Hierarchy Builder

Cyril Cohen[1], Kazuhiko Sakaguchi[2], and Enrico Tassi[3]

[1] Inria, Université Côte d'Azur, France
Cyril.Cohen@inria.fr
[2] University of Tsukuba, Japan
sakaguchi@logic.cs.tsukuba.ac.jp
[3] Inria, Université Côte d'Azur, France
Enrico.Tassi@inria.fr

Context and Talk proposal Abstract algebra is extremely important in the mathematical vernacular and libraries of machine checked code are, nowadays, organized around hierarchies of structures. Unfortunately the language of Type Theory and the features provided by the Coq system make the construction of a hierarchy hard even for expert users. The difficulty begins with the non-orthogonal choices, between storing information as record fields or parameters, and between using type classes and canonical structures for inference. To this, one may add the concerns about performance and about the usability, by a non expert, of the final hierarchy.

$\mathcal{HB}$ gives the library designer a language to describe the building blocks of algebraic structures and to assemble them into a hierarchy. Similarly it provides the final user linguistic constructs to build instances (examples) of structures and to teach the elaborator of Coq how to take advantage of this knowledge during type inference. Finally $\mathcal{HB}$ lets the library designer improve the usability of his library by providing alternative interfaces to the primitive ones, a feature that can also be used to accommodate changes to the hierarchy without breaking user code. The details of $\mathcal{HB}$, including its compilation to a variant of the Packed Classes discipline and its implementation in the Elpi extension language for Coq, are given in [1].

In this talk we focus on how to use $\mathcal{HB}$ in a Coq development without any prior knowledge, walking slowly through the example below, describing the bricks of the language from mixins – i.e. bare bone building block of the hierarchy, packing operations and axioms – to structures.

$\mathcal{HB}$ is available for download at https://github.com/math-comp/hierarchy-builder.

Example: A hierarchy with a diamond in $\mathcal{HB}$ We first introduce the base structure of our example, the structure `CMonoid` of commutative monoids, which is defined in two phases: first its contents using `HB.mixin` and then the structure itself is declared using `HB.structure`.

```
HB.mixin Record CMonoid_of_Type A := { (* The set of axioms making A a commutative monoid. *)
  zero  : A; add   : A -> A -> A;
  addrA : associative add;  (* `add` is associative  *)
  addrC : commutative add;  (* `add` is commutative  *)
  add0r : left_id zero add; (* `zero` is a neutral element *)
}.
HB.structure Definition CMonoid := { A of CMonoid_of_Type A }. (* The structure thereof. *)
Notation "0" := zero.
Infix    "+" := add.

(* The type of the operations and axioms depend on a CMonoid.type structure. *)
Check addrC. (* ?M : CMonoid.type |- commutative (@add ?M) *)
```

We extend this hierarchy with abelian group, semi ring and ring structures, which happen to form a diamond (cf Figure 1), using only the bare minimum information: we use the same pattern as before to create the abelian group and semi-ring structures, then a ring being no more than the combination of the former structures, we do not even need to provide a mixin.

```
HB.mixin Record AbelianGrp_of_CMonoid A of CMonoid A := {
  opp   : A -> A;
  (* We can write `add` here since A is a  CMonoid   *)
  addNr : left_inverse zero opp add; (* `opp` is the additive inverse *)
}.
HB.structure Definition AbelianGrp := { A of AbelianGrp_of_CMonoid A }.
Notation "- x"   := (opp x).
Notation "x - y" := (add x (opp y)).

HB.mixin Record SemiRing_of_CMonoid A of CMonoid A := {
  one    : A;
  mul    : A -> A -> A;
  mulrA  : associative mul;  (* `mul` is associative   *)
  mul1r  : left_id one mul;  (* `one` is left neutral  *)
  mulr1  : right_id one mul; (* `one` is right neutral *)
  mulrDl : left_distributive mul add;  (* `mul` distributes over *)
  mulrDr : right_distributive mul add; (*   `add` on both sides  *)
  mul0r  : left_zero zero mul;  (* `zero` is absorbing `mul`    *)
  mulr0  : right_zero zero mul; (*   on both sides               *)
}.
HB.structure Definition SemiRing := { A of SemiRing_of_CMonoid A }.
Notation "1"  := one.
Infix    "*"  := mul.

(* We need no additional mixin to declare the Ring structure. *)
HB.structure Definition Ring := { A of SemiRing_of_CMonoid A & AbelianGrp_of_CMonoid A }.
```

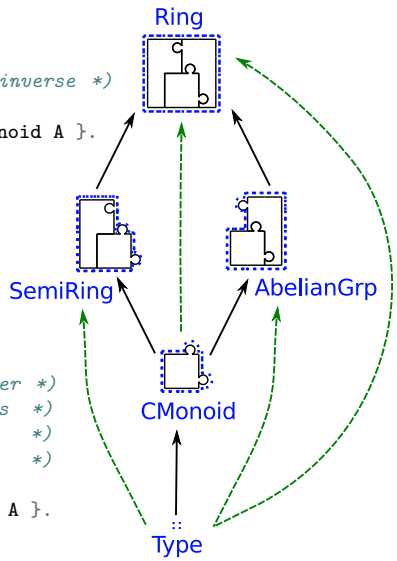

Figure 1: The hierarchy

Usage We can write abstract statements combining successive signatures:

```
(* An example statement in the signature of an Abelian group G, combining 0 and -. *)
Check forall G : AbelianGrp.type, forall x : G, x - x = 0.
(* An example statement in the signature of a Semiring S, combining 0, +, and *.  *)
Check forall S : SemiRing.type, forall x : S, x * 1 + 0 = x.
(* An example statement in the signature of a Ring R, combining -, 1 and *.  *)
Check forall R : Ring.type, forall x y : R, x * - (1 * y) = - x * y.
```

We can also equip $\mathbb{Z}$ with commutative monoid, abelian group and semi ring instances, and $\mathcal{HB}$ equips it automatically with a ring instance, because there is only one possible choice.

```
HB.instance Definition Z_CMonoid    := CMonoid_of_Type.Build Z 0%Z Z.add
  Z.add_assoc Z.add_comm Z.add_0_l.
HB.instance Definition Z_AbelianGrp := AbelianGrp_of_CMonoid.Build Z Z.opp Z.add_opp_diag_l.
HB.instance Definition Z_SemiRing   := SemiRing_of_CMonoid.Build Z 1%Z Z.mul
  Z.mul_assoc Z.mul_1_l Z.mul_1_r Z.mul_add_distr_r Z.mul_add_distr_l Z.mul_0_l Z.mul_0_r.

(* An example statement in the signature of the Z ring, combining Z, 0, +, -, 1 and * *)
Check forall x : Z, x * - (1 + x) = 0 + 1.
```

Factories For space constraints we omit the declaration of the dashed arrows in Figure 1, which are Factories, i.e. alternative ways of building the same structures from different inputs.

Factories are used to allow the insertion of intermediate structures in a hierarchy without breaking user code, as well as to implement shortcuts: e.g. the factory to build a `Ring` from a `Type` can omit the commutativity axiom of the addition, since it is implied by the other axioms.

References

[1] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi. In *FSCD*, 2020. URL: https://hal.inria.fr/hal-02478907.