



HAL
open science

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.

1 Hierarchy Builder: algebraic hierarchies made easy 2 in Coq with Elpi

3 Cyril Cohen

4 Inria, Université Côte d'Azur, France

5 Cyril.Cohen@inria.fr

6 Kazuhiko Sakaguchi

7 University of Tsukuba, Japan

8 sakaguchi@logic.cs.tsukuba.ac.jp

9 Enrico Tassi

10 Inria, Université Côte d'Azur, France

11 Enrico.Tassi@inria.fr

12 — Abstract —

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

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

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

27 **2012 ACM Subject Classification** Software and its engineering → Formal language definitions;
28 Theory of computation → Type theory; Theory of computation → Constraint and logic programming;
29 Computing methodologies → Symbolic and algebraic manipulation

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

31 **Digital Object Identifier** 10.4230/LIPIcs.FSCD.2020.8

32 **Category** System Description

33 **Supplement Material** Coq package source: <https://github.com/math-comp/hierarchy-builder>

34 **1** Introduction

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



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

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).

Editor: Zena M. Ariola; Article No. 8; pp. 8:1–8:21



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 The hierarchy of the Mathematical Components library is implemented following the dis-
 45 cipline of Packed Classes initially introduced by Garillot, Gonthier, Mahboubi and Rideau [13]
 46 and later also adopted by Affeldt, Nowak, and Saikawain to describe a hierarchy of monadic
 47 effects [2] and by Boldo, Lelay, and Melquiond in the Coquelicot library of real analysis [6].
 48 We call Packed Classes a discipline, and not a language, because, in spite of its many virtues,
 49 it is unwieldy to use. In particular it leaks to the user many of the technical details of
 50 the Coq system. As a result, one needs to be a Coq expert in order to build or modify a
 51 hierarchy, and even experts make mistakes as shown in [26]. Another inconvenience of the
 52 Packed Classes discipline is that even simple changes to the hierarchy, such as splitting a
 53 structure into two simpler ones, break user code.

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

61
 62 To sum up, the main contributions of the paper are:

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

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

70 2 \mathcal{HB} by examples: building and evolving a hierarchy

71 The first version of our hierarchy (that we name V1) features only two structures: `Monoid`
 72 and `Ring`. We use notations from Appendix A to define them.

```

1  HB.mixin Record Monoid_of_Type M := {
2    zero : M;
3    add : M -> M -> M;
4    addrA : associative add;          (* `add` is associative.          *)
5    addOr : 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 := { M of Monoid_of_Type M }.
9
10 HB.mixin Record Ring_of_Monoid R of Monoid R := {
11   one : R;
12   opp : R -> R;
13   mul : R -> R -> R;
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 := { R of Monoid R & Ring_of_Monoid R }.

```

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

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

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

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

88 The last line declares the `Ring` structure to hold all the axioms declared so far. We can
 89 now inspect the contents of the hierarchy and then proceed to build a theory about abstract
 90 rings, register examples (instances) of ring structures and finally use the abstract theory on
 91 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 {R : Ring.type} : commutative (@add R).
6 Proof. (* Proof by Hankel 1867, in Appendix B *) Qed.
7
8 Definition Z_Monoid_axioms : Monoid_of_Type Z :=
9   Monoid_of_Type.Build Z 0%Z 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 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.

```

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

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

8:4 Hierarchy Builder

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 `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    addrNr : 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   mulr1 : 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 addrNr. 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.

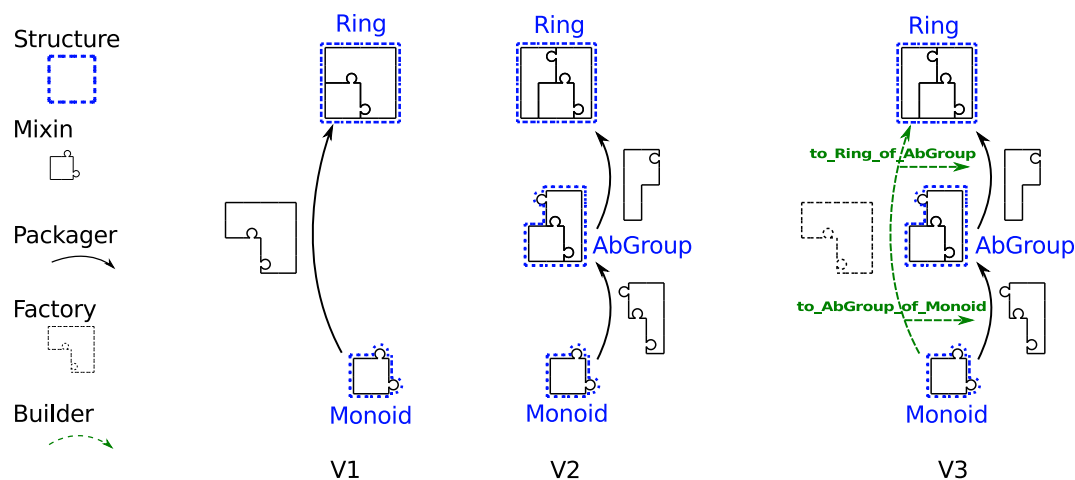
129 It is worth clarifying here what axioms and lemmas are by looking at the two distinct
130 use cases for a hierarchy: 1) develop the abstract theory of some structure; 2) apply the
131 abstract theory on some concrete example. In the first case all the axioms of the structure
132 are (assumed to be) theorems so the distinction between axiom and theorem does not matter.
133 This is what happens in the proof of `addrNr` 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

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

138 With this new version of the hierarchy, that we name `V2`, the axiomatic that was previously
 139 exposed to user changed, and indeed code written for version `V1` breaks. For example the
 140 declaration of the canonical ring over the integers fails, if only because we do not have a
 141 `Ring_of_Monoid.Build` abbreviation anymore.

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

144 2.2 The missing puzzle piece



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

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

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

```

1 HB.factory Record Ring_of_Monoid R of Monoid R := { ... (* unchanged *) ...}.
2
3 HB.builders Context R (f : Ring_of_Monoid R).
4
5 Lemma addrC : commutative add. Proof. (* The same proof as before *) Qed.
6
7 Definition to_AbelianGroup_of_Monoid :=
8   AbelianGroup_of_Monoid.Build R opp addrC addNr. (* addrN unused *)
9 HB.instance R to_AbelianGroup_of_Monoid.
10
11 Definition to_Ring_of_AbelianGroup := Ring_of_AbelianGroup.Build R one mul
12   mulrA mulr mulr1 mulrD1 mulrDr.
13 HB.instance R to_Ring_of_AbelianGroup.
14
15 HB.end.
```

8:6 Hierarchy Builder

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 *HB* in a nutshell

176 By using *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 ■ *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 *HB* language

190 The Coq terms handled by *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

194 command `HB.structure` and builders are generated when using `HB.instance` while declaring
 195 a factory, and the user may only refer to structure types, but should never rely on their
 196 implementation, neither should they rely on explicit builders.

197 3.1 Mixins, factories and instances

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

200 ► **Definition 1** (\mathcal{M} , mixins). A mixin $m \in \mathcal{M}$ is a distinguished Coq record with one or more
 201 parameters. The first parameter must be a $(T : \mathbf{Type})$, while the other parameters are mixins
 202 $(m_i)_{i \in \{1, \dots, n\}}$, each of which is applied to T and possibly previous mixin variables. I.e.

1 **Record** $m (T : \mathbf{Type}) \overline{(p : m T p_\sigma)^n} : \mathbf{Type} := \{ \dots \}$.

203 where $\overline{(p : m T p_\sigma)^n} = (p_1 : m_1 T) \dots (p_n : m_n T p_{\sigma(n,1)} \dots p_{\sigma(n,q_n)})$ and where for all
 204 $i \in \{1, \dots, n\}$ we have $q_i \in \mathbb{N}$ and the arguments of $m_i \in \mathcal{M}$ consist in q_i of the previously
 205 quantified mixin parameters p_k , i.e. for all $k \in \{1, \dots, q_i\}$, we have $\sigma(i, k) \in \{1, \dots, i-1\}$.

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

208 ► **Remark 3.** For all $i \in \{1, \dots, n\}$, we have $dep(m_i) = \{m_{\sigma(i,1)}, \dots, m_{\sigma(i,q_i)}\}$.

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

► **Proposition 5** (dep is transitively closed and describes a DAG).

210 $\forall m \in \mathcal{M}, \quad dep^*(dep^*(m)) \subseteq dep^*(m) \quad \text{and} \quad m \notin dep(m).$

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

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

1 **Record** (** or Definition **) $f (T : \mathbf{Type}) \overline{(p : m T p_\sigma)^n} : \mathbf{Type} := \dots$

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

218 The following property holds,

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

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

221 ► **Definition 9** (builders \mathcal{B}). A builder $\mu \in \mathcal{B}$ is a distinguished function whose return
 222 type is a mixin $m_{n+1} \in \mathcal{M}$ and whose parameters are the carrier type $(T : \mathbf{Type})$, mixins
 223 $\{m_1, \dots, m_n\} \in \wp(\mathcal{M})$ and a factory $f \in \mathcal{F}$, such that $requires(f) = \{m_1, \dots, m_n\}$. I.e.

1 **Definition** $\mu (T : \mathbf{Type}) \overline{(p : m T p_\sigma)^n} : f T p_1 \dots p_n \rightarrow m_{n+1} T p_{\sigma(n+1,1)} \dots p_{\sigma(n+1,q_{n+1})} := \dots$

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

8:8 Hierarchy Builder

225 ► **Definition 10** (*from*). We define $\text{from}(f, m_{n+1}) = \mu$ 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 $\text{from}(f, m)$ 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*). $\text{provides}(f) = \{m \mid \text{from}(f, m) \text{ is defined}\} \in \mathcal{P}(\mathcal{M})$ 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 $\text{requires}(m) = \text{dep}(m)$, $\text{provides}(m) = \{m\}$ and $\text{from}(m, m) =$
 236 $(\text{fun } T \ (p : m \ T \ p_\sigma)^n \ (x : m \ T \ p_1 \ \dots \ p_n) \Rightarrow x) \in \mathcal{B}$. ◀

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 $\text{requires}(\text{M.axioms}) = \text{dep}(\text{M.axioms}) = \text{provides}^*(\{f_1, \dots, f_n\})$ 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 b1 ... bk`, synthesizes terms
 243 corresponding to all the mixins that can be built from the b_i . Indeed if $b_i : f_i \ T \ \dots$, then this
 244 command creates elements of types $\text{provides}^*(\{f_1, \dots, f_k\})$. 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 bn+1 : fn+1 T .. := ...  
5 HB.instance T bn+1.  
6 ..  
7 Definition bn+k : fn+k T .. := ...  
8 HB.instance T bn+k.  
9 HB.end.
```

247 Declares the record `axioms` inside a module `F`. This record `F.axioms` depends on the mixins
 248 $\text{requires}(\text{F.axioms}) = \text{provides}^*(\{f_1, \dots, f_n\})$ 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}, \dots, b_{n+k} provided by the user in order to derive builders, so that $\text{provides}(\text{F}) =$
 252 $\text{provides}^*(\{f_{n+1}, \dots, f_{n+k}\})$.

253 It is thus necessary that $\text{requires}^*(\{f_{n+1}, \dots, f_{n+k}\}) \subseteq \text{provides}^*(\{f_1, \dots, f_n\})$.

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 ($T : \text{Type}$). The type of each field is a mixin in \mathcal{M} applied to T and, if needed, any number of other fields:

$\text{Record } c (T : \text{Type}) := \{ p_1 : m_1 T; \dots; p_i : m_i T p_{\sigma(i,1)} \dots p_{\sigma(i,q_i)}; \dots \}.$

► **Definition 14** (def , class definition). We call $\text{def}(c) \in \wp(\mathcal{M})$ the set of mixins mentioned in the fields of the class, i.e. $\{m_1, \dots, m_n\}$. 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. $\exists c \in \mathcal{C}, \text{def}(c) = \text{requires}^*(f) \cup \text{provides}^*(f),$
2. $\exists C \subseteq \mathcal{C}, \text{def}^*(C) = \text{requires}^*(f).$

Classes could also be seen as factories.

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

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 ($\text{sort} : \text{Type}$) and the second field is ($\text{class} : c \text{ sort}$) 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 $f_1 \dots f_n$ }` crafts a class $M.\text{class_of} \in \mathcal{C}$ where $\text{def}(c) = \text{provides}^*(\{f_1, \dots, f_n\})$ and the corresponding structure $M.\text{type} \in \mathcal{S}$, together with unification hints as described in Section 4.

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

3.3 Automatic inference of mixins

Since mixins may change but factories stay the same, `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, `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.

$\text{Record } f_{aux} T \overrightarrow{(p : m T p_\sigma)^n} := F_{aux} \{ \dots \}.$

Notation $f T := (f_{aux} T \dots (* p_1 \dots p_n \text{ inferred when } T \text{ is known } *)).$

Notation $F T x_1 \dots x_k := (F_{aux} T \dots (* p_1 \dots p_n \text{ inferred when } T \text{ is known } *) x_1 \dots x_k).$

Definition $b : f T := F T x_1 \dots x_n.$

294 **4** The target language: Coq with Packed, flat, Classes

295 The language of Packed Classes [13] is used directly to describe the algebraic hierarchy of the
 296 Mathematical Components library. It is based on a disciplined use of records and projections
 297 and on the possibility of extending the elaborator of Coq via the declaration of Canonical
 298 Structures [17] instances. In this section, we describe the *flat* variant of Packed Classes, the
 299 target language of `hierarchy-builder`, through the hierarchy V3 extended with semirings.

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

302 **4.1** Describing structures with records and projections

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

```

1 Module Monoid.
2 Record axioms (M : Type) : Type :=
3   Class { Monoid_of_Type_mixin : Monoid_of_Type M; }.
4 Structure type : Type := Pack { sort : Type; class : axioms sort; }.
5 End Monoid.
```

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

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

```

1 Coercion Monoid.sort : Monoid.type >-> Sortclass.
2
3 Definition zero {M : Monoid.type} : M :=
4   Monoid_of_Type.zero M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
5 Definition add {M : Monoid.type} : M -> M -> M :=
6   Monoid_of_Type.add M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
7 Lemma addrA {M : Monoid.type} : associative (@add M).
8 (* Two monoid axioms `addOr` and `addrO` are omitted. *)
```

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

```

1 Module AbelianGroup.
2 Record axioms (A : Type) : Type := Class {
3   Monoid_of_Type_mixin : Monoid_of_Type A;
4   AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid A Monoid_of_Type_mixin; }.
5 Structure type : Type := Pack { sort : Type; class : axioms sort }.
6 End AbelianGroup.
```

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

inheritance significantly. Since we do not need to care about the amount of code in automated generation, `hierarchy-builder` uses the flat variant of Packed Classes as its target language.

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.

```

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. *)

```

Generally, a coercion from a structure X to another structure Y can (and should) have the 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)).

```

4.2 Multiple inheritance

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

```

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.

```

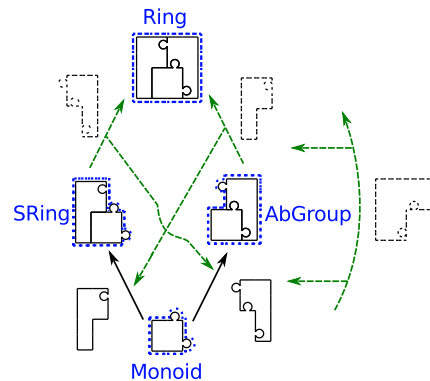


Figure 2 V4 introduces multiple inheritance. For brevity we omit the factories/builders for the upper arrows.

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

```

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.

```

8:12 Hierarchy Builder

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 $?_M : \text{Monoid.type}$ and $?_{SR} : \text{SemiRing.type}$. In order to type check the applica-
359 tion Coq has to unify $\text{Monoid.sort } ?_M$ with $\text{SemiRing.sort } ?_{SR}$, 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 $\text{SemiRing_to_Monoid} : \text{SemiRing.type} \rightarrow \text{Monoid.type}$ as canonical. With that hint Coq will
364 pick $?_M$ to be $\text{SemiRing_to_Monoid } ?_{SR}$ 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 $\text{AbelianGroup.sort } ?_{AbG} = \text{SemiRing.sort } ?_{SR}$ by instantiating
370 $?_{AbG}$ and $?_{SR}$ with $\text{Ring_to_AbelianGroup } ?_R$ and $\text{Ring_to_SemiRing } ?_R$ respectively where $?_R$ is
371 a fresh unification variable of type Ring.type . This hint can be given as follows:

```

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

```

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

375 5 The implementation of \mathcal{HB} in Coq-Elpi

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

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

```

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

```

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

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

```

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

```

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

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

```

1 kind indt-decl type. % The type of an inductive type declaration
2 type record string -> term -> string -> record-decl -> indt-decl.
3 type field bool -> string -> term -> (term -> record-decl) -> record-decl.
4 type end-record record-decl.
5 ... % constructors for non-record inductive types are omitted for brevity
6
7 external pred coq.env.add-indt i:indt-decl, o:inductive.
8 external pred coq.CS.canonical-projections i:inductive, o:list (option constant).

```

8:14 Hierarchy Builder

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

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

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

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

```
1 pred declare-structure i:gref, o:term, o:term, o:term.  
2  
3 declare-structure ClassName Structure SortProjection ClassProjection :- std.do! [  
4   StructureDeclaration =  
5     record "type" {{ Type }} "Pack" (  
6       field tt "sort" {{ Type }} s\  
7       field ff "class" (app [global ClassName, s]) _\  
8     end-record),  
9   coq.typecheck-indt-decl StructureDeclaration,  
10  coq.env.add-indt StructureDeclaration StructureName,  
11  coq.CS.canonical-projections StructureName [some SortP, some ClassP],  
12  Structure = global (indt StructureName),  
13  SortProjection = global (const SortP),  
14  ClassProjection = global (const ClassP),  
15 ].
```

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

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

```
1 pred synthesize-fields.field-for-mixin i:mixinname, o:term.  
2 pred synthesize-fields i:list mixinname, i:term, o:record-decl.  
3  
4 synthesize-fields [] _ Decl :- Decl = end-record.  
5 synthesize-fields [M|ML] T Decl :- std.do! [  
6   get-mixin-modname M ModName, Name is ModName ^ "_mixin",  
7   dep1 M Deps,  
8   std.map Deps synthesize-fields.field-for-mixin Args,  
9   Type = app[ global M, T | Args ],  
10  Decl = (field ff Name Type f\< Fields f),  
11  pi m\  
12    synthesize-fields.field-for-mixin M m =>  
13    synthesize-fields ML T (Fields m)  
14 ].
```

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

424 The second predicate proceeds by recursion on the (topologically sorted) list of mixins,
 425 and terminates when the list is empty. If the list contains a mixin M then it crafts a `Name` for
 426 it (line 6), fetches its dependencies (line 7) and finds the (previously declared) record fields
 427 holding these mixins (line 8). The `(std.map L1 P L2)` predicate relates the two lists $L1$ and
 428 $L2$ point wise using the predicate P .

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

435 6 Related work

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

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

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

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

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

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

481 **7 Conclusion**

482 In this paper we design and implement \mathcal{HB} , a high level language to describe hierarchies of
 483 algebraic structures. The implementation of \mathcal{HB} is based on the Elpi extension language for
 484 the Coq system and is available at <https://github.com/math-comp/hierarchy-builder>.
 485 The implementation amounts to approximately a thousand lines of (commented) Elpi code
 486 and tree hundred lines of Coq vernacular. It took less than one month to implement \mathcal{HB} , but
 487 its design took several years of attempts and fruitful discussions with the people we thank
 488 below.

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

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

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

506 **Acknowledgments**

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

513 — References —

- 514 1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and
515 Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: a case study
516 in functional analysis. accepted in the proceedings of IJCAR 2020, available at <https://hal.inria.fr/hal-02463336>, 2020.
517
- 518 2 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for
519 program verification using equational reasoning. In *Mathematics of Program Construction -*
520 *13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*,
521 pages 226–254, 2019. doi:10.1007/978-3-030-33636-3_9.
- 522 3 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita
523 interactive theorem prover. In *Automated Deduction - CADE-23 - 23rd International Confer-*
524 *ence on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages
525 64–69, 2011. doi:10.1007/978-3-642-22438-6_7.
- 526 4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in
527 unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel,
528 editors, *Theorem Proving in Higher Order Logics*, pages 84–98, Berlin, Heidelberg, 2009.
529 Springer Berlin Heidelberg.
- 530 5 Gerhard Betsch. On the beginnings and development of near-ring theory. In *Near-rings and*
531 *near-fields. Proceedings of the conference held in Fredericton, New Brunswick, July 18-24,*
532 *1993*, pages 1–11. Mathematics and its Applications, 336. Kluwer Academic Publishers Group,
533 Dordrecht, 1995.
- 534 6 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library
535 of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. doi:10.1007/
536 s11786-014-0181-1.
- 537 7 Jacques Carette and Russell O’Connor. Theory presentation combinators. In Johan Jeuring,
538 John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and
539 Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 202–215, Berlin, Heidelberg,
540 2012. Springer Berlin Heidelberg.
- 541 8 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD
542 thesis, Ecole Polytechnique X, 2012. URL: <https://pastel.archives-ouvertes.fr/pastel-00780446>.
543
- 544 9 The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.11*,
545 October 2020. URL: <http://coq.inria.fr>.
- 546 10 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von
547 Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25*
548 *- 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015,*
549 *Proceedings*, pages 378–388, 2015. doi:10.1007/978-3-319-21401-6_26.
- 550 11 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
551 embeddable, λProlog interpreter. In *Logic for Programming, Artificial Intelligence, and*
552 *Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015,*
553 *Proceedings*, pages 460–468, 2015. doi:10.1007/978-3-662-48899-7_32.
- 554 12 Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson
555 Education India, 1995.
- 556 13 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging
557 mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International*
558 *Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 327–342,
559 2009. doi:10.1007/978-3-642-03359-9_23.
- 560 14 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François
561 Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana
562 Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-
563 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 Kornilowicz, and Christoph Schwarzeweller. 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/S0890540113000631>, doi:<https://doi.org/10.1016/j.ic.2013.06.001>.
- 590 **23** Damien Rouhling. *Formalisation Tools for Classical Analysis – A Case Study in Control Theory*.
 591 PhD thesis, Université Côte d’Azur, 2019. URL: <https://hal.inria.fr/tel-02333396>.
- 592 **24** Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *24th ACM SIGPLAN-*
 593 *SIGACT Symposium on Principles of Programming Languages (POPL 1997), Paris, France,*
 594 *15–17 January 1997*, pages 292–301. ACM, 1997.
- 595 **25** Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation*
 596 *des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formal-*
 597 *ization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration.*
 598 *Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France,
 599 1999. URL: <https://tel.archives-ouvertes.fr/tel-00523810>.
- 600 **26** Kazuhiko Sakaguchi. Validating mathematical structures. accepted in the proceedings of
 601 IJCAR 2020, available at <https://arxiv.org/abs/2002.00620>, 2020.
- 602 **27** Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathemat-*
 603 *ical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.
- 604 **28** Enrico Tassi. Coq-Elpi, Coq plugin embedding Elpi. <https://github.com/LPCIC/coq-elpi>,
 605 2020.
- 606

A Coq reference

607

```

1 Section OperationProperties.
2
3 Variables (T : Type) (e : T) (inv : T -> T) (op : T -> T -> T) (add : T -> T -> T).
4
5 Definition left_id := forall x, op e x = x.
6 Definition right_id := forall x, op x e = x.
7
8 Definition left_inverse := forall x, op (inv x) x = e.
9 Definition right_inverse := forall x, op x (inv x) = e.
10
11 Definition commutative := forall x y, op x y = op y x.
12 Definition associative := forall x y z, op x (op y z) = op (op x y) z.
13
14 Definition left_distributive := forall x y z, op (add x y) z = add (op x z) (op y z).
15 Definition right_distributive := forall x y z, op x (add y z) = add (op x y) (op x z).
16
17 End OperationProperties.

```

B Proof of addrC

608

```

1 Lemma addrC {R : Ring.type} : commutative (@add R).
2 Proof.
3 have innerC (a b : R) : a + b + (a + b) = a + a + (b + b).
4   by rewrite -[a+b]mulr -mulrDl mulrDr !mulrDl !mulr.
5 have addKl (a b c : R) : a + b = a + c -> b = c.
6   apply: can_inj (add a) (add (opp a)) _ _ _ .
7   by move=> x; rewrite addrA addrNr addrOr.
8 have addKr (a b c : R) : b + a = c + a -> b = c.
9   apply: can_inj (add ^~ a) (add ^~ (opp a)) _ _ _ .
10  by move=> x; rewrite /= -addrA addrN addrO.
11 move=> x y; apply: addKl (x) _ _ _; apply: addKr (y) _ _ _ .
12 by rewrite -!addrA [in RHS]addrA innerC !addrA.
13 Qed.

```

C Full V4 code

609

```

1 From Coq Require Import ssreflect ssrfun ZArith.
2 From HB Require Import structures.
3
4 Declare Scope hb_scope.
5 Delimit Scope hb_scope with G.
6 Open Scope hb_scope.
7
8 (* Bottom mixin in Fig. 2. *)
9 HB.mixin Record Monoid_of_Type M := {
10   zero : M;
11   add : M -> M -> M;
12   addrA : associative add;
13   addrOr : left_id zero add;
14   addrO : right_id zero add;
15 }.
16 HB.structure Definition Monoid := { M of Monoid_of_Type M }.
17 Notation "0" := zero : hb_scope.
18 Infix "+" := (@add _) : hb_scope.
19
20 (* Bottom right mixin in Fig. 2. *)
21 HB.mixin Record AbelianGroup_of_Monoid A of Monoid A := {
22   opp : A -> A;
23   addrC : commutative (add : A -> A -> A);
24   addrNr : left_inverse zero opp add;
25 }.
26 HB.structure Definition AbelianGroup := { A of Monoid A & AbelianGroup_of_Monoid A }.

```

8:20 Hierarchy Builder

```

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   mulr1 : left_id one mul;
36   mulr1 : right_id one mul;
37   mulrDl : left_distributive mul add;
38   mulrDr : right_distributive mul add;
39   mulOr : 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   mulr1 : 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 mulOr : left_zero zero mul.
64 Proof.
65 move=> x; rewrite -[LHS]addOr addrC.
66 rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
67 by rewrite -mulrDl addOr addrC addNr.
68 Qed.
69
70 Fact mulr0 : right_zero zero mul.
71 Proof.
72 move=> x; rewrite -[LHS]addOr addrC.
73 rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
74 by rewrite -mulrDr addOr addrC addNr.
75 Qed.
76
77 Definition to_SemiRing_of_Monoid := SemiRing_of_Monoid.Build A _ mul mulrA
78   mulr1 mulr1 mulrDl mulrDr mulOr 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   mulr1 : 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]mulr1 -mulrDl mulrDr !mulrDl !mulr1.
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 addOr.
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 addrO.
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 mulr1 mulr1 mulrDl mulrDr.
131 HB.instance R to_Ring_of_AbelianGroup.
132
133 HB.end.

```