Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi

Cyril Cohen, Kazuhiko Sakaguchi, Enrico Tassi

To cite this version:

Cyril Cohen, Kazuhiko Sakaguchi, Enrico Tassi. Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi. FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction, Jun 2020, Paris, France. pp.34:1–34:21, 10.4230/LIPIcs.FSCD.2020.34 . hal-02478907v5

HAL Id: hal-02478907
https://inria.hal.science/hal-02478907v5

Submitted on 6 May 2020 (v5), last revised 23 Sep 2022 (v6)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 Software and its engineering → Formal language definitions; Theory of computation → Type theory; Theory of computation → Constraint and logic programming; Computing methodologies → Symbolic and algebraic manipulation

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

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.8

Category System Description

Supplement Material Coq package source: `https://github.com/math-comp/hierarchy-builder`

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].

© Cyril Cohen and Kazuhiko Sakaguchi and Enrico Tassi;
licensed under Creative Commons License CC-BY

The hierarchy of the Mathematical Components library is implemented following the discipline of Packed Classes initially introduced by Garillot, Gonthier, Mahboubi and Rideau [13] and later also adopted by Affeldt, Nowak, and Saikawain to describe a hierarchy of monadic effects [2] and by Boldo, Lelay, and Melquiond in the Coquelicot library of real analysis [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*, *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 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`. We use notations from Appendix A to define them.

```
HB.mixin Record Monoid_of_Type M := {
  zero : M;
  add : M -> M -> M;
  addrA : associative add;            (* `add` is associative.             *)
  add0r : left_id zero add;           (* `zero` is the left and right neutral  *)
  addr0 : right_id zero add;          (*   element with respect to `add`.    *)
}.
HB.structure Definition Monoid := { M of Monoid_of_Type M }.

HB.mixin Record Ring_of_Monoid R of Monoid R := {
  one : R;
  opp : R -> R;
  mul : R -> R -> R;
  addNr : left_inverse zero opp add;  (* `opp x` is the left and right additive *)
  addrN : right_inverse zero opp add; (*   inverse of `x`.                     *)
  mulrA : associative mul;            (* `mul` is associative.                 *)
  mul1r : left_id one mul;            (* `one` is the left and right neutral   *)
  mulr1 : right_id one mul;           (*   element with respect to `mul`.      *)
  mulrDl : left_distributive mul add; (* `mul` is left and right distributive  *)
  mulrDr : right_distributive mul add; (*   over `add`.                        *)
}.
HB.structure Definition Ring := { R of Monoid R & Ring_of_Monoid R }.
```

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 that gathers operations and properties.

Mixins are declared via the `HB.mixin` command that takes a record declaration with a type parameter 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 `M` into a monoid, hence we chose the name to be `Monoid_of_Type`.

The `HB.structure` command takes in input a definition for a sigma type `S` that equips a type with a list of factories. 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 `R` is followed by `Monoid` that enriches `R` 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.

```
Print Monoid.type. (* Monoid.type  :=  { sort : Type;  ... }                      *)
Check @add.        (* add          :   forall M : Monoid.type, M -> M -> M         *)
Check @addNr.      (* addNr        :   forall R : Ring.type, left_inverse zero opp add *)

Lemma addrC {R : Ring.type} : commutative (@add R).
Proof. (* Proof by Hankel 1867, in Appendix B *) Qed.

Definition Z_Monoid_axioms : Monoid_of_Type Z :=
  Monoid_of_Type.Build Z 0%Z Z.add Z.add_assoc Z.add_0_l Z.add_0_r.

HB.instance Z Z_Monoid_axioms.

Definition Z_Ring_axioms : Ring_of_Monoid Z :=
  Ring_of_Monoid.Build Z 1%Z Z.opp Z.mul
    Z.add_opp_diag_l Z.add_opp_diag_r Z.mul_assoc Z.mul_1_l Z.mul_1_r
    Z.mul_add_distr_r Z.mul_add_distr_l.

HB.instance Z Z_Ring_axioms.

Lemma exercise (m n : Z) : (n + m) - n * 1 = m.
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 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

105 abstract rings with new results.
106 We use the `Monoid_of_Type.Build` *abbreviation* (line 8) in order to build an instance of
107 the `Monoid` structure for binary integers `Z`. We then register that monoid instance as the
108 canonical one on `Z` (line 11) via the command `HB.instance`. We can similarly declare that `Z`
109 forms a ring by using the `Ring_of_Monoid.Build` abbreviation (lines 13 and 18). Note that
110 the `Ring_of_Monoid.Build` abbreviation is not a plain record constructor for `Ring_of_Monoid`,
111 since that would require more arguments, namely the monoid ones (see the _ at line 13).
112 The abbreviation synthesized by $\mathcal{HB}$ infers them automatically (as in [17, Section 7]) thanks
113 to the `HB.instance` declaration given just above.
114 From now on the axioms as well as the abstract theory of rings apply to integers, as shown
115 in lemma `exercise`. The details of the proof do not matter here, what is worth pointing
116 out is that in a single statement we mix monoid (e.g. +) and ring (e.g. -) operations and
117 in the proof we use monoid axioms (e.g. `addrA`), ring axioms (e.g. `addrN`) and ring lemmas
118 (e.g. `addrC`), all seamlessly.

119 2.1 Evolution of the hierarchy

120 We proceed by accommodating the intermediate structure of Abelian groups.

```
1  HB.mixin Record Monoid_of_Type M := { ... (* unchanged *) ... }.
2  HB.structure Definition Monoid := { M of Monoid_of_Type M }.
3
4  HB.mixin Record AbelianGroup_of_Monoid A of Monoid A := {
5    opp : A -> A;
6    addrC : commutative (add : A -> A -> A);
7    addNr : left_inverse zero opp add;
8  }.
9  HB.structure Definition AbelianGroup := { A of Monoid A & AbelianGroup_of_Monoid A }.
10
11 HB.mixin Record Ring_of_AbelianGroup R of AbelianGroup R := {
12   one : R;
13   mul : R -> R -> R;
14   mulrA : associative mul;
15   mul1r : left_id one mul;              mulr1 : right_id one mul;
16   mulrDl : left_distributive mul add;   mulrDr : right_distributive mul add;
17 }.
18 HB.structure Definition Ring := { R of AbelianGroup R & Ring_of_AbelianGroup R }.
19
20 Lemma addrN {A : AbelianGroup.type} : right_inverse zero opp add.
21 Proof. by move=> x; rewrite addrC addNr. Qed.
```

121 Some operations and properties were moved from the old mixin for rings into a newborn
122 mixin `AbelianGroup_of_Monoid` that gathers the axioms needed to turn a monoid into an
123 Abelian group. Consequently the mixin for rings is now called `Ring_of_AbelianGroup` (instead
124 of `Ring_of_Monoid`) since it expects the type `R` to be already an Abelian group and hence
125 gathers fewer axioms. While operations moved from one structure to another, some properties
126 undergo a deep change in their status. The lemma `addrC` part of the abstract theory of rings
127 is now an axiom of Abelian groups, while `addrN` is no more an axiom of rings, but rather a
128 theorem of the abstract theory of Abelian groups proved at line 20.
129 It is worth clarifying here what axioms and lemmas are by looking at the two distinct
130 use cases for a hierarchy: 1) develop the abstract theory of some structure; 2) apply the
131 abstract theory on some concrete example. In the first case all the axioms of the structure
132 are (assumed to be) theorems so the distinction between axiom and theorem does not matter.
133 This is what happens in the proof of `addNr` by taking `R` to be of type `AbelianGroup.type` as an
134 assumption. In the second case, in order to show Coq that a data type and some operations

forms a structure, one must pick the axioms of the structure an prove them for that specific type and operations. For example the `Ring_of_Monoid.Build` abbreviation is used at page 3 to package the set of proofs that make `Z` a ring (proviso `Z` is already a monoid).

With this new version of the hierarchy, that we name V2, the axiomatic that was previously exposed to user changed, and indeed 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

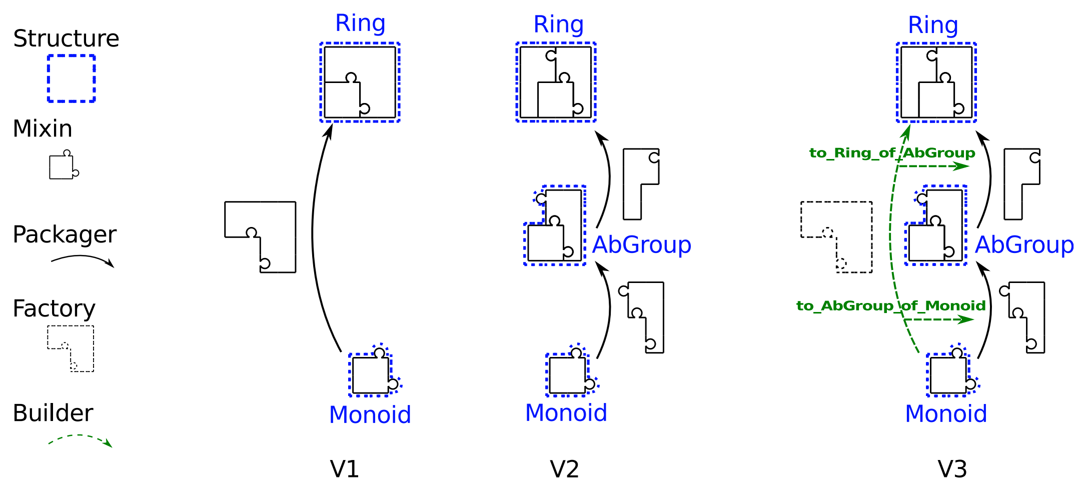

Figure 1 The evolution of the hierarchy. V3 is backward compatible with V1, while V2 is not.

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.

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.

```
HB.factory Record Ring_of_Monoid R of Monoid R := { ... (* unchanged *) ...}.

HB.builders Context R (f : Ring_of_Monoid R).

  Lemma addrC : commutative add. Proof. (* The same proof as before *) Qed.

  Definition to_AbelianGroup_of_Monoid :=
    AbelianGroup_of_Monoid.Build R opp addrC addNr. (* addrN unused *)
  HB.instance R to_AbelianGroup_of_Monoid.

  Definition to_Ring_of_AbelianGroup := Ring_of_AbelianGroup.Build R one mul
    mulrA mul1r mulr1 mulrDl mulrDr.
  HB.instance R to_Ring_of_AbelianGroup.

HB.end.
```

153 The record Ring_of_Monoid is the same we declared as a mixin in version V1. In order to
154 make a factory out of it we equip it with two definitions that embody the *builders*. The first
155 is to_AbelianGroup_of_Monoid and explains how to build an AbelianGroup structure out of the
156 factory axioms (named f, line 3). This construction is also registered as canonical for R, so
157 that the next construction to_Ring_of_AbelianGroup can call the Ring_of_AbelianGroup.Build
158 abbreviation that requires R to be an AbelianGroup. It is worth pointing out that the proof
159 of addrC we had in V1 is now required in order to write the builder for Abelian groups, while
160 the addrN field is not used (the same statement is already part of the theory of Abelian
161 groups, see line 20 of the previous code snippet).
162 Thanks to this factory we can now declare Z to be an instance of a ring using the
163 Ring_of_Monoid.Build abbreviation. The associated builders generate, behind the scenes,
164 instances of the Ring_of_AbelianGroup and AbelianGroup_of_Monoid mixins that in turn are
165 used to build instances of the AbelianGroup and Ring structures. Indeed, when used in the
166 context of the hierarchy version V3, the command HB.instance Z Z_Ring_axioms makes Z an
167 instance of *both* structures, and not just the Ring one as in version V1. Thanks to that the
168 proof of example can use the theory of both structures, for example addrC holds on Abelian
169 groups, addrA holds on monoids, while addr1 holds on rings. As a result the very same proof
170 works on both version V1 and V3.
171 Last, it is worth pointing out that the new factory makes the following two lines equivalent
172 (the former declares rings in V1) since they both describe the same set of mixins.

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

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

175 2.3 $\mathcal{HB}$ in a nutshell

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

- 177 Operations and properties (axioms) are made available to the user as soon as they are
178 used in a structure. The hierarchy designer is free to move them from one to another and
179 replace an axiom by a lemma and vice versa.
- 180 Structures cannot disappear but the way they are built may change. The hierarchy
181 designer is free to split structures into smaller ones in order to better factor and reuse
182 parts of the hierarchy and the library that follows it.
- 183 Mixins cannot disappear but can change considerably in their implementation. A mixin
184 can become a full fledged factory equipped with builders to ensure backward compatibility.
- 185 $\mathcal{HB}$ high level commands compile to the discipline of Packed Classes (Coq modules, records,
186 coercions, implicit arguments, canonical structure instances, notations, abbreviations).
187 This process lifts a considerable burden from the shoulders of the hierarchy designer and
188 the final user who are no longer required to master the details of Packed Classes.

189 3 The $\mathcal{HB}$ language

190 The Coq terms handled by $\mathcal{HB}$ are subdivided in five categories, the mixins $\mathcal{M}$, the factories $\mathcal{F}$,
191 builders $\mathcal{B}$, the classes $\mathcal{C}$ and the structures $\mathcal{S}$. Mixins, factories and instances are tagged
192 by the user, through the commands HB.mixin, HB.factory and HB.instance, and the user
193 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 Mixins, factories and instances

In this section, we call "distinguished" a Coq definition or record that the developer of a library has labeled "mixin", "factory", "builder", "class" or "structure".

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

$$\texttt{Record}\ m\ \texttt{(T : Type)}\ \overrightarrow{(p : m\ T\ p_\sigma)}^n\ \texttt{: Type := \{ .. \}.}$$

where $\overrightarrow{(p : m\ T\ p_\sigma)}^n = (p_1 : m_1\ T)\ldots(p_n : m_n\ T\ p_{\sigma(n,1)}\ \ldots\ p_{\sigma(n,q_n)})$ *and where for all* $i \in \{1,\ldots,n\}$ *we have* $q_i \in \mathbb{N}$ *and the arguments of* $m_i \in \mathcal{M}$ *consist in* q_i *of the previously quantified mixin parameters* p_k*, i.e. for all* $k \in \{1,\ldots,q_i\}$*, 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 $dep(m_i) = \left\{m_{\sigma(i,1)},\ldots,m_{\sigma(i,q_i)}\right\}$.

▶ **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 and describes a DAG).

$$\forall m \in \mathcal{M}, \qquad dep^\star(dep^\star(m)) \subseteq dep^\star(m) \quad \textit{and} \quad m \notin dep(m).$$

Proof. Indeed dep is transitively closed because records are well typed in the empty context and describes a DAG since Coq does not admit circular definitions. ◀

▶ **Definition 6** (factories $\mathcal{F}$). *A* factory $f \in \mathcal{F}$ *is a distinguished Coq record or definition with one or more parameters. The first parameter must be a* `(T : Type)`*, while the other parameters are* n *mixins, applied to* `T` *and previous mixin variables. I.e.*

$$\texttt{Record (* or Definition *)}\ f\ \texttt{(T : Type)}\ \overrightarrow{(p : m\ T\ p_\sigma)}^n\ \texttt{: Type := ...}$$

▶ **Definition 7** ($requires$, factory requirements.). *Given* $f \in \mathcal{F}$*, we define* $requires(f) \in \wp(\mathcal{M})$ *as the set of all mixins that occur as parameters of* f*, i.e.* $dep(f) = \{m_1,\ldots,m_n\}$*.*

The following property holds,

▶ **Proposition 8** ($requires$ is closed under dep). $\forall f \in \mathcal{F}, dep^\star(requires(f)) \subseteq requires(f)$.

Proof. Because Coq records and global definitions are well typed in the empty context. ◀

▶ **Definition 9** (builders $\mathcal{B}$). *A* builder $\mu \in \mathcal{B}$ *is a distinguished function whose return type is a mixin* $m_{n+1} \in \mathcal{M}$ *and whose parameters are the carrier type* `(T : Type)`*, mixins* $\{m_1,\ldots,m_n\} \in \wp(\mathcal{M})$ *and a* factory $f \in \mathcal{F}$*, such that* $requires(f) = \{m_1,\ldots,m_n\}$*. I.e.*

$$\texttt{Definition}\ \mu\ \texttt{(T : Type)}\ \overrightarrow{(p : m\ T\ p_\sigma)}^n\texttt{:}\ f\ T\ p_1\ldots p_n\ \texttt{->}\ m_{n+1}\ T\ p_{\sigma(n+1,1)}\ \ldots\ p_{\sigma(n+1,q_{n+1})}\ \texttt{:= ...}$$

Note that the builders of a given factory have the same set of dependencies.

225 ▶ **Definition 10** (from). *We define* from$(f, m_{n+1}) = \mu$ *to be the (unique) builder for* m_{n+1}
226 *from the factory* f*, when it exists.*

227 Note that from is not a total function, and that from(f, m) is defined if and only if there
228 is a declared builder that shows how to build m from f. In this case, we say f provides m.

229 ▶ **Definition 11** (*provides*). $\mathit{provides}(f) = \{m \mid \mathsf{from}(f,m) \text{ is defined}\} \in \wp(\mathcal{M})$ *the set of*
230 *mixins that a factory* $f \in \mathcal{F}$ *provides, through its builders.*

231 Mixins are declared by the user as the fundamental building blocks of a hierarchy. As the
232 next proposition shows they shall not be regarded as different from regular factories, since
233 they are a special case.

234 ▶ **Proposition 12** ($\mathcal{M} \subseteq \mathcal{F}$). *There is a way to see mixins as factories.*

235 **Proof.** For all $m \in \mathcal{M}$ we have $\mathit{requires}(m) = \mathit{dep}(m)$, $\mathit{provides}(m) = \{m\}$ and from$(m, m) =$
236 $\left(\texttt{fun T } \overrightarrow{(p : m\ T\ p_\sigma)}^n \texttt{ (x : } m \texttt{ T } p_1 \ \ldots p_n\texttt{) => x}\right) \in \mathcal{B}$. ◀

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

```
1 HB.mixin Record M T of f1 ... fn := { .. }.
```

238 Declares the record `axioms` inside a module `M`. This record `M.axioms` depends on the mixins
239 $\mathit{requires}(\texttt{M.axioms}) = \mathit{dep}(\texttt{M.axioms}) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and is registered both as a
240 mixin and a factory. Finally It exports an abbreviation `M.Build` and a notation `M` standing
241 for `M.axioms`, so that the module name can be used to denote the `axioms` record it contains.

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

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

```
1 HB.factory Record F T of f1 ... fn := { .. }.
2 HB.builders Context T (a : F T).
3
4   Definition b_{n+1} : f_{n+1} T .. := ...
5   HB.instance T b_{n+1}.
6   ..
7   Definition b_{n+k} : f_{n+k} T .. := ...
8   HB.instance T b_{n+k}.
9 HB.end.
```

247 Declares the record `axioms` inside a module `F`. This record `F.axioms` depends on the mixins
248 $\mathit{requires}(\texttt{F.axioms}) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and is registered as a factory. It exports an
249 abbreviation `F.Build` and a notation `F` standing for `F.axioms`, so that the module name
250 can be used to denote the `axioms` record it contains. Finally it uses the factory instances
251 $b_{n+1}, \ldots, b_{n+k}$ provided by the user in order to derive builders, so that $\mathit{provides}(\texttt{F}) =$
252 $\mathit{provides}^\star(\{f_{n+1}, \ldots, f_{n+k}\})$.
253 It is thus necessary that $\mathit{requires}^\star(\{f_{n+1}, \ldots, f_{n+k}\}) \subseteq \mathit{provides}^\star(\{f_1, \ldots, f_n\})$.

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

3.2 Classes and structures

▶ **Definition 13** ($\mathcal{C}$, class). *A class* `c` $\in \mathcal{C}$ *is a distinguished 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) := { p1 : m1 T; .. ; pi : mi T pσ(i,1) ...pσ(i,qi); ..}.
```

▶ **Definition 14** (*def*, class definition). *We call* $def(c) \in \wp(\mathcal{M})$ *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. $\exists c \in \mathcal{C},\ def(c) = requires^\star(f) \cup provides^\star(f)$,
2. $\exists C \subseteq \mathcal{C},\ def^\star(C) = requires^\star(f)$.

Classes could also be seen as factories.

▶ **Proposition 15** ($\mathcal{C} \subseteq \mathcal{F}$). *For all* $m \in def(c)$ *we have* $\mathsf{from}(c, m_i) = p_i$*, and it follows that* $requires(c) = \emptyset$*,* $provides(c) = def(c)$*.*

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 16** ($\mathcal{S}$ Structure). *A structure* `s` $\in \mathcal{S}$ *is a distinguished 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* `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 $def(c) = provides^\star(\{f_1, \ldots, f_n\})$ and the corresponding structure `M.type` $\in \mathcal{S}$, together with unification hints as described in Section 4.

▶ **Definition 17** ($\leq\ \in \mathcal{C} \times \mathcal{C}$, subclass). $c1 \ \leq\ c2$ *iff* $def(c2) \subseteq def(c1)$

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 (vector) := F_aux { ... }.
Notation f T := (f_aux T ...(* p1...pn inferred when T is known *)).
Notation F T  x1 ... xk := (F_aux T ...(* p1...pn inferred when T is known *) x1 ... xk).
Definition b : f T := F T x1 ... xn.
```

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

The language of Packed Classes [13] is 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 [17] instances. In this section, we describe the *flat* variant of Packed Classes, the target language of `hierarchy-builder`, through the hierarchy V3 extended with semirings.

The `hierarchy-builder` addon can generate all the Coq declarations in this section automatically, but some details are omitted for brevity in this section.

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 (M : Type) : Type :=
  Class { Monoid_of_Type_mixin : Monoid_of_Type M; }.
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 {M : Monoid.type} : M :=
  Monoid_of_Type.zero M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
Definition add {M : Monoid.type} : M -> M -> M :=
  Monoid_of_Type.add M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
Lemma addrA {M : Monoid.type} : associative (@add M).
(* 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 of 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 (A : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type A;
  AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid A Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort }.
End AbelianGroup.
```

In the flat variant of Packed Classes, a class record gathers mixins as its fields directly as in above. In contrast, a class record of the non-flat variant packs a class instance of one of the superclasses with remaining mixins, which reduces the amount of code to implement

322 inheritance significantly. Since we do not need to care about the amount of code in automated
323 generation, `hierarchy-builder` uses the flat variant of Packed Classes as its target language.
324 As in the monoid structure, we declare `AbelianGroup.sort` as an implicit coercion and lift
325 additive inverse `opp` and Abelian group axioms. Since Abelian groups inherit from monoids,
326 we also declare an implicit coercion from Abelian groups to monoids. A coercion between
327 structures can be defined in two steps: first a coercion between the class record of the
328 superclass to the class record of the subclass (line 2); and a coercion between structure
329 records (line 4) relying on the first one.

```
1 Coercion AbelianGroup.sort : AbelianGroup.type >-> Sortclass.
2 Coercion AbelianGroup_class_to_Monoid_class (A : Type) (c : AbelianGroup A) :
3   Monoid A := Monoid.Class A (AbelianGroup.Monoid_of_Type_mixin A c).
4 Coercion AbelianGroup_to_Monoid (A : AbelianGroup.type) : Monoid.type :=
5   Monoid.Pack (AbelianGroup.sort A) (AbelianGroup.class A).
6
7 Definition opp {A : AbelianGroup.type} : A -> A := ... .
8 (* Two Abelian group axioms `addrC` and `addNr` are omitted. *)
```

330 Generally, a coercion from a structure X to another structure Y can (and should) have the
331 following form, thanks to the corresponding coercion between classes `X_class_to_Y_class`.

```
1 Coercion X_to_Y (T : X.type) : Y.type :=
2   Y.Pack (X.sort T) ((* X_class_to_Y_class _ *) (X.class T)).
```

332 4.2 Multiple inheritance

333 In order to introduce multiple inheritance, we extend
334 the hierarchy described in Section 2.2 with the struc-
335 ture of semirings as depicted in Figure 2, and name
336 it V4. Semirings introduce the binary multiplication
337 operator `mul` and its identity element `one`. For the
338 sake of completeness, the full code of V4 is available
339 in Appendix C.

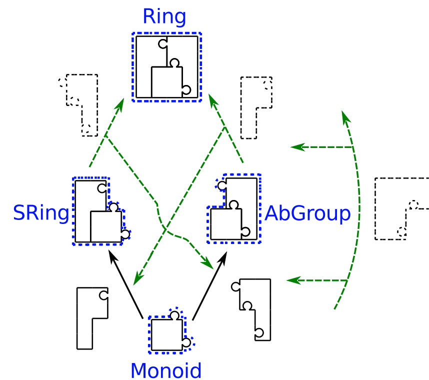

Figure 2 V4 introduces multiple inheritance. For brevity we omit the factories/builders for the upper arrows.

```
1  HB.mixin Record SemiRing_of_Monoid S
2    of Monoid S := {
3    one : S;
4    mul : S -> S -> S;
5    (* 7 axioms are omitted. *)
6  }.
7
8  HB.factory Record Ring_of_AbelianGroup R
9    of AbelianGroup R := {
10   (* 2 operators and 5 axioms are omitted. *)
11 }.
12 HB.builders Context R (f : Ring_of_AbelianGroup R).
13 ..
14 HB.end.
```

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

```
1 Module SemiRing.
2 Record axioms (S : Type) : Type := Class {
3   Monoid_of_Type_mixin : Monoid_of_Type S;
4   SemiRing_of_Monoid_mixin : SemiRing_of_Monoid S Monoid_of_Type_mixin;
5 }.
6 Structure type : Type := Pack { sort : Type; class : axioms sort; }.
7 End SemiRing.
```

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

```
1 Coercion SemiRing.sort : SemiRing.type >-> Sortclass.
2 Coercion SemiRing_to_Monoid : SemiRing.type >-> Monoid.type.
3
4 Definition one {S : SemiRing.type} : S := ... .
5 Definition mul {S : SemiRing.type} : S -> S -> S := ... .
6 (* 7 semiring axioms are omitted. *)
```

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

```
1 Module Ring.
2 Record axioms (R : Type) : Type := Class {
3   Monoid_of_Type_mixin : Monoid_of_Type R;
4   SemiRing_of_Monoid_mixin : SemiRing_of_Monoid R Monoid_of_Type_mixin;
5   AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid R Monoid_of_Type_mixin; }.
6 Structure type : Type := Pack { sort : Type; class : axioms sort; }.
7 End Ring.
8
9 Coercion Ring.sort : Ring.type >-> Sortclass.
10 Coercion Ring_to_Monoid : Ring.type >-> Monoid.type.
11 Coercion Ring_to_SemiRing : Ring.type >-> SemiRing.type.
12 Coercion Ring_to_AbelianGroup : Ring.type >-> AbelianGroup.type.
```

347 4.3 Linking structures and instances via Canonical Structures

348 We recall here how to use the Canonical Structures [17, 25] mechanism, which lets the user
349 extend the elaborator of Coq, in order to handle inheritance and inference of structure [1,
350 26, 13]. This software component takes as input a term as written by the user and has to
351 infer all the missing information that is necessary in order to make the term well typed.
352 A first example of elaboration that requires canonical structures is $0 + 1$. After removing
353 all syntactic facilities, the underlying Coq term is `(add _ (zero _)) (one _)` where `_` stands
354 for an implicit piece of information to be inferred and the constants `add` and `zero` come from
355 the monoid structure while `one` comes from semirings. When the term is type checked each `_`
356 is replaced by a unification variable written $?_v$. Respectively, the head and the argument of
357 the top application can be typed as follows:

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

358 where $?_{\mathtt{M}}$ `: Monoid.type` and $?_{\mathtt{SR}}$ `: SemiRing.type`. In order to type check the applica-
359 tion Coq has to unify `Monoid.sort` $?_{\mathtt{M}}$ with `SemiRing.sort` $?_{\mathtt{SR}}$, which is not trivial: it
360 amounts at finding a structure that is both a monoid and a semiring, possibly the smal-
361 lest one [26, Sect. 4]. This piece of information can be inferred from the hierarchy and
362 its inheritance relation (Definition 17) and we can tell Coq to exploit it by declaring
363 `SemiRing_to_Monoid : SemiRing.type -> Monoid.type` as canonical. With that hint Coq will
364 pick $?_{\mathtt{M}}$ to be `SemiRing_to_Monoid` $?_{\mathtt{SR}}$ as the canonical solution for this unification problem.
365 In general, all the coercions between structures must be declared as canonical.
366 Another example of elaboration problem is -1, which hides the term `opp _ (one _)`. Here
367 `opp` and `one` are respectively from Abelian groups and semirings, which do not inherit each
368 other but whose smallest common substructure is rings; thus we have to extend the unifier
369 to solve a unification problem `AbelianGroup.sort` $?_{\mathtt{AbG}}$ `=` `SemiRing.sort` $?_{\mathtt{SR}}$ by instantiating
370 $?_{\mathtt{AbG}}$ and $?_{\mathtt{SR}}$ with `Ring_to_AbelianGroup` $?_{\mathtt{R}}$ and `Ring_to_SemiRing` $?_{\mathtt{R}}$ respectively where $?_{\mathtt{R}}$ is
371 a fresh unification variable of type `Ring.type`. This hint can be given as follows:

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

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 of $\mathcal{HB}$ 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 backtracking: 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 superset of the one of Canonical Structures), but we could not find any written trace of that. Language-wise they provide 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 axiom 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 $\mathcal{HB}$, but its design took several years of attempts and fruitful discussions with the people we thank below.

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.

Acknowledgments

We are particularly grateful to Georges Gonthier, Assia Mahboubi and Florent Hivert for the many discussions we had that were instrumental in coining the concepts formalized here. We thank as well other developers of the Mathematical Components library and members of Dagstuhl seminar 18341, for the helpful discussions we had. We also thank many Coq developers and users who showed interest and support in this work. Finally, we thank the anonymous reviewers of this paper for their useful comments.

References

1. Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: a case study in functional analysis. accepted in the proceedings of IJCAR 2020, available at `https://hal.inria.fr/hal-02463336`, 2020.
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. `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. `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. `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. `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. `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. `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*

564 *Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013.
565 doi:10.1007/978-3-642-39634-2_14.
566 **15** Adam Grabowski, Artur Korniłowicz, and Christoph Schwarzweller. On algebraic hierarchies
567 in mathematical repository of Mizar. In *2016 Federated Conference on Computer Science and*
568 *Information Systems (FedCSIS)*, pages 363–371, Sep. 2016.
569 **16** Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical
570 analysis in Isabelle/HOL. In *Interactive Theorem Proving - 4th International Conference,*
571 *ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 279–294. Springer, 2013.
572 doi:10.1007/978-3-642-39634-2_21.
573 **17** Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In *Interactive*
574 *Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013.*
575 *Proceedings*, pages 19–34, 2013. doi:10.1007/978-3-642-39634-2_5.
576 **18** The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM*
577 *SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages
578 367—381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/
579 3372885.3373824.
580 **19** Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge
581 University Press, New York, NY, USA, 1st edition, 2012.
582 **20** Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the*
583 *ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*,
584 PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery.
585 doi:10.1145/53990.54010.
586 **21** Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*,
587 13:386–402, 2002. doi:10.1007/s001650200018.
588 **22** Florian Rabe and Michael Kohlhase. A scalable module system. *Information and Com-*
589 *putation*, 230:1–54, 2013. URL: http://www.sciencedirect.com/science/article/pii/
590 S0890540113000631, doi:https://doi.org/10.1016/j.ic.2013.06.001.
591 **23** Damien Rouhling. *Formalisation Tools for Classical Analysis – A Case Study in Control Theory.*
592 PhD thesis, Université Côte d'Azur, 2019. URL: https://hal.inria.fr/tel-02333396.
593 **24** Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *24th ACM SIGPLAN-*
594 *SIGACT Symposium on Principles of Programming Languages (POPL 1997), Paris, France,*
595 *15–17 January 1997*, pages 292–301. ACM, 1997.
596 **25** Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation*
597 *des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formal-*
598 *ization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration.*
599 *Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France,
600 1999. URL: https://tel.archives-ouvertes.fr/tel-00523810.
601 **26** Kazuhiko Sakaguchi. Validating mathematical structures. accepted in the proceedings of
602 IJCAR 2020, available at https://arxiv.org/abs/2002.00620, 2020.
603 **27** Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathemat-*
604 *ical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.
605 **28** Enrico Tassi. Coq-Elpi, Coq plugin embedding Elpi. https://github.com/LPCIC/coq-elpi,
606 2020.

A Coq reference

```
Section OperationProperties.

Variables (T : Type) (e : T) (inv : T -> T) (op : T -> T -> T) (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 right_inverse := forall x, op x (inv 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).

End OperationProperties.
```

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 (opp 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 ^~ (opp 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.
```

C Full V4 code

```
From Coq Require Import ssreflect ssrfun ZArith.
From HB Require Import structures.

Declare Scope hb_scope.
Delimit Scope hb_scope with G.
Open Scope hb_scope.

(* Bottom mixin in Fig. 2. *)
HB.mixin Record Monoid_of_Type M := {
  zero : M;
  add : M -> M -> M;
  addrA : associative add;
  add0r : left_id zero add;
  addr0 : right_id zero add;
}.
HB.structure Definition Monoid := { M of Monoid_of_Type M }.
Notation "0" := zero : hb_scope.
Infix "+" := (@add _) : hb_scope.

(* Bottom right mixin in Fig. 2. *)
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 }.
```

```
27  Notation "- x" := (@opp _ x) : hb_scope.
28  Notation "x - y" := (x + - y) : hb_scope.
29
30  (* Bottom left mixin in Fig. 2. *)
31  HB.mixin Record SemiRing_of_Monoid S of Monoid S := {
32    one : S;
33    mul : S -> S -> S;
34    mulrA : associative mul;
35    mul1r : left_id one mul;
36    mulr1 : right_id one mul;
37    mulrDl : left_distributive mul add;
38    mulrDr : right_distributive mul add;
39    mul0r : left_zero zero mul;
40    mulr0 : right_zero zero mul;
41  }.
42  HB.structure Definition SemiRing := { S of Monoid S & SemiRing_of_Monoid S }.
43  Notation "1" := one : hb_scope.
44  Infix "*" := (@mul _) : hb_scope.
45
46  Lemma addrN {A : AbelianGroup.type} : right_inverse (zero : A) opp add.
47  Proof. by move=> x; rewrite addrC addNr. Qed.
48
49  (* Top right factory in Fig. 2. *)
50  HB.factory Record Ring_of_AbelianGroup R of AbelianGroup R := {
51    one : R;
52    mul : R -> R -> R;
53    mulrA : associative mul;
54    mul1r : left_id one mul;
55    mulr1 : right_id one mul;
56    mulrDl : left_distributive mul add;
57    mulrDr : right_distributive mul add;
58  }.
59
60  (* Builder arrow from top right to bottom left in Fig. 2. *)
61  HB.builders Context (A : Type) (f : Ring_of_AbelianGroup A).
62
63    Fact mul0r : left_zero zero mul.
64    Proof.
65    move=> x; rewrite -[LHS]add0r addrC.
66    rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
67    by rewrite -mulrDl add0r addrC addNr.
68    Qed.
69
70    Fact mulr0 : right_zero zero mul.
71    Proof.
72    move=> x; rewrite -[LHS]add0r addrC.
73    rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
74    by rewrite -mulrDr add0r addrC addNr.
75    Qed.
76
77    Definition to_SemiRing_of_Monoid := SemiRing_of_Monoid.Build A _ mul mulrA
78       mul1r mulr1 mulrDl mulrDr mul0r mulr0.
79    HB.instance A to_SemiRing_of_Monoid.
80
81  HB.end.
82  HB.structure Definition Ring := { A of AbelianGroup A & Ring_of_AbelianGroup A }.
83
84  (* Top left factory in Fig. 2. *)
85  (* It is an exact copy of the bottom right mixin. *)
86  HB.factory Definition Ring_of_SemiRing R of SemiRing R := AbelianGroup_of_Monoid R.
87  (* The corresponding builder is the identity. *)
88  HB.builders Context (R : Type) (f : Ring_of_SemiRing R).
89    Definition to_AbelianGroup_of_Monoid : AbelianGroup_of_Monoid R := f.
90    HB.instance R to_AbelianGroup_of_Monoid.
91  HB.end.
92
```

```
93  (* Right-most factory in Fig. 2. *)
94  HB.factory Record Ring_of_Monoid R of Monoid R := {
95    one : R;
96    opp : R -> R;
97    mul : R -> R -> R;
98    addNr : left_inverse zero opp add;
99    addrN : right_inverse zero opp add;
100   mulrA : associative mul;
101   mul1r : left_id one mul;
102   mulr1 : right_id one mul;
103   mulrDl : left_distributive mul add;
104   mulrDr : right_distributive mul add;
105 }.
106
107 HB.builders Context (R : Type) (f : Ring_of_Monoid R).
108
109   Lemma addrC : commutative (add : R -> R -> R).
110   Proof. (* Exactly the same as in Appendix B. *)
111   have innerC (a b : R) : a + b + (a + b) = a + a + (b + b).
112     by rewrite -[a+b]mul1r -mulrDl mulrDr !mulrDl !mul1r.
113   have addKl (a b c : R) : a + b = a + c -> b = c.
114     apply: can_inj (add a) (add (opp a)) _ _ _.
115     by move=> x; rewrite addrA addNr add0r.
116   have addKr (a b c : R) : b + a = c + a -> b = c.
117     apply: can_inj (add ^~ a) (add ^~ (opp a)) _ _ _.
118     by move=> x; rewrite /= -addrA addrN addr0.
119   move=> x y; apply: addKl (x) _ _ _; apply: addKr (y) _ _ _.
120   by rewrite -!addrA [in RHS]addrA innerC !addrA.
121   Qed.
122
123   (* Builder to the bottom right mixin. *)
124   Definition to_AbelianGroup_of_Monoid :=
125     AbelianGroup_of_Monoid.Build R opp addrC addNr.
126   HB.instance R to_AbelianGroup_of_Monoid.
127
128   (* Builder to the top right factory, which is compiled to the bottom left mixin. *)
129   Definition to_Ring_of_AbelianGroup := Ring_of_AbelianGroup.Build R one mul
130     mulrA mul1r mulr1 mulrDl mulrDr.
131   HB.instance R to_Ring_of_AbelianGroup.
132
133 HB.end.
```