Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi

Cyril Cohen *(Inria)*, Pierre Roux, Kazuhiko Sakaguchi, Enrico Tassi

JFLA 2023
February 2nd, 2023

Structures in Mathematics

Standard definition:

- A **carrier** in Set / Type,
- A set of **constants** in the carrier, and **operations**,
- Proofs of the **axioms** of the structure

Structures in Mathematics

Standard definition:

- A **carrier** in Set / Type,
- A set of **constants** in the carrier, and **operations**,
- Proofs of the **axioms** of the structure

E.g. an (additive) monoid is given by

- a carrier `T : Type`,
- a constant `zero : T` and a binary operation `add : T -> T -> T`
- three axioms:
  associativity of the addition, left and right neutrality of zero.

Dependent Type Theory (e.g. CIC, MLTT, ...)

- Mixes types and terms: `forall (n : nat), 'I_n -> 'I_n`,
- Equality as a type: `eq : forall T, T -> T -> Prop`,
  Combined with the dependent types: `forall n, n + 0 = n`
- Record Types:

```
Record R p1 .. pn : Type := MkR {
    f1 : ...
    f2 : ...
  }.
```

- The keywords `Record` and `Structure` are synonyms for Coq.
- A `Class` is also a record, but with special meta-data.

Implementations in DTT (unbundled classes) [MSCS2011]

```
Class is_monoid T (zero : T) (add : T -> T -> T) := {
    addrA : associative add;
    add0r : forall x, 0 + x = x;
    addr0 : forall x, x + 0 = x;
  }.
```

Implementations in DTT (semi-bundled classes)

```
Class is_monoid (T : Type) : Type := {
    zero   : T;
    add    : T -> T -> T;
    addrA  : associative add;
    add0r  : forall x, 0 + x = x;
    addr0  : forall x, x + 0 = x;
  }.
```

Implementations in DTT (semi-bundled classes)

```
Class is_monoid (T : Type) : Type := {
    zero  : T;
    add   : T -> T -> T;
    addrA : associative add;
    add0r : forall x, 0 + x = x;
    addr0 : forall x, x + 0 = x;
  }.
```

```
Class monoid_is_group T : is_monoid T -> Type :={
    opp   : T -> T;
    subrr : forall x, x + (- x) = 0;
    addNr : forall x, (- x) + x = 0;
  }.
```

Implementations in DTT (semi-bundled classes)

```
Class is_monoid (T : Type) : Type := {
    zero  : T;
    add   : T -> T -> T;
    addrA : associative add;
    add0r : forall x, 0 + x = x;
    addr0 : forall x, x + 0 = x;
  }.
```

```
Class is_group (T : Type) : Type := {
    zero  : T;
    add   : T -> T -> T;
    opp   : T -> T;
    addrA : associative add;
    add0r : forall x, 0 + x = x;
 (* addr0 : forall x, x + 0 = x;  (* spurious *) *)
    subrr : forall x, x + (- x) = 0;
    addNr : forall x, (- x) + x = 0;
  }.
```

Implementations in DTT (bundled record)

```
Structure monoidType : Type := {
    sort   :> Type;
    zero   : sort;
    add    : sort -> sort -> sort;
    addrA  : associative add;
    add0r  : forall x, 0 + x = x;
    addr0  : forall x, x + 0 = x;
  }.
```

Implementations in DTT (simplified packed classes)

```
Class is_monoid (T : Type) : Type := {
    zero  : T;
    add   : T -> T -> T;
    addrA : associative add;
    add0r : forall x, 0 + x = x;
    addr0 : forall x, x + 0 = x;
  }.
```

```
Structure monoidType : Type := {
    sort  :> Type;
    class : is_monoid sort;
  }.
```

Implementations in DTT (packed classes) [TPHOLs 2009]

```
Record is_monoid (T : Type) : Type := { zero ; ..}.

Structure monoidType : Type :=
  { sort :> Type;      class : is_monoid sort }.
```

```
Record monoid_is_group T : is_monoid T -> Type := ...

Record is_group (T : Type) := {
    monoid_of_group : is_monoid T;
    group_of_group  : monoid_is_group T monoid_of_group
  }.

Structure groupType : Type :=
  { sort :> Type;      class : is_group sort }.
```

Implementation in DTT (other)

Many other possibilities:

- Modules a la OCaml *(not first class in Coq!)*,
- Fully-bundled typeclasses *(bad!)*,
- Telescopes *(bad!)*,
- Records without inference *(tedious!)*,
- ...

Implementations in proof assistants

The variety of representations is out there!

- Coq/MathComp: Packed classes.
- Coq/Math-Classes: Fully unbundled records (+ special case for varieties).
- Lean3/Mathlib: Semi-bundled records.
- Agda: Bundled and semi-bundled records.
- ...

Implementations in proof assistants

The variety of representations is out there!

- Coq/MathComp: Packed classes.
- Coq/Math-Classes: Fully unbundled records (+ special case for varieties).
- Lean3/Mathlib: Semi-bundled records.
- Agda: Bundled and semi-bundled records.
- ...

Implementations in proof assistants

The variety of representations is out there!

- Coq/MathComp: Packed classes inside canonical structures.
- Coq/Math-Classes: Fully unbundled type classes
  (+ special case for varieties).
- Lean3/Mathlib: Semi-bundled type classes.
- Agda: Bundled and semi-bundled records.
- ...

Representations work hand in hand with tooling.

More than "just records"

- Coq/Mathcomp 1: canonicals
  + heavy boilerplate + validator [IJCAR K.S. paper]
- Coq/Math-Classes: type classes + boilerplate + hints
- Lean3/Mathlib: type classes + priorities + linter
- Agda: records + open and renaming directives

More than “just records”

- Coq/Mathcomp 1: canonicals
  + heavy boilerplate + validator [IJCAR K.S. paper]
- Coq/Math-Classes: type classes + boilerplate + hints
- Lean3/Mathlib: type classes + priorities + linter
- Agda: records + open and renaming directives

None of these encoding are straightforward:

- they all need expert knowledge and/or checkers/linters,
- some encodings are unnecessarily verbose,
- some known design problems might be detected too late (e.g. priority of instance, typeclass indexing, forgetful inheritance, etc)

More than "just records"

- Coq/Mathcomp 1: canonicals
  + heavy boilerplate + validator [IJCAR K.S. paper]
- Coq/Math-Classes: type classes + boilerplate + hints
- Lean3/Mathlib: type classes + priorities + linter
- Agda: records + open and renaming directives

None of these encoding are straightforward:

- they all need expert knowledge and/or checkers/linters,
- some encodings are unnecessarily verbose,
- some known design problems might be detected too late (e.g. priority of instance, typeclass indexing, forgetful inheritance, etc)

Hierarchy Builder provides a DSL!

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Requirements:

- declare a **new instance**,

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Requirements:

- declare a **new instance**,
- declare a **new structure**
  - above, below or in the middle
  - handle diamonds (e.g. monoid, group, commutative or not),
  - by amending existing code, or not,

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Requirements:

- declare a **new instance**,
- declare a **new structure**
  - above, below or in the middle
  - handle diamonds (e.g. monoid, group, commutative or not),
  - by amending existing code, or not,
- provide **several ways** to instantiate them

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Requirements:

- declare a **new instance**,
- declare a **new structure**
  - above, below or in the middle
  - handle diamonds (e.g. monoid, group, commutative or not),
  - by amending existing code, or not,
- provide **several ways** to instantiate them
- **predictability** of inferred instance,

Hierarchies in formalization

Purpose:

- **factor theorems**, using the *theory* of each structure,
- **automatically find** which structures hold on which types.

Requirements:

- declare a **new instance**,
- declare a **new structure**
  - above, below or in the middle
  - handle diamonds (e.g. monoid, group, commutative or not),
  - by amending existing code, or not,
- provide **several ways** to instantiate them
- **predictability** of inferred instance,
- **robustness** of user code with regard to *new declarations*.

Hierarchy Builder in two bullets

1. **Hierarchy Builder provides a DSL to generate and extend a hierarchy from minimal input.**

2. **Hierarchy Builder lets you amend a hierarchy without breaking your code.**

Hierarchy Builder adopts the point of view that Type Theory is an assembly language, and takes care of generating structures in a uniform way across whole sets of libraries.

Hierarchy Builder in practice

- Hierarchy Builder generates/extends a hierarchy using MATHEMATICAL COMPONENTS packed class methodology.
- Hierarchy Builder enforces a discipline of *mixins* and *factories* to make client code robust to hierarchy changes.
- Hierarchy Builders lets us encode built-in safety measures (e.g. detection of overlapping instances)

Structures relating to each other

Examples:

- Monoid $\leftarrow$ Group $\leftarrow$ Ring $\leftarrow$ Field $\leftarrow$ ...
- Normed Space $\rightarrow$ Metric Spaces $\rightarrow$ Topological Spaces $\rightarrow$ ...

Structures relating to each other

Examples:

- Monoid $\leftarrow$ Group $\leftarrow$ Ring $\leftarrow$ Field $\leftarrow$ ...
- Normed Space $\rightarrow$ Metric Spaces $\rightarrow$ Topological Spaces $\rightarrow$ ...

Going through arrows must be automated.

Structures relating to each other

Examples:

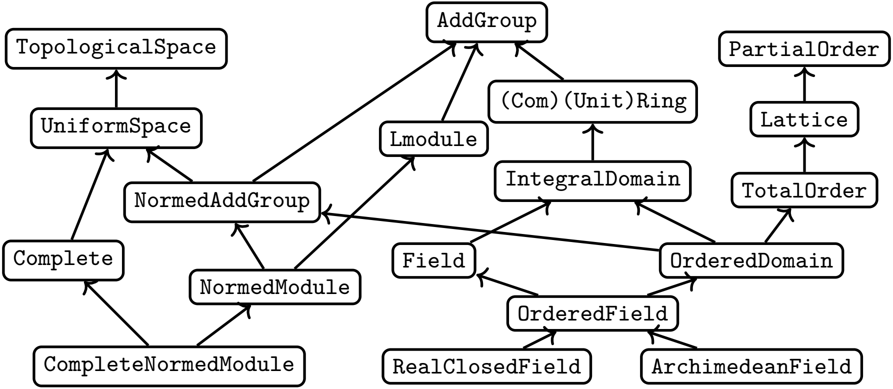
- Monoid $\leftarrow$ Group $\leftarrow$ Ring $\leftarrow$ Field $\leftarrow$ ...
- Normed Space $\rightarrow$ Metric Spaces $\rightarrow$ Topological Spaces $\rightarrow$ ...

Going through arrows must be automated.

Arrows represent both

- Extensions: add operations, axioms or combine structures
- Entailment/Induction/Deduction/Generalization.

More examples [IJCAR]



"Calculus" structures

"Algebraic" structures

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)

Structure entailment

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),

Structure entailment

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),
- **Non-robust** when adding new intermediate structures,

Structure entailment

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),
- **Non-robust** when adding new intermediate structures,

Structure entailment

- **Flexible**: no need to cut structures into small bits,

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),
- **Non-robust** when adding new intermediate structures,

Structure entailment

- **Flexible**: no need to cut structures into small bits,
- **Robust**: we can fix operations and axioms once and for all.

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),
- **Non-robust** when adding new intermediate structures,

Structure entailment

- **Flexible**: no need to cut structures into small bits,
- **Robust**: we can fix operations and axioms once and for all.
- **Not suitable for inference**: Major breakage when arbitrary entailment is automatic. (cf IJCAR *Competing Inheritance Paths in Dependent Type Theory)*

Structure extension vs Structure entailment

Structure extension

- **Compositional**: no need to start from scratch every time. (E.g. the product of two groups is a group)
- **Noisy** internal definition of a structure. (E.g. defining a commutative monoid from a monoid, one gets an unnecessary axiom),
- **Non-robust** when adding new intermediate structures,

Structure entailment

- **Flexible**: no need to cut structures into small bits,
- **Robust**: we can fix operations and axioms once and for all.
- **Not suitable for inference**: Major breakage when arbitrary entailment is automatic. (cf IJCAR *Competing Inheritance Paths in Dependent Type Theory)*

HB Design

The best of two the worlds:

- **Extension**, through *mixins* for **automatic inference**
- **Entailment**, through *factories* for **smart instantiation**

HB Design

The best of two the worlds:

- **Extension**, through *mixins* for **automatic inference**
- **Entailment**, through *factories* for **smart instantiation**

Five primitives:

```
1. HB.mixin Record <mixin name> T of <dependencies> := {..}.

2. HB.factory Record <factory name> T of <dependencies> := {..}.
3. HB.builders Context T (f : <factory name> T). ... HB.end.

4. HB.structure Definition <structure name> :=
   { T & <dependencies>}
5. HB.instance Definition <name> : <axioms name> <type> := ...
```

HB Design

The best of two the worlds:

- **Extension**, through *mixins* for **automatic inference**
- **Entailment**, through *factories* for **smart instantiation**

Five primitives:

1. HB.mixin Record <mixin name> T of <dependencies> := {..}.

2. HB.factory Record <factory name> T of <dependencies> := {..}.
3. HB.builders Context T (f : <factory name> T). ... HB.end.

4. HB.structure Definition <structure name> :=
   { T & <dependencies>}
5. HB.instance Definition <name> : <axioms name> <type> := ...

see https://github.com/math-comp/hierarchy-builder

A very short example

https:

//github.com/math-comp/hierarchy-builder/tree/master/examples/GReTA_talk

```
HB.mixin Record is_monoid (M : Type) := {
  zero   : M;
  add    : M -> M -> M;
  addrA  : associative add; (* add is associative. *)
  add0r  : forall x, 0 + x = x; (* zero is neutral *)
  addr0  : forall x, x + 0 = x; (*         wrt add. *)
}.
HB.structure Definition Monoid := { M of is_monoid M }.

HB.instance Definition Z_is_monoid : is_monoid Z
  := is_monoid.Build Z 0%Z Z.add
    Z.add_assoc Z.add_0_l Z.add_0_r.
```

Breaking down monoid

We split the monoid structure into a semi-group and a monoid

```
HB.mixin Record is_semigroup (S : Type) := {
  add    : S -> S -> S;
  addrA  : associative add;
}.
HB.structure Definition SemiGroup :=
  { S of is_semigroup S }.
```

```
HB.mixin Record semigroup_is_monoid (M : Type)
    of is_semigroup M := {
  zero  : M;
  add0r : forall x, 0 + x = x;
  addr0 : forall x, x + 0 = x;
}.
HB.structure Definition Monoid :=
  { M of is_semigroup M & semigroup_is_monoid M }.
```

But we must provide `is_monoid` again.

Recovering the lost mixin (is_monoid)

It becomes a *factory* with the **exact** same contents as before

```
HB.factory Record is_monoid (M : Type) := {
  zero  : M;
  add   : M -> M -> M;
  addrA : associative add;
  add0r : forall x, 0 + x = x;
  addr0 : forall x, x + 0 = x;
}.
HB.builders Context (M : Type) (f : is_monoid M).
  HB.instance Definition is_monoid_semigroup
    : is_semigroup M := ... (* trivial *)
  HB.instance Definition is_monoid_monoid
    : monoid_of_semigroup M := ... (* trivial *)
HB.end
```

Factories can only be used at instantiation time:

```
HB.instance Definition Z_is_monoid : is_monoid Z := ...
```

Why use HB?

- High-level commands to declare structures and instances, easy to use.
- Predictable outcome of inference,
- Takes into account the evolution of knowledge
  - which is formalized, and
  - which the user has.

  The two knowledge do not need to be correlated.
- Robustness with regard to new declaration *and even changes of internal implementation*.

Why use HB?

- High-level commands to declare structures and instances, easy to use.
- Predictable outcome of inference,
- Takes into account the evolution of knowledge
  - which is formalized, and
  - which the user has.

  The two knowledge do not need to be correlated.
- Robustness with regard to new declaration *and even changes of internal implementation*.
- One also can envision changing the target representation, the design pattern at use, without changing the surface language and declarations.

Applications of Hierarchy Builder

- Mathcomp `2.0+alpha1`
  Porting the Mathematical Components library to HB
  Reynald Affeldt, Xavier Allamigeon, Yves Bertot, Quentin Canu, CC, Pierre Roux, Kazuhiko Sakaguchi, Enrico Tassi, Laurent Théry, Anton Trunov.
  `https://hal.inria.fr/hal-03463762/` and
  `https://github.com/math-comp/math-comp/pull/733`

- Mathcomp Analysis (released versions)
  cf `https://github.com/math-comp/analysis`

- Monae: Monadic effects and equational reasoning in Coq
  cf `https://github.com/affeldt-aist/monae`

Applications of Hierarchy Builder

- Mathcomp `2.0+alpha1`
  Porting the Mathematical Components library to HB
  Reynald Affeldt, Xavier Allamigeon, Yves Bertot, Quentin Canu, CC, Pierre Roux, Kazuhiko Sakaguchi, Enrico Tassi, Laurent Théry, Anton Trunov.
  `https://hal.inria.fr/hal-03463762/` and
  `https://github.com/math-comp/math-comp/pull/733`

- Mathcomp Analysis (released versions)
  cf `https://github.com/math-comp/analysis`

- Monae: Monadic effects and equational reasoning in Coq
  cf `https://github.com/affeldt-aist/monae`