Hierarchy Builder in the Rocq proof assistant

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

WP4 Orsay
September 15th, 2025

Representing mathematical structures in Type Theory

Provide a representation for mathematical objects.

Two extremes:

- a mathematical object is represented by several pieces,

Representing mathematical structures in Type Theory

Provide a representation for mathematical objects.

Two extremes:

- a mathematical object is represented by several pieces, e.g.
  - a group is a set, a neutral, a binary operation etc.
  - a measurable space is a set, a distinguished set of sets, closed under complement and countable unions and intersections.

Representing mathematical structures in Type Theory

Provide a representation for mathematical objects.

Two extremes:

- a mathematical object is represented by several pieces, e.g.
  - a group is a set, a neutral, a binary operation etc.
  - a measurable space is a set, a distinguished set of sets, closed under complement and countable unions and intersections.
- a mathematical object is represented by a single piece,

Representing mathematical structures in Type Theory

Provide a representation for mathematical objects.

Two extremes:

- a mathematical object is represented by several pieces, e.g.
  - a group is a set, a neutral, a binary operation etc.
  - a measurable space is a set, a distinguished set of sets, closed under complement and countable unions and intersections.
- a mathematical object is represented by a single piece, e.g.
  - a group is an element of `groupType`,
  - a measurable space is an element of `measurableType`

Representing mathematical structures in Type Theory

Two extremes:

- a mathematical object is represented by several pieces, or
- a mathematical object is represented by a single piece

Proper regroupments may lead to more concisness, e.g.

- `poly : ringType -> ringType`, instead of
- `poly : forall R : Type, (R -> bool) -> Type`,
  `poly_add : forall R, (R -> R -> R) -> (poly R -> poly R -> poly R)`. etc.

Representing mathematical structures in Type Theory

Two extremes:

- a mathematical object is represented by several pieces, or
- a mathematical object is represented by a single piece

Proper regroupments may lead to more concisness, e.g.

- `poly : ringType -> ringType`, instead of
- `poly : forall R : Type, (R -> bool) -> Type`,
  `poly_add : forall R, (R -> R -> R) -> (poly R -> poly R -> poly R)`. etc.

or less, e.g.

- `Z : Type`, `Z_group : groupType`, `Z_ring : ringType`, etc.
- `prod T T : Type`, `prod_group G G : groupType`, `prod_ring R R : ringType`, etc.

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.

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 Rocq!)*,
- Fully-bundled typeclasses *(bad!)*,
- Telescopes *(bad!)*,
- Records without inference *(tedious!)*,
- ...

Implementations in proof assistants

The variety of representations is out there!

- ROCQ/MATHCOMP: Packed classes.
- ROCQ/MATH-CLASSES: Fully unbundled records (+ special case for varieties).
- LEAN/MATHLIB: Semi-bundled records.
- AGDA: Bundled and semi-bundled records.
- ...

Implementations in proof assistants

The variety of representations is out there!

- ROCQ/MATHCOMP: Packed classes.
- ROCQ/MATH-CLASSES: Fully unbundled records

  (+ special case for varieties).
- LEAN/MATHLIB: Semi-bundled records.
- AGDA: Bundled and semi-bundled records.
- ...

Implementations in proof assistants

The variety of representations is out there!

- ROCQ/MATHCOMP: Packed classes inside canonical structures.
- ROCQ/MATH-CLASSES: Fully unbundled type classes (+ special case for varieties).
- LEAN/MATHLIB: Semi-bundled type classes.
- AGDA: Bundled and semi-bundled records.
- ...

Representations work hand in hand with tooling.

More than "just records"

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

More than "just records"

- Rocq/Mathcomp: canonicals
  + heavy boilerplate + validator [IJCAR K.S. paper]
- Rocq/Math-Classes: type classes + boilerplate + hints
- Lean/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"

- Rocq/Mathcomp: canonicals
  + heavy boilerplate + validator [IJCAR K.S. paper]
- Rocq/Math-Classes: type classes + boilerplate + hints
- Lean/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 and non-forgetful inheritance)

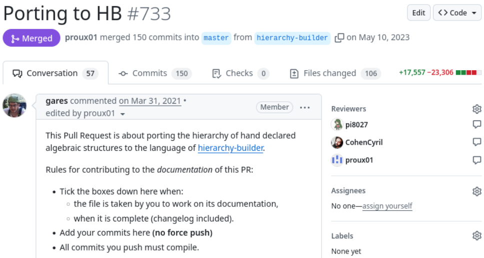
Hierarchy Builder in practice

- Hierarchy Builder generates/extends a hierarchy using Mathematical Components packed class methodology. ... but this is changing!
- 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 and non-forgetful inheritance)

Applications of Hierarchy Builder

- Mathcomp $\geq$ `2.0`
  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
  cf `https://github.com/math-comp/analysis`
- Monae: Monadic effects and equational reasoning in Rocq
  cf `https://github.com/affeldt-aist/monae`
- ...

Porting the Mathematical Components library



Porting the Mathematical Components library to Hierarchy Builder

Reynald Affeldt[3], Xavier Allamigeon[4], Yves Bertot[1], Quentin Canu[4], Cyril Cohen[1], Pierre Roux[6], Kazuhiko Sakaguchi[2], Enrico Tassi[1], Laurent Théry[1], and Anton Trunov[5]

[1] Université Côte d'Azur, Inria, France
[2] University of Tsukuba, Japan
[3] National Institute of Advanced Industrial Science and Technology (AIST), Japan
[4] Inria, CMAP, CNRS, Ecole Polytechnique, Institut Polytechnique de Paris, France
[5] Zilliqa Research
[6] ONERA / DTIS, Université de Toulouse, France

10 people, 2 weeks, 140kLOC

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

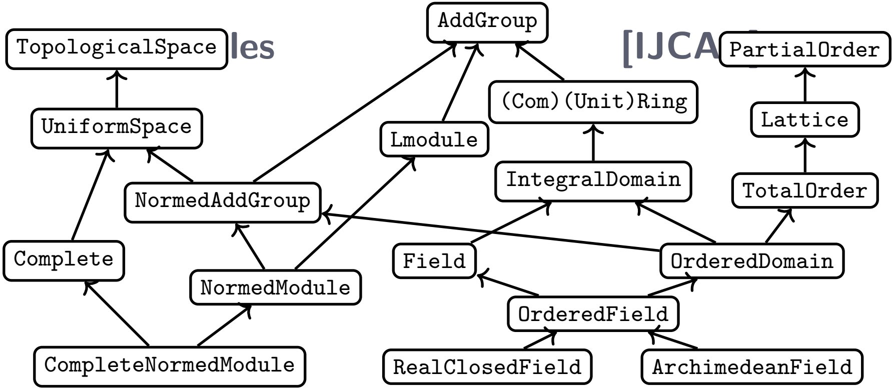
Examples:

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

Going through arrows must be automated.

Arrows represent both

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



"Calculus" structures

"Algebraic" structures

Structure extension vs Structure entailment

Structure extension

Structure entailment

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

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 : semigroup_is_monoid M := ... (* trivial *)
HB.end
```

Factories can only be used at instantiation time:

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

Measurable spaces

We may define a measurable space as follows:

```
HB.mixin Record isMeasurable T := {
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableC : forall A, measurable A -> measurable (~` A) ;
  measurable_bigcup : forall F : (set T)^nat, (forall i, measurable (F i)) ->
    measurable (\bigcup_i (F i))
}.

#[short(type="measurableType")]
HB.structure Definition Measurable := {T of isMeasurable T }.
```

Measurable spaces (modified)

But we need to

```
HB.factory Record isMeasurable T := {
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableC : forall A, measurable A -> measurable (~` A) ;
  measurable_bigcup : forall F : (set T)^nat, (forall i, measurable (F i)) ->
    measurable (\bigcup_i (F i))
}.

HB.builders Context T of isMeasurable T.
(* ... *)
HB.end.

#[short(type="measurableType")]
HB.structure Definition Measurable := {T of isMeasurable T }.
```

Semiring and rings of sets

So that we can introduce semirings of sets and rings of set

```
HB.mixin Record isSemiRingOfSets T := {
  measurable : set (set T) ;
  measurable0 : measurable set0 ;
  measurableI : setI_closed measurable;
  semi_measurableD : semi_setD_closed measurable;
}.

#[short(type="semiRingOfSetsType")]
HB.structure Definition SemiRingOfSets := {T of isSemiRingOfSets T}.

HB.mixin Record SemiRingOfSets_isRingOfSets T of SemiRingOfSets T :=
  { measurableU : @setU_closed T measurable }.

#[short(type="ringOfSetsType")]
HB.structure Definition RingOfSets :=
  {T of SemiRingOfSets T & SemiRingOfSets_isRingOfSets T }.
```

A hierarchy of measures

We also have a hierarchy of *functions* on measurable spaces:

```
HB.mixin Record isContent (T : semiRingOfSetsType) (R : numFieldType)
    (mu : set T -> \bar R) := {
  measure_ge0 : forall x, 0 <= mu x ;
  measure_semi_additive : semi_additive mu
}.
#[short(type=content)]
HB.structure Definition Content (T : semiRingOfSetsType) (R : numFieldType) :=
  { mu & isContent T R mu }.

HB.mixin Record Content_isMeasure (T : semiRingOfSetsType)
  (R : numFieldType) (mu : set T -> \bar R) of Content mu := {
measure_semi_sigma_additive : semi_sigma_additive mu }.

#[short(type=measure)]
HB.structure Definition Measure (T : semiRingOfSetsType) (R : numFieldType) :=
{mu of Content mu & Content_isMeasure T R mu }.
```

Upcoming contribution: wrapping

E.g defining measure spaces.

Upcoming contribution: wrapping

E.g defining measure spaces.

```
HB.mixin Record hasMeasure R T := { meas : T -> R }.

HB.structure Record measureType :=
  { T of Measurable T & hasMeasure R T & Measure meas }.
```

Thanks to Matteo Calosci and Enrico Tassi

Upcoming contribution: typeclass-like inference

The current target of HB is *Canonical structures*.
This forces the following style:

```
forall R : ringType, x + y = 0
```

Upcoming contribution: typeclass-like inference

The current target of HB is *Canonical structures*.
This forces the following style:

```
forall R : ringType, x + y = 0
```

However HB defines both the class `Ring R` and the structure `ringType`.

Upcoming contribution: typeclass-like inference

The current target of HB is *Canonical structures*.
This forces the following style:

```
forall R : ringType, x + y = 0
```

However HB defines both the class `Ring R` and the structure `ringType`.

Upcoming contribution: typeclass-like inference

The current target of HB is *Canonical structures*.
This forces the following style:

```
forall R : ringType, x + y = 0
```

However HB defines both the class `Ring R` and the structure `ringType`.

Soon there it should support simultaneously Structures and Typeclasses styles.

```
forall R `{Ring R}, x + y = 0
```

Meta programming in Rocq-ELPI

Rocq-ELPI turned out to be a very comfortable meta-programming language for this (approx. 5000 loc).

Meta programming in Rocq-ELPI

Rocq-ELPI turned out to be a very comfortable meta-programming language for this (approx. 5000 loc).

Elpi **is a programming language,** [LPAR-20]

- prolog-like: **programs and data are clauses**
- with binders, unification and constraints

Meta programming in Rocq-ELPI

Rocq-ELPI turned out to be a very comfortable meta-programming language for this (approx. 5000 loc).

Elpi **is a programming language,** [LPAR-20]

- prolog-like: **programs and data are clauses**
- with binders, unification and constraints
- capable of representing Rocq terms in HOAS, its typing judgements, evaluation and unification.

Meta programming in Rocq-ELPI

Rocq-ELPI turned out to be a very comfortable meta-programming language for this (approx. 5000 loc).

Elpi **is a programming language,** [LPAR-20]

- prolog-like: **programs and data are clauses**
- with binders, unification and constraints
- capable of representing Rocq terms in HOAS, its typing judgements, evaluation and unification.

Rocq-Elpi **is a plugin for** Rocq that lets one use Elpi as a meta-programming language,

Meta programming in ROCQ-ELPI

ROCQ-ELPI turned out to be a very comfortable meta-programming language for this (approx. 5000 loc).

ELPI **is a programming language,** [LPAR-20]

- prolog-like: **programs and data are clauses**
- with binders, unification and constraints
- capable of representing ROCQ terms in HOAS, its typing judgements, evaluation and unification.

ROCQ-ELPI **is a plugin for** ROCQ that lets one use ELPI as a meta-programming language, in particular

- one can write new commands and tactics,
- one can add new definitions, inductive, sections, modules, etc to the environment,
- one can maintain databases across ROCQ files

Two main HB databases

- The predicate `from` stores an association between a factory `F`, a mixin `M` and the term `B` that can be used to build mixin `M` from factory `F`.

```
pred from o:factoryname, o:mixinname, o:term.
```

- The predicate `factory-requires` stores an association between a factory and a list of mixins that are pre-requisites to inhabiting this factory.

```
pred factory-requires o:factoryname, o:list mixinname.
```

e.g. `monoid_is_group T` has the prerequisite that `T` is a monoid.

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*.
- We envision changing the target representation, the design pattern at use, without changing the surface language and declarations.

Thanks! Questions?