing.sort : SemiRing.type >-> Sortclass.
Coercion SemiRing_to_Monoid : SemiRing.type >-> Monoid.type.

Definition one {T : SemiRing.type} : T := ... .
Definition mul {T : SemiRing.type} : T -> T -> T := ... .
(* 7 semiring axioms are omitted. *)
```

The class record is defined by gathering the monoid, Abelian group, and semiring mixins. Since the rings inherit from monoids, semirings, and Abelian groups, we define implicit coercions from the ring structure to those three structures.

```
Module Ring.
Record axioms (T : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type T;
  SemiRing_of_Monoid_mixin : SemiRing_of_Monoid T Monoid_of_Type_mixin;
  AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid T Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End Ring.

Coercion Ring.sort : Ring.type >-> Sortclass.
Coercion Ring_to_Monoid : Ring.type >-> Monoid.type.
Coercion Ring_to_SemiRing : Ring.type >-> SemiRing.type.
Coercion Ring_to_AbelianGroup : Ring.type >-> AbelianGroup.type.
```

4.3 Linking structures and instances via Canonical Structures

We recall here how to use the Canonical Structures [17, 25] mechanism, which lets the user extend the elaborator of Coq, in order to handle inheritance and inference of structure [1, 26, 13]. This software component takes in input a term as written by the user and has to infer all the missing information that is necessary in order to make the term well typed.

A first example of elaboration that requires canonical structures is $0 + 1$. After removing all syntactic facilities, the underlying Coq term is `(add _ (zero _)) (one _)` where `_` stands for an implicit piece of information to be inferred and the constants `add` and `zero` come from the monoid structure while `one` comes from semirings. When the term is type checked each `_` is replaced by a unification variable written $?_v$. Respectively, the head and the argument of the top application can be typed as follows:

```
  add ?M (zero ?M) : Monoid.sort ?M -> Monoid.sort ?M
  one ?SR : SemiRing.sort ?SR
```

where $?_M$: `Monoid.type` and $?_{SR}$: `SemiRing.type`. In order to type check the application Coq has to unify `Monoid.sort` $?_M$ with `SemiRing.sort` $?_{SR}$, which is not trivial: it amounts at finding a structure that is both a monoid and a semiring, possibly the smallest one [26, Sect. 4]. This piece of information can be inferred from the hierarchy and its inheritance relation (Definition 13) and we can tell Coq to exploit it by declaring `SemiRing_to_Monoid : SemiRing.type -> Monoid.type` as canonical. With that hint Coq will pick $?_M$ to be `SemiRing_to_Monoid` $?_{SR}$ as the canonical solution for this unification problem. In general, all the coercions between structures must be declared as canonical.

Another example of elaboration problem is -1, which hides the term `opp _ (one _)`. Here `opp` and `one` are respectively from Abelian groups and semirings, which do not inherit each other but whose smallest common substructure is rings; thus we have to extend the unifier to solve a unification problem `AbelianGroup.sort` $?_{AbG}$ = `SemiRing.sort` $?_{SR}$ by instantiating $?_{AbG}$ and $?_{SR}$ with `Ring_to_AbelianGroup` $?_R$ and `Ring_to_SemiRing` $?_R$ respectively where $?_R$ is a fresh unification variable of type `Ring.type`. This hint can be given as follows:

```
Canonical AbelianGroup_to_SemiRing (T : Ring.type) :=
  SemiRing.Pack (AbelianGroup.sort (Ring_to_AbelianGroup T)) (Ring.class T)).
```

Similarly, one can apply an algebraic theory to an instance (an example) of that structure, e.g., 2×3 where 2 and 3 have type $\mathbb{Z}$. The same mechanism of canonical structures let us extend the unifier to solve `SemiRing.type` $? = \mathbb{Z}$.

5 The implementation in Coq-Elpi

The implementation is based on the Elpi extension language for Coq. In this section we introduce the features of the programming language that came in handy in the development of $\mathcal{HB}$ and comment a few code snippets.

Coq-Elpi [28] is a Coq plugin embedding Elpi and providing an extensive, high level, API to access and script the Coq system at the vernacular level. This API lives in the `coq` namespace and lets one easily declare records, coercions, canonical structures, modules, implicit arguments, etc. The most basic Coq data type exposed to Elpi is the one of references to global declarations:

```
kind gref  type.                % The data type of references to global terms
type indt  inductive -> gref.   % eg: Coq.Init.Datatypes.nat
type indc  constructor -> gref. % eg: Coq.Init.Datatypes.O
type const constant -> gref.    % eg: Coq.Init.Peano.plus
```

The arguments of the three constructors are opaque to Elpi, that can only use values of these types via dedicated APIs. For example the API for declaring an inductive type will generate a value of type `inductive` that is printed as, for example, `«nat»`.

Elpi [11] is a dialect of λProlog [19], an higher order logic programming language that makes it easy to manipulate abstract syntax tree with binders. Coq-Elpi takes full advantage of this capability by representing Coq terms in Higher Order Abstract Syntax [20] style, reusing the binder of the programming language in order to represent Coq's ones. Here is an excerpt of the data type of Coq terms:

```
kind term type.                          % The data type of Coq terms
type global gref -> term.                % eg: nat, O, S, plus, ...
type fun    term -> (term -> term) -> term. % eg: fun x : t => b(x)
type app    list term -> term.           % eg: app [hd|args]
... % all other term constructors are omitted for brevity
```

Note that the `fun` constructor holds a λProlog function. In this syntax the Coq term `(fun x : nat => x x)` becomes `(fun (global (indt «nat»)) x\ app[x, x])` where `x\` binds `x` in the body of the function. Substitution of a bound variable for a term can be computed by applying a term (of function type) to an argument.

Data types with binders are also used as input to high level APIs that build terms behind the scenes. For example a Coq record is just an inductive type and the API to declare one must allow the type of a field to depend on the fields that comes before it. Note that the `field` constructor takes a coercion flag, the name of the field, its type and binds a term in the remaining record declaration.

```
kind indt-decl type.              % The type of an inductive type declaration
type record      string -> term -> string -> record-decl -> indt-decl.
type field       bool -> string -> term -> (term -> record-decl) -> record-decl.
type end-record  record-decl.
... % constructors for non-record inductive types are omitted for brevity

external pred coq.env.add-indt i:indt-decl, o:inductive.
external pred coq.CS.canonical-projections i:inductive, o:list (option constant).
```

The `pred` keyword documents types and modes (input or output) of the arguments of a predicate, while `external` signals that the predicate is a builtin (in other words it is implemented in OCaml rather than λProlog).

We comment these two builtin predicates and the `indt-decl` type while looking at the code of `declare-structure` that is in charge of scripting the following Coq code:

```
Structure type : Type := Pack { sort : Type; class : axioms sort }.
```

The following Elpi code builds the declaration, type checks it, adds it to the Coq environment and finally returns the projections for the sort and the class fields.

```
pred declare-structure i:gref, o:term, o:term, o:term.

declare-structure ClassName Structure SortProjection ClassProjection :- std.do! [
  StructureDeclaration =
    record "type" {{ Type }} "Pack" (
      field tt "sort" {{ Type }} s\
      field ff "class" (app [global ClassName, s]) _\
    end-record),
  coq.typecheck-indt-decl StructureDeclaration,
  coq.env.add-indt StructureDeclaration StructureName,
  coq.CS.canonical-projections StructureName [some SortP, some ClassP],
  Structure = global (indt StructureName),
  SortProjection = global (const SortP),
  ClassProjection = global (const ClassP),
].
```

Note that the binder `s\` at line 6 lets one mention the first field in the type of the second. The syntax `{{ Type }}` is a quotation: it lets one use the syntax of Coq to write an Elpi expression of type `term`. The API `coq.CS.canonical-projections` lets us find the projections automatically generated by Coq for a given record. The last detail worth mentioning is that this program makes no use of backtraking: the `std.do!` combinator signals that.

In the simple case of the structure record, the number of fields, and hence the number of binders, is fixed. On the contrary the class record has one field per mixin and each of them can depend on the previous ones. In order to synthesize terms with binders in an inductive fashion (using a recursive predicate) λProlog lets one postulate fresh nominal constants using the `pi` operator and attach to the nominal some knowledge in the form of a clause via the `=>` operator. This process is called binder mobility: the binder is moved from the data (that we are building) to the program (the context of the current computation). This feature is key to the following code that synthesizes the declaration of the fields of the class record.

```
pred synthesize-fields.field-for-mixin i:mixinname, o:term.
pred synthesize-fields i:list mixinname, i:term, o:record-decl.

synthesize-fields [] _ Decl :- Decl = end-record.
synthesize-fields [M|ML] T Decl :- std.do! [
  get-mixin-modname M ModName, Name is ModName ^ "_mixin",
  dep1 M Deps,
  std.map Deps synthesize-fields.field-for-mixin Args,
  Type = app[ global M, T | Args ],
  Decl = (field ff Name Type f\ Fields f),
  pi m\
    synthesize-fields.field-for-mixin M m =>
    synthesize-fields ML T (Fields m)
].
```

The first predicate `synthesize-fields.field-for-mixin` is used to link a mixin to a nominal that corresponds to the record field for that mixin. It has no clauses in the base program but some clauses are added dynamically by `synthesize-fields`.

The second predicate proceeds by recursion on the (topologically sorted) list of mixins, and terminates when the list is empty. If the list contains a mixin `M` then it crafts a `Name` for it (line 6), fetches its dependencies (line 7) and finds the (previously declared) record fields holding these mixins (line 8). The `(std.map L1 P L2)` predicate relates the two lists `L1` and `L2` point wise using the predicate `P`.

Line 9 builds the type of the field: the mixin name applied to the type (sort) and all its dependencies. Note that the `Fields` variable, representing the declaration of the next fields, is under the binder for `f` (the current field). In order to make the recursive call under that binder (line 13) and recursively process `ML` we postulate a nominal `m` (line 11) that is a term satisfying any future dependency on the current mixin (line 12) and we replace `f` by `m` in `Fields` by writing `(Fields m)`.

6 Related work

The most closely related work is the one about Packed Classes [13] on which we build. The main differences are that $\mathcal{HB}$ is a higher level language that is compiled down to (flat) Packed Classes. The systematic use of factories makes the user interface of a hierarchy stable under the insertion of structures, a property that Packed Classes lacks. Finally many of the intricacies of Packed Classes are hidden to the user by the compilation step, in particular creating all the necessary coercions and canonical structure declarations, especially in the case of diamonds or when merging several libraries, which used to cause the need for *a posteriori* validation of a hierarchy design [26]. It also opens the way to automatically detect and solve problems tied to non judgmental commutative diagrams when forgetting structure [1].

In [7] Carette and O'Connor describe the language of Theory Presentation Combinators that can be used to describe a hierarchy of algebraic structures. They focus on the categorical semantics of the language that is built upon the category of context. They do not describe any actual compilation to the language of a mainstream interactive prover, indeed they claim their language to be mostly type theory independent. We know they considered targeting type theory and the language of unification hints [4] (a super set of the one of Canonical Structures), but we could not find any written trace of that. Language wise they provides keywords such as `combine` and `over` to share (reuse) a property when defining a new structure. For example in order to avoid restating the commutativity property when defining Abelian groups they combine a commutative monoid and a group forcing the subjacent monoid to coincide: `CommutativeGroup := combine CommutativeMonoid , Group over Monoid.` In our language $\mathcal{HB}$ the same role is played by *mixins*. A mixin lets one write once and for all a property and abstract it over types and operations so that it can be reused in all the structures that need it. One operation $\mathcal{HB}$ allows for but that does not seem to be possible in the setting of Presentation Combinators is the one of replacing an axioms with a lemma and vice versa. As shown in subsection 2.1 $\mathcal{HB}$ supports that.

The MMT system [22] provides a framework to describe formal languages in a logical framework, providing good support for binders and notations. It also provides an expressive module system to organize theories and express relations among them in the form of functors. At the time of writing it provides limited support for elaborating user input taking systematic advantage of the contents of the theories. The elaborator can be extended by the user writing Scala code, and in principle use the contents of the libraries to make sense of an incomplete expressions, but no higher level language or mechanism is provided.

The library of the Mizar system features a hierarchy of algebraic structures [15]. In spite of lacking dependent types, Mizar provides the concept of attributed types and adjectives

that can be used to describe the signature of structures as one would do with a dependently typed record and their properties as one would define a conjunctive predicate. The Mizar language also provides the notion of cluster that is used to link structures: by showing that property P implies property Q one can inform automation that structures characterized by P are instances of structures characterized by Q. The foundational theory of Mizar features an extensional notion of equality that makes it easy to share the signature or the properties of structures by just requiring a proof of their equivalence that is in turn used by automation to treat equivalent structures as equal.

The concepts of factory and builder presented here is akin to the `AbstractFactory` and `Builder` pattern from "the Gang Of Four" design patterns [12] in the sense that factories are used to build an arbitrary number of objects (here mixin instances).

7 Conclusion

In this paper we design and implement $\mathcal{HB}$, a high level language to describe hierarchies of algebraic structures. The implementation of $\mathcal{HB}$ is based on the Elpi extension language for the Coq system and is available at `https://github.com/math-comp/hierarchy-builder`. The implementation amounts to approximately a thousand lines of (commented) Elpi code and tree hundred lines of Coq vernacular. It took less than one month to implement $\mathcal{HB}$ while it took several years of attempts and fruitful discussions with the other developers of the Mathematical Components library to design it, as well as Coq users and developers and members of Dagstuhl seminar 18341. In particular, discussions with Georges Gonthier, Assia Mahboubi and Florent Hivert were instrumental in coining the concepts formalized here.

The $\mathcal{HB}$ language is only loosely tied to Coq or even Type Theory. We believe it could be adopted with no major change to other tools. Indeed the properties and invariants that link factories, mixins and classes are key to rule out meaningless or ambiguous sentences and are not specific to our logic setting. Moreover most logics feature packing construction similar to records.

The compilation scheme we present in section 4 is tied to dependent records, that are available in Coq but also in other provers based on Type Theory such a Matita [3] and Lean [10]. We chose the flat variant of Packed Classes as the target for $\mathcal{HB}$ because the Mathematical Components library uses Packed Classes as well: As of today Packed Classes offer the best compromise between flexibility and performance [17, Section 8] in Coq. Of course one could imagine a future were other approaches such as telescopes [21] or unbundled classes [27] would offer equal or better performances in Coq, or in another system. It is a virtue of our work to provide a user language that is separate from the implementation one. We believe it would take a minor coding effort to retarget $\mathcal{HB}$ to another bundling approach.

We leave to future work extending $\mathcal{HB}$ to support structures parametrized by structures such as the one of module over a ring. We also leave to future work the automatic synthesis of the notion of morphism between structures of the hierarchy.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Formalizing functional analysis structures in dependent type theory. working paper or preprint, January 2020. URL: `https://hal.inria.fr/hal-02463336`.

2 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*,

pages 226–254, 2019. URL: https://doi.org/10.1007/978-3-030-33636-3_9, doi:10.1007/978-3-030-33636-3_9.

3 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 64–69, 2011. URL: https://doi.org/10.1007/978-3-642-22438-6_7, doi:10.1007/978-3-642-22438-6_7.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 84–98, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

5 Gerhard Betsch. On the beginnings and development of near-ring theory. In *Near-rings and near-fields. Proceedings of the conference held in Fredericton, New Brunswick, July 18-24, 1993*, pages 1–11. Mathematics and its Applications, 336. Kluwer Academic Publishers Group, Dordrecht, 1995.

6 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. URL: https://doi.org/10.1007/s11786-014-0181-1, doi:10.1007/s11786-014-0181-1.

7 Jacques Carette and Russell O'Connor. Theory presentation combinators. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 202–215, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

8 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* PhD thesis, Ecole Polytechnique X, 2012. URL: https://pastel.archives-ouvertes.fr/pastel-00780446.

9 The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.11*, October 2020. URL: http://coq.inria.fr.

10 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015. URL: https://doi.org/10.1007/978-3-319-21401-6_26, doi:10.1007/978-3-319-21401-6_26.

11 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 460–468, 2015. URL: https://doi.org/10.1007/978-3-662-48899-7_32, doi:10.1007/978-3-662-48899-7_32.

12 Erich Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

13 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 327–342, 2009. URL: https://doi.org/10.1007/978-3-642-03359-9_23, doi:10.1007/978-3-642-03359-9_23.

14 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013. URL: https://doi.org/10.1007/978-3-642-39634-2_14, doi:10.1007/978-3-642-39634-2_14.

15 Adam Grabowski, Artur Korniłowicz, and Christoph Schwarzweller. On algebraic hierarchies in mathematical repository of Mizar. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 363–371, Sep. 2016.

16 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 279–294. Springer, 2013. URL: https://doi.org/10.1007/978-3-642-39634-2_21, doi:10.1007/978-3-642-39634-2_21.

17 Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, 2013. URL: https://doi.org/10.1007/978-3-642-39634-2_5, doi:10.1007/978-3-642-39634-2_5.

18 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367—381, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3372885.3373824, doi:10.1145/3372885.3373824.

19 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

20 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery. URL: https://doi.org/10.1145/53990.54010, doi:10.1145/53990.54010.

21 Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002. doi:10.1007/s001650200018.

22 Florian Rabe and Michael Kohlhase. A scalable module system. *Information and Computation*, 230:1–54, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0890540113000631, doi:https://doi.org/10.1016/j.ic.2013.06.001.

23 Damien Rouhling. *Formalisation Tools for Classical Analysis – A Case Study in Control Theory*. PhD thesis, Université Côte d'Azur, 2019. URL: https://hal.inria.fr/tel-02333396.

24 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997), Paris, France, 15–17 January 1997*, pages 292–301. ACM, 1997.

25 Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 1999. URL: https://tel.archives-ouvertes.fr/tel-00523810.

26 Kazuhiko Sakaguchi. Validating mathematical structures, 2020. arXiv:2002.00620.

27 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

28 Enrico Tassi. Coq-Elpi, Coq plugin embedding Elpi. https://github.com/LPCIC/coq-elpi, 2020.

A Coq reference

```
Section OperationProperties.
Variable T : Type.
Variable e : T.
Variable inv : T -> T.
Variable op : T -> T -> T.
Variable add : T -> T -> T.

Definition left_id  := forall x, op e x = x.
Definition right_id := forall x, op x e = x.

Definition left_inverse := forall x, op (inv x) x = e.

Definition commutative := forall x y, op x y = op y x.
Definition associative := forall x y z, op x (op y z) = op (op x y) z.

Definition left_distributive  := forall x y z, op (add x y) z = add (op x z) (op y z).
Definition right_distributive := forall x y z, op x (add y z) = add (op x y) (op x z).
```

B Proof of addrC

```
Lemma addrC {R : Ring.type} : commutative (@add R).
Proof.
have innerC (a b : R) : (a + b) + (a + b) = (a + a) + (b + b).
  by rewrite -[a+b]mul1r -mulrDl mulrDr !mulrDl !mul1r.
have addKl (a b c : R) : a + b = a + c -> b = c.
  apply: can_inj (add a) (add (-a)) _ _ _.
  by move=> x; rewrite addrA addNr add0r.
have addKr (a b c : R) : b + a = c + a -> b = c.
  apply: can_inj (add ^~ a) (add ^~ (-a)) _ _ _.
  by move=> x; rewrite /= -addrA addrN addr0.
move=> x y; apply: addKl (x) _ _ _; apply: addKr (y) _ _ _.
by rewrite -!addrA [in RHS]addrA innerC !addrA.
Qed.
```