Validating Mathematical Structures

Kazuhiko Sakaguchi

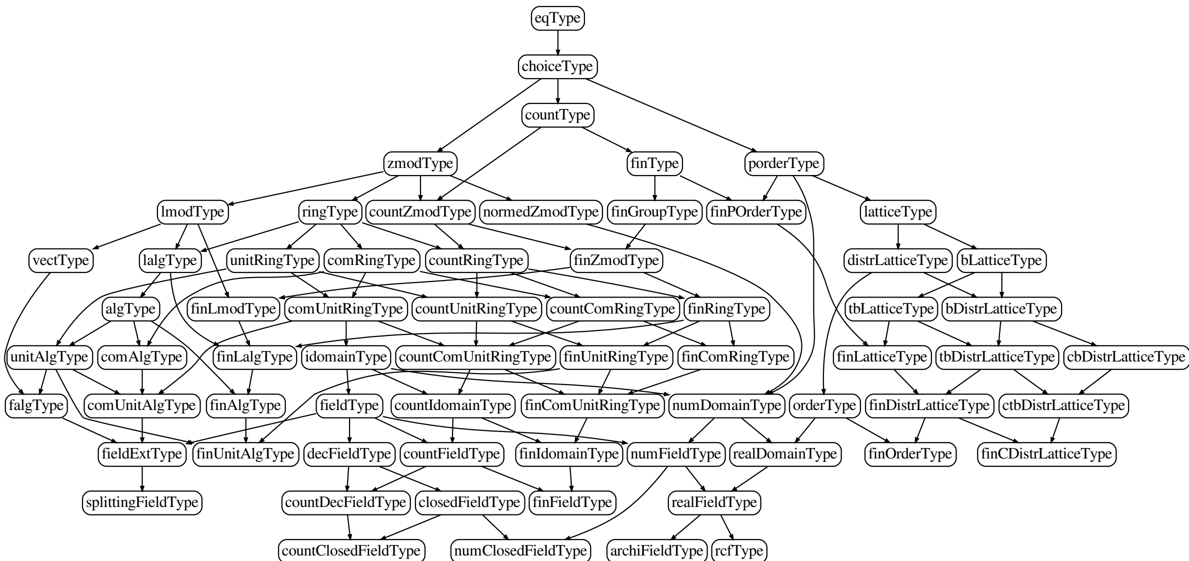
University of Tsukuba

IJCAR 2020

Packed classes [Garillot et al. 2009; Garillot 2011]

- Hierarchies of mathematical structures are a key ingredient of modern formalizations of mathematics.
- Packed classes methodology is a generic design pattern to define and combine mathematical structures using dependent records, which supports:
  - multiple inheritance,
  - maximal sharing of notations and theories, and
  - automated structure inference using canonical structures [Mahboubi et al. 2013] or unification hints [Asperti et al. 2009] .
- It has been successfully used in Mathematical Components and the formal proof of the Odd Order Theorem [Gonthier et al. 2013] .

The hierarchy of mathematical structures in MathComp



Issues of packed classes

Packed classes are hard to master for library designers and requires a **substantial amount of work** to maintain as libraries evolve.

- Structure inference requires **quadratically many** implicit coercions and unification hints for the number of structures.
  - MathComp 1.11.0: 67 structures, 705 coercions, and 982 unification hints.
- To insert a new structure in the middle of a hierarchy, the **changes required are not local**. All the structures that inherit from the new structure have to be changed.

Our solution

We indentify two hierarchy invariants in packed classes, and propose checking algorithms and tools for them to address the issues.

- The first invariant (coherence) concerning implicit coercions ensures modularity of reasoning, and is not specific to packed classes and useful for other representations such as [The mathlib Community 2020] .
- The second invariant (well-formedness) concerning unification hints ensures predictability of inference, and is specific to packed classes.
- Our checking tools are implemented only for Coq, but their methodology should be applicable to other provers.

The running example

A minimal hierarchy with multiple inheritance

Type

$0, +$

Monoids

Semirings

Groups

$1, \times$

$-$

(additive inverse)

Rings

less axioms,
smaller structures

more axioms,
larger structures

How to define structures? - Monoids

```
Module Monoid.
 (* A mixin gathers operators and axioms newly introduced by a structure.*)
 Record mixin_of (A : Type) := Mixin { zero : A; add : A → A → A; .. }.

 (* A class assembles all the mixins of superclasses of the structure. *)
 Record class_of (A : Type) := Class { mixin : mixin_of A }.

 (* A structure bundles a carrier (sort : Type) and its class instance. *)
 Structure type := Pack { sort : Type; class : class_of sort }.
End Monoid.
```

How to define structures? - Monoids

```
Module Monoid.
 (* A mixin gathers operators and axioms newly introduced by a structure.*)
 Record mixin_of (A : Type) := Mixin { zero : A; add : A → A → A; .. }.

 (* A class assembles all the mixins of superclasses of the structure. *)
 Record class_of (A : Type) := Class { mixin : mixin_of A }.

 (* A structure bundles a carrier (sort : Type) and its class instance. *)
 Structure type := Pack { sort : Type; class : class_of sort }.
End Monoid.
```

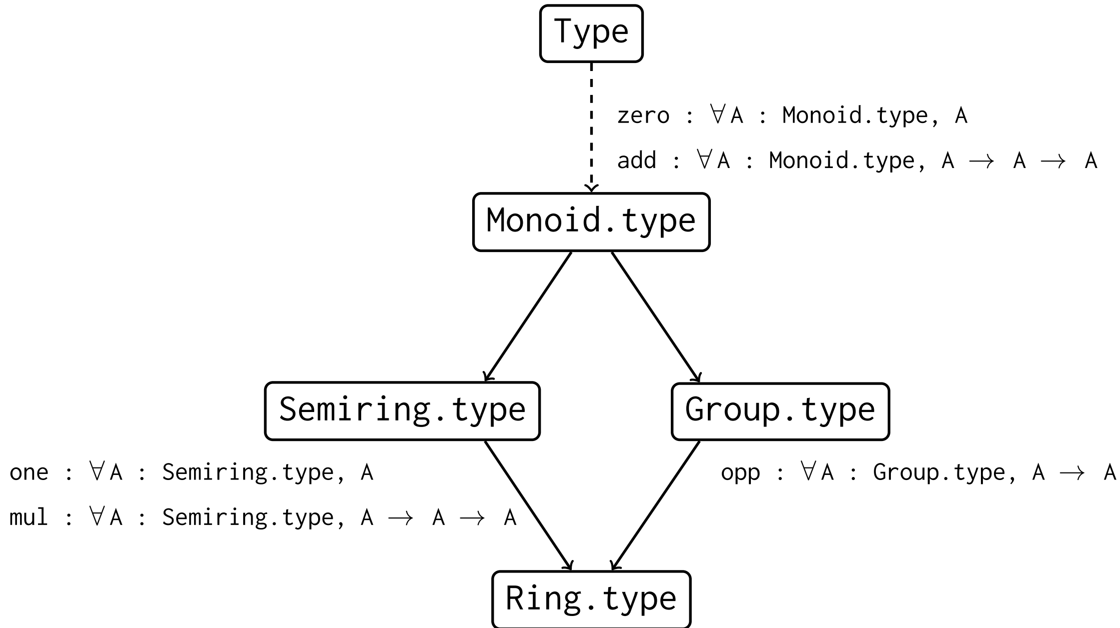
```
zero : ∀A : Monoid.type, Monoid.sort A,
add  : ∀A : Monoid.type, Monoid.sort A → Monoid.sort A → Monoid.sort A.
```

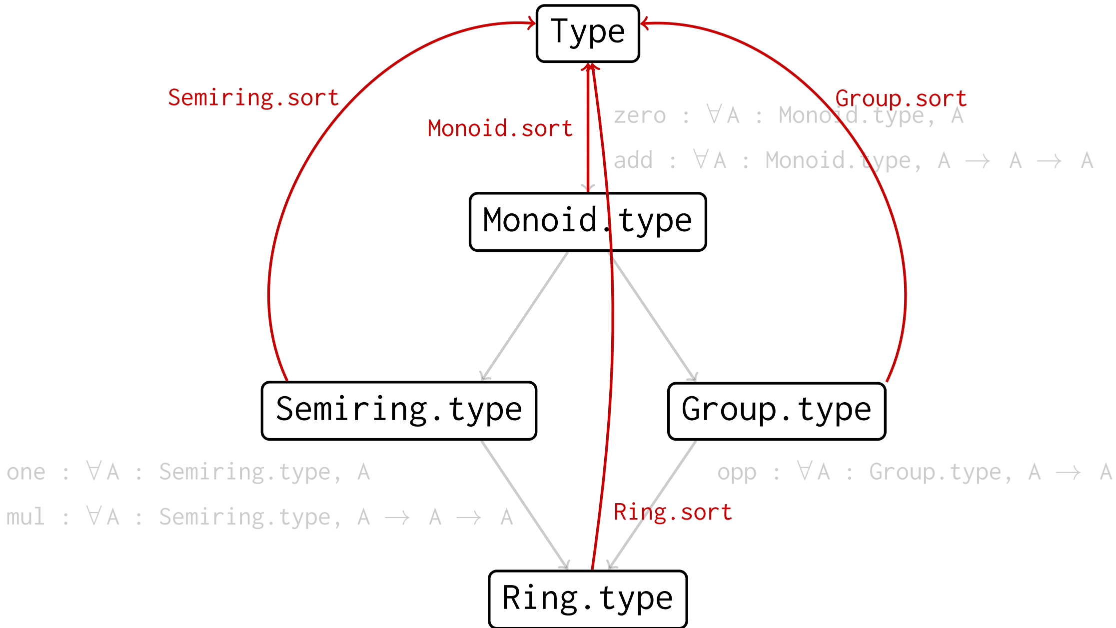
How to define structures? - Monoids

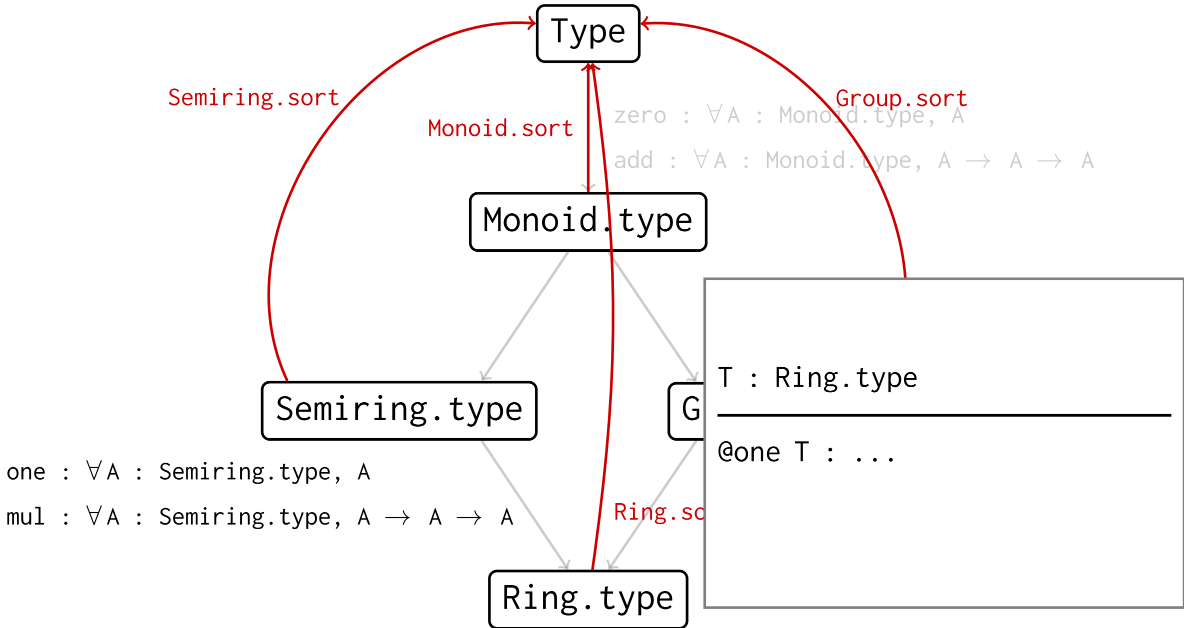
```
Module Monoid.
 (* A mixin gathers operators and axioms newly introduced by a structure.*)
 Record mixin_of (A : Type) := Mixin { zero : A; add : A → A → A; .. }.

 (* A class assembles all the mixins of superclasses of the structure. *)
 Record class_of (A : Type) := Class { mixin : mixin_of A }.

 (* A structure bundles a carrier (sort : Type) and its class instance. *)
 Structure type := Pack { sort : Type; class : class_of sort }.
End Monoid.

Coercion Monoid.sort : Monoid.type >-> Sortclass.

zero : ∀A : Monoid.type, A,
add  : ∀A : Monoid.type, A → A → A.
```

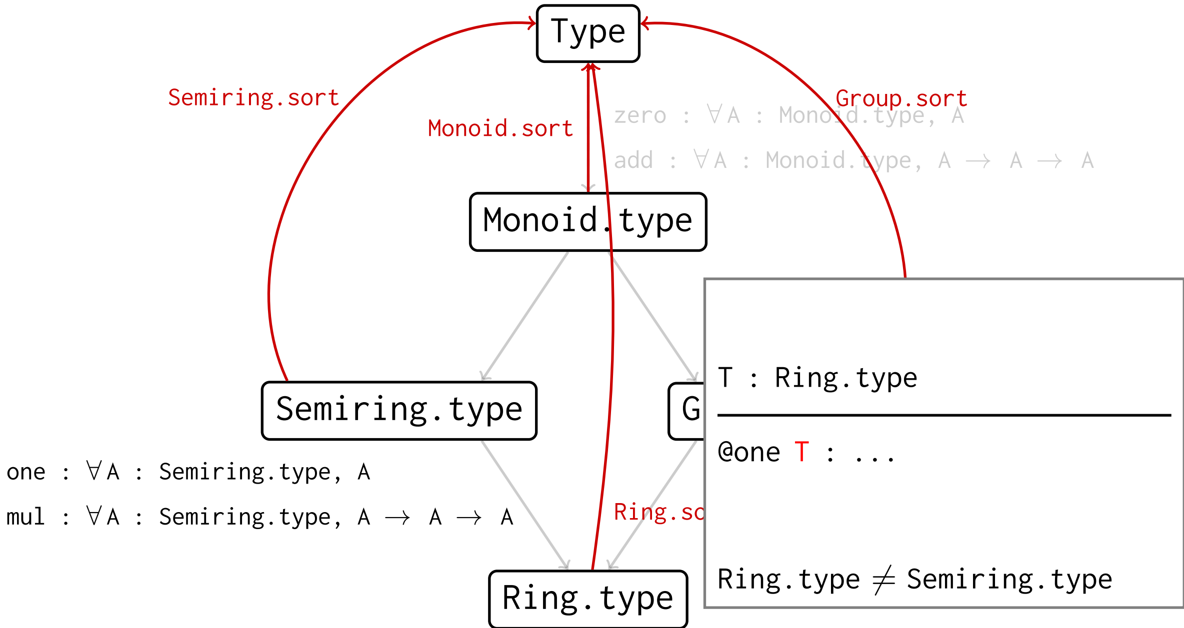
How to define structures? - Semirings

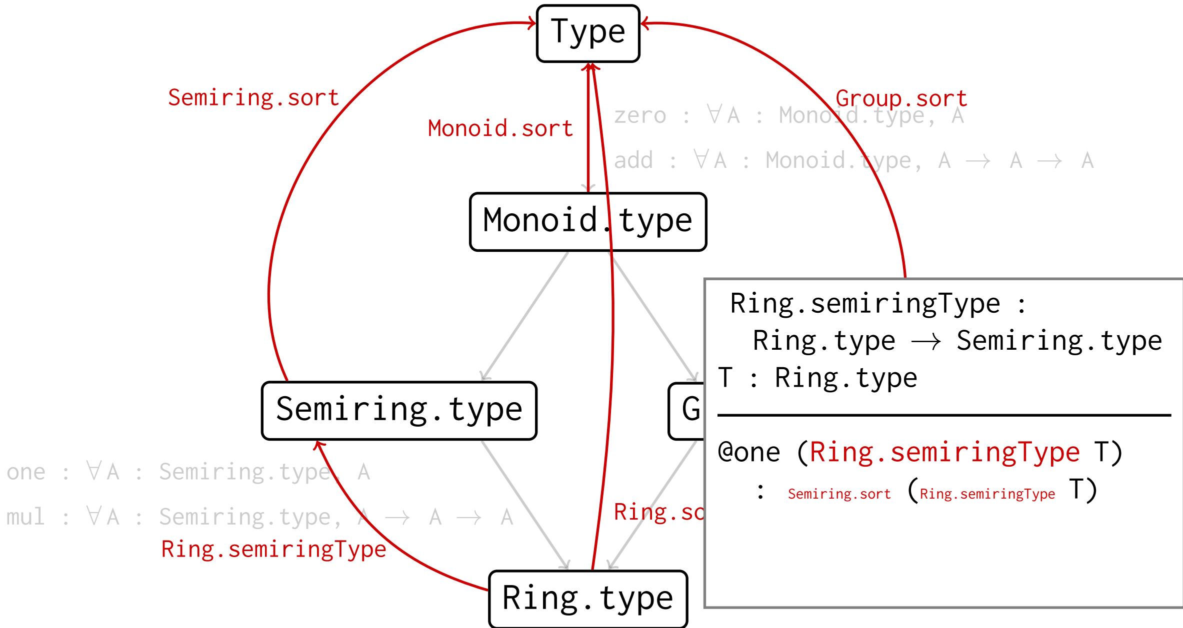
```
Module Semiring.
 Record mixin_of (A : Monoid.type) := Mixin {
   one : A; mul : A → A → A; .. mulDl : left_distributive mul add; .. }.

 Record class_of (A : Type) := Class {
   base : Monoid.class_of A;
   mixin : mixin_of (Monoid.Pack A base) }.

 Structure type := Pack { sort : Type; class : class_of sort }.
End Semiring.

Coercion Semiring.sort : Semiring.type >-> Sortclass.

one  : ∀A : Semiring.type, A,
mul  : ∀A : Semiring.type, A → A → A.
```

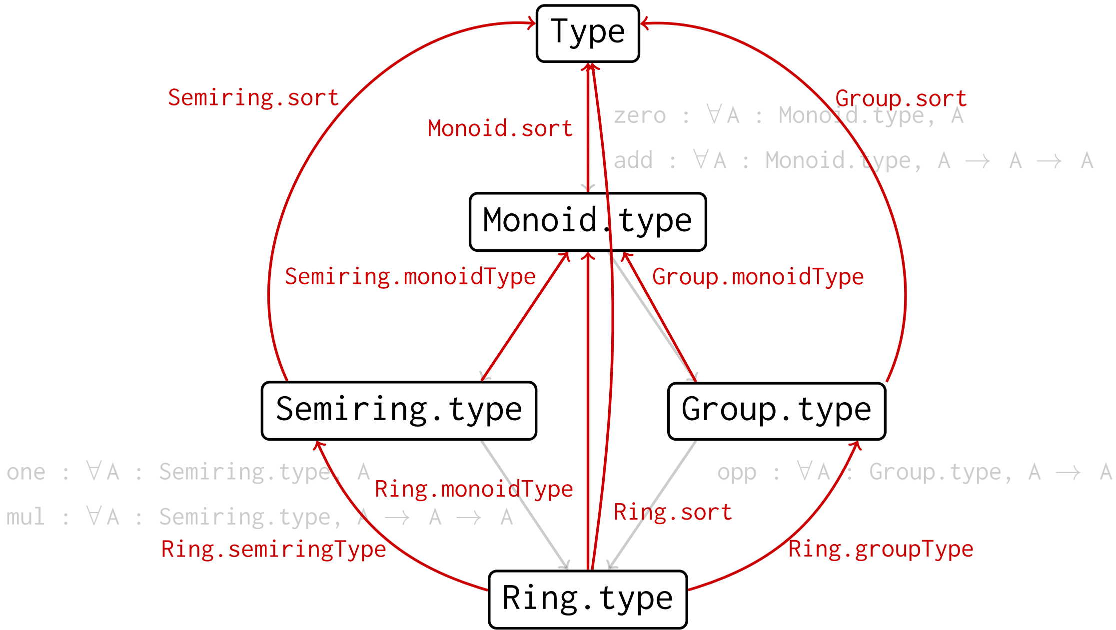

Type
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
Group.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
opp : ∀A : Group.type, A → A
Ring.type


Type
Semiring.sort
Monoid.sort
Group.sort
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
Group.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
opp : ∀A : Group.type, A → A
Ring.sort
Ring.type


Type
Semiring.sort
Monoid.sort
Group.sort
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
G
T : Ring.type
@one T : ...
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
Ring.so
Ring.type


Type
Semiring.sort
Monoid.sort
Group.sort
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
Ring.type
T : Ring.type
@one T : ...
Ring.type ≠ Semiring.type


Type
Semiring.sort
Monoid.sort
Group.sort
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
G
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
Ring.semiringType
Ring.so
Ring.type
Ring.semiringType :
Ring.type → Semiring.type
T : Ring.type
@one (Ring.semiringType T)
: Semiring.sort (Ring.semiringType T)


Type
Semiring.sort
Monoid.sort
Group.sort
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.monoidType
Group.monoidType
Semiring.type
Group.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
opp : ∀A : Group.type, A → A
Ring.monoidType
Ring.sort
Ring.semiringType
Ring.groupType
Ring.type

Implicit coercions

- If a structure `B` (transitively) inherits from a structure `A`, an implicit coercion `B >-> A` should be declared.
  - Transitive ones can be automatically computed by Coq, but we declare them explicitly to mitigate performance issues of type inference.
- Multiple inheritance makes coercion paths ambiguous, e.g.,

```
[Ring.monoidType] : Ring.type >-> Monoid.type.
[Ring.groupType; Group.monoidType] : Ring.type >-> Monoid.type.
[Ring.semiringType; Semiring.monoidType] : Ring.type >-> Monoid.type.
```

- These ambiguous paths should be convertible with each other, otherwise, the hierarchy is broken,
e.g., it may prevent us to prove $\forall R\,(x\,y : R), (-x) \times y = -(x \times y)$ by reporting type mismatch $R \not\equiv R$.

Coherence of coercion graphs

(Definitional) equality of inheritance paths is also known as *coherence*, and is a general interest in dependent type theories with inheritance:

Definition (Coherence [Barthe 1996, Sect. 3.2] [Saïbi 1997, Sect. 7])

An inheritance graph is coherent if and only if the following two conditions hold:

1. For any circular inheritance path $p : C \rightarrowtail C$, $p\,x \equiv x$, and
2. For any two inheritance paths $p, q : C \rightarrowtail D$, $p\,x \equiv q\,x$,

where x is a fresh variable of class C.

In Coq 8.11, we relaxed the condition of ambiguous paths to report only paths that break the coherence.

Coherence checking

- In Coq, coercion classes and implicit coercions are allowed to have parameters.
- Coherence checking involves unification of type parameters, that in the higher order case is **undecidable**.
- For coercions that respect the **uniform inheritance** condition, coherence checking is **decidable**.

Definition (uniform inheritance condition [Saïbi 1997])

For classes C and D with n and m parameters respectively, a coercion $f : C \rightarrowtail D$ is a uniform inheritance iff

$$f : \forall (x_1 : A_1) \ldots (x_n : A_n)\,(y : C\,x_1 \ldots x_n),\, D\,u_1 \ldots u_m.$$

Coherence checking

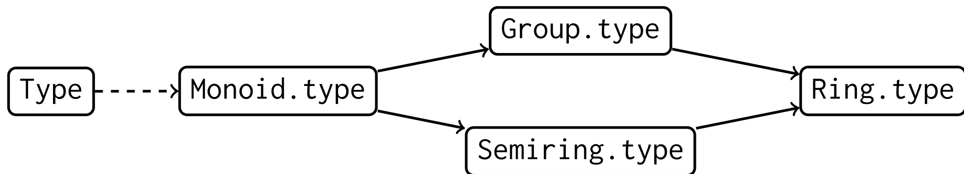
- In Coq, coercion classes and implicit coercions are allowed to have parameters.
- Coherence checking involves unification of type parameters, that in the higher order case is **undecidable**.
- For coercions that respect the **uniform inheritance** condition, coherence checking is **decidable**.

Definition (uniform inheritance condition [Saïbi 1997])

For classes C and D with n and m parameters respectively, a coercion $f : C \rightarrowtail D$ is a uniform inheritance iff

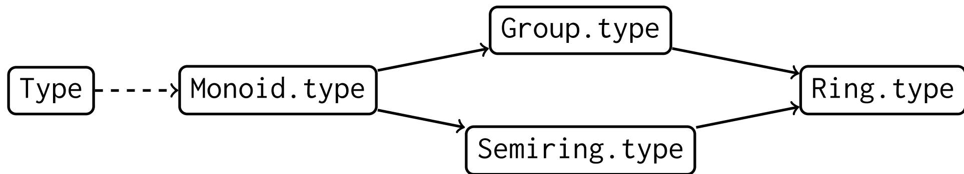
$$f : \forall (x_1 : A_1) \ldots (x_n : A_n)\,(y : C\,x_1 \ldots x_n),\, D\,u_1 \ldots u_m.$$

Automated structure inference [Mahboubi et al. 2013]



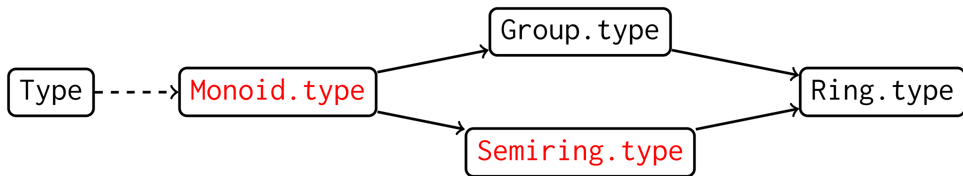
⊢ `@add _ (@zero _) (@one _) : ...`

Automated structure inference [Mahboubi et al. 2013]



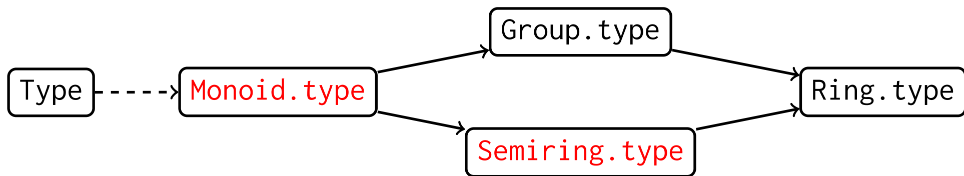
$$\dfrac{\dfrac{\vdots}{\texttt{@add}\ ?_M\ (\texttt{@zero}\ ?_M) : \texttt{Monoid.sort}\ ?_M \to \texttt{Monoid.sort}\ ?_M} \quad \dfrac{\vdots}{\texttt{@one}\ ?_{SR} : \texttt{Semiring.sort}\ ?_{SR}}}{\vdash \texttt{@add}\ ?_M\ (\texttt{@zero}\ ?_M)\ (\texttt{@one}\ ?_{SR}) : \texttt{Monoid.sort}\ ?_M}$$

Automated structure inference [Mahboubi et al. 2013]



$$\dfrac{\dfrac{\vdots}{\texttt{@add } ?_M \ (\texttt{@zero } ?_M) : \texttt{Monoid.sort } ?_M \to \texttt{Monoid.sort } ?_M} \quad \dfrac{\vdots}{\texttt{@one } ?_{SR} : \texttt{Semiring.sort } ?_{SR}} \quad \texttt{Monoid.sort } ?_M \equiv \texttt{Semiring.sort } ?_{SR}}{\vdash \texttt{@add } ?_M \ (\texttt{@zero } ?_M) \ (\texttt{@one } ?_{SR}) : \texttt{Monoid.sort } ?_M}$$

Automated structure inference [Mahboubi et al. 2013]

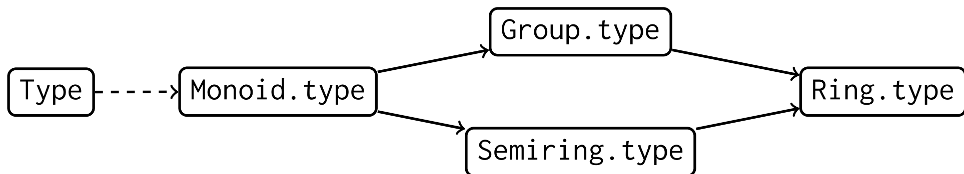


$$\dfrac{\dfrac{\vdots}{\texttt{@add } ?_M \ (\texttt{@zero } ?_M) : \texttt{Monoid.sort } ?_M \to \texttt{Monoid.sort } ?_M} \quad \dfrac{\vdots}{\texttt{@one } ?_{SR} : \texttt{Semiring.sort } ?_{SR}} \quad \texttt{Monoid.sort } ?_M \equiv \texttt{Semiring.sort } ?_{SR}}{\vdash \texttt{@add } ?_M \ (\texttt{@zero } ?_M) \ (\texttt{@one } ?_{SR}) : \texttt{Monoid.sort } ?_M}$$

The canonical solution is:

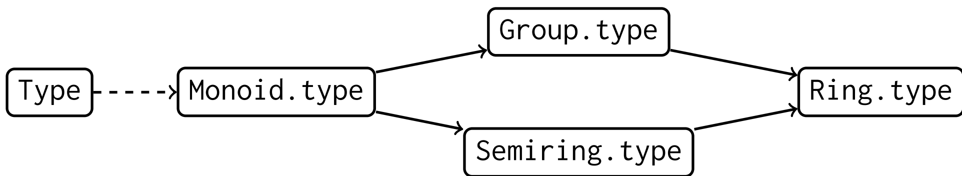
$$?_M := \texttt{Semiring.monoidType } ?_{SR}.$$

Automated structure inference [Mahboubi et al. 2013]



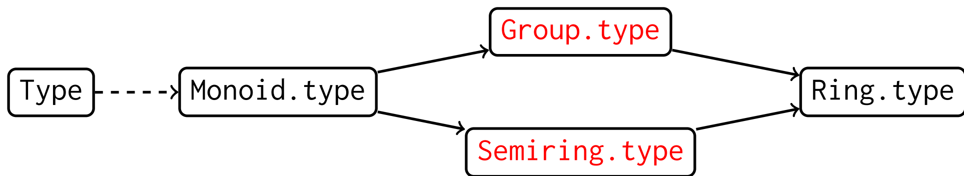
$$\frac{}{\vdash \texttt{@opp _ (@one _) : ...}}$$

Automated structure inference [Mahboubi et al. 2013]



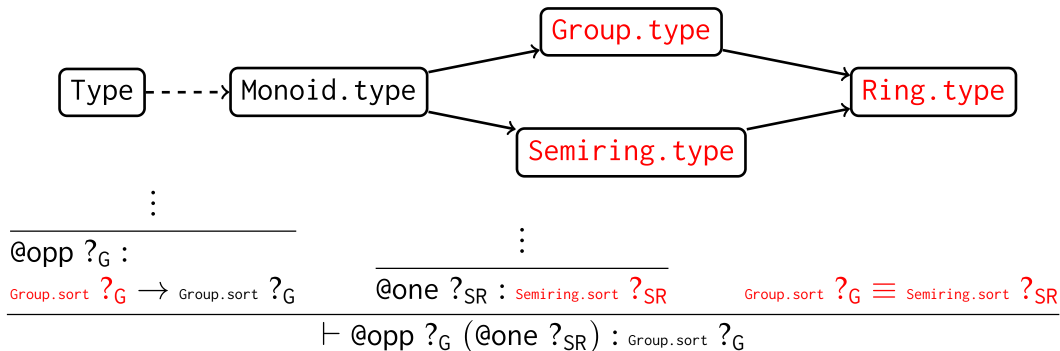
$$\dfrac{\dfrac{\vdots}{\texttt{@opp}\ ?_G : \texttt{Group.sort}\ ?_G \to \texttt{Group.sort}\ ?_G} \qquad \dfrac{\vdots}{\texttt{@one}\ ?_{SR} : \texttt{Semiring.sort}\ ?_{SR}}}{\vdash \texttt{@opp}\ ?_G\ (\texttt{@one}\ ?_{SR}) : \texttt{Group.sort}\ ?_G}$$

Automated structure inference [Mahboubi et al. 2013]



$$\dfrac{\dfrac{\vdots}{\texttt{@opp}\ ?_G : \texttt{Group.sort}\ ?_G \to \texttt{Group.sort}\ ?_G} \qquad \dfrac{\vdots}{\texttt{@one}\ ?_{SR} : \texttt{Semiring.sort}\ ?_{SR}} \qquad \texttt{Group.sort}\ ?_G \equiv \texttt{Semiring.sort}\ ?_{SR}}{\vdash \texttt{@opp}\ ?_G\ (\texttt{@one}\ ?_{SR}) : \texttt{Group.sort}\ ?_G}$$

Automated structure inference [Mahboubi et al. 2013]



$$\dfrac{\dfrac{\vdots}{\texttt{@opp}\ ?_G : \texttt{Group.sort}\ ?_G \to \texttt{Group.sort}\ ?_G} \quad \dfrac{\vdots}{\texttt{@one}\ ?_{SR} : \texttt{Semiring.sort}\ ?_{SR}} \quad \texttt{Group.sort}\ ?_G \equiv \texttt{Semiring.sort}\ ?_{SR}}{\vdash \texttt{@opp}\ ?_G\ (\texttt{@one}\ ?_{SR}) : \texttt{Group.sort}\ ?_G}$$

The canonical solution is:

$$\begin{aligned} \text{fresh } ?_R &: \texttt{Ring.type}, \\ ?_G &:= \texttt{Ring.groupType}\ ?_R, \\ ?_{SR} &:= \texttt{Ring.semiringType}\ ?_R. \end{aligned}$$

Automated structure inference [Mahboubi et al. 2013]

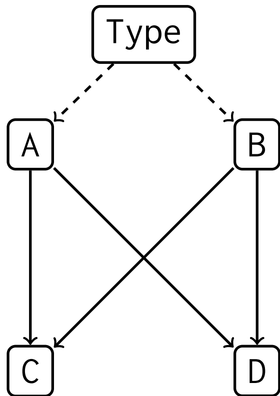
To solve a unification problem `A.sort _` $\triangleq$ `B.sort _`, we need to find a **join** structure (a minimal common subclass) `C` of `A` and `B`.

- `C` is `A` (resp. `B`) if `A` (resp. `B`) inherits from `B` (resp. `A`).
- `C` is undefined if `A` and `B` have no common subclass.

For any two structures, their **join must be unique**. (well-formedness)

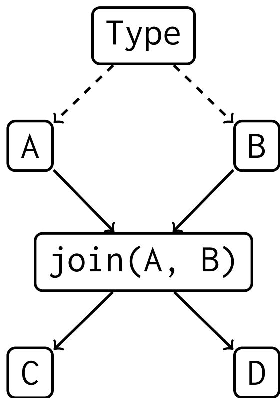
- Since we can enumerate the minimal common subclasses of any two structures, this invariant can be automatically checked.
- It is also possible to generate an exhaustive set of assertions while well-formedness checking, which state that "`C` is the join of `A` and `B`".

Why joins must be unique?



- The structures A and B have two minimal common subclasses C and D which inherit from both A and B and have extra axioms independent from each other.
- If C is inferred as the join of A and B, it can never be instantiated with D, and vice versa.

Why joins must be unique?



- The structures `A` and `B` have two minimal common subclasses `C` and `D` which inherit from both `A` and `B` and have extra axioms independent from each other.
- If `C` is inferred as the join of `A` and `B`, it can never be instantiated with `D`, and vice versa.
- We must disambiguate it by declaring an intermediate structure that just inherits from both `A` and `B` without extra axioms.

A metatheorem

- We modeled hierarchies as finite sets of structures partially ordered by the inheritance relation,
- defined the join as a binary function on a hierarchy extended with the largest structure $\top$, and
- proved the following theorem in Coq.

Theorem

The join operator on an extended well-formed hierarchy is associative, commutative, and idempotent; that is, an extended well-formed hierarchy is a join-semilattice.

A metatheorem

- This theorem implies that permuting, duplicating, and contracting unification problems do not change the result of inference; thus, it states the predictability of structure inference at a very abstract level.

Theorem

The join operator on an extended well-formed hierarchy is associative, commutative, and idempotent; that is, an extended well-formed hierarchy is a join-semilattice.

Assertion generation and checking

We generate assertions for structure inference while well-formedness checking. In Coq, those assertions can be checked by executing them as tactics.

```
check_join Group.type    Monoid.type   Group.type.
check_join Group.type    Ring.type     Ring.type.
check_join Group.type    Semiring.type Ring.type.
check_join Monoid.type   Group.type    Group.type.
check_join Monoid.type   Ring.type     Ring.type.
check_join Monoid.type   Semiring.type Semiring.type.
check_join Ring.type     Group.type    Ring.type.
check_join Ring.type     Monoid.type   Ring.type.
check_join Ring.type     Semiring.type Ring.type.
check_join Semiring.type Group.type    Ring.type.
check_join Semiring.type Monoid.type   Semiring.type.
check_join Semiring.type Ring.type     Ring.type.
```

Assertion generation and checking

We generate assertions for structure inference while well-formedness checking. In Coq, those assertions can be checked by executing them as tactics.

```
check_join Group.type    Monoid.type   Group.type.
check_join Group.type    Ring.type     Ring.type.
check_join Group.type    Semiring.type Ring.type.
check_join Monoid.type   Group.type    Group.type.
check_join Monoid.type   Ring.type     Ring.type.
check_join Monoid.type   Semiring.type Semiring.type.
check_join Ring.type     Group.type    Ring.type.
check_join Ring.type     Monoid.type   Ring.type.
check_join Ring.type     Semiring.type Ring.type.
check_join Semiring.type Group.type    Ring.type.
check_join Semiring.type Monoid.type   Semiring.type.
check_join Semiring.type Ring.type     Ring.type.
```

These lines assert that the join of monoids and semirings are semirings.

Assertion generation and checking

We generate assertions for structure inference while well-formedness checking. In Coq, those assertions can be checked by executing them as tactics.

```
check_join Group.type    Monoid.type   Group.type.
check_join Group.type    Ring.type     Ring.type.
check_join Group.type    Semiring.type Ring.type.
check_join Monoid.type   Group.type    Group.type.
check_join Monoid.type   Ring.type     Ring.type.
check_join Monoid.type   Semiring.type Semiring.type.
check_join Ring.type     Group.type    Ring.type.
check_join Ring.type     Monoid.type   Ring.type.
check_join Ring.type     Semiring.type Ring.type.
check_join Semiring.type Group.type    Ring.type.
check_join Semiring.type Monoid.type   Semiring.type.
check_join Semiring.type Ring.type     Ring.type.
```

These lines assert that the join of groups and semirings are rings.

Detecting and fixng inheritance bugs with our tools

We found and fixed many inheritance bugs in MathComp 1.7.0:

math-comp/math-comp#291: In countalg and finalg, there were 7 non-unique joins and 6 missing joins because finalg structures did not inherit from countalg structures. There were 2 missing joins in ssrnum.

math-comp/math-comp#318: The `finType` instance of `extremal_group` wrongly overwrote the join of `finType` and `countType`.

Improvements of the development process of MathComp

Our tools are also helpful to extend an existing hierarchy without paying too much attention to inheritance bugs.

math-comp/math-comp#270: Integration of the order library which introduced 16 new structures.
(We will present it in the Coq Workshop.)

MathComp Analysis [Affeldt et al. 2020] : A hierarchy of structures for functional analysis including topology, normed module, a unified notion of norms and absolute values, etc.

Now, the well-formedness checker runs as part of the CI of MathComp to reduce the reviewing burden. It allows us to focus better on the mathematical contents of changes.

Related work in IJCAR-FSCD 2020

- [Affeldt et al. 2020] discuss the convertibility issues of structure instances (and their operators) built by several inheritance constructions in detail. (IJCAR day 1)
- [Cohen et al. 2020] provide the `hierarchy-builder` tool that synthesizes definitions of mathematical structures, their inheritance, and instances based on packed classes, from high level commands. (FSCD day 2)

Conclusion

- We present checking algorithms and tools for two hierarchy invariants that ensure modularity of reasoning and predictability of inference with packed classes.
- An extended well-formed hierarchy forms a join-semilattice. This metatheorem states the predictability of structure inference at a very abstract level.
- Our checking tools are helpful to find and fix inheritance bugs, and also improve the process of further development/extension.

References I

Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. "Competing inheritance paths in dependent type theory: a case study in functional analysis". In: *IJCAR '20*. Vol. 12167. LNCS. Springer, 2020, pp. 3–20. DOI: 10.1007/978-3-030-51054-1_1.

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. "Hints in Unification". In: *TPHOLs '09*. Vol. 5674. LNCS. Springer, 2009, pp. 84–98. DOI: 10.1007/978-3-642-03359-9_8.

Gilles Barthe. "Implicit coercions in type systems". In: *TYPES '95*. Vol. 1158. LNCS. Springer, 1996, pp. 1–15. DOI: 10.1007/3-540-61780-9_58.

Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. "Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi (System Description)". In: *FSCD '20*. Vol. 167. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 34:1–34:21. DOI: 10.4230/LIPIcs.FSCD.2020.34.

François Garillot. "Generic Proof Tools and Finite Group Theory. (Outils génériques de preuve et théorie des groupes finis)". PhD thesis. École Polytechnique, Palaiseau, France, 2011. URL: https://tel.archives-ouvertes.fr/pastel-00649586.

François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *TPHOLs '09*. Vol. 5674. LNCS. Springer, 2009, pp. 327–342. DOI: 10.1007/978-3-642-03359-9_23.

References II

Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *ITP ’13*. Vol. 7998. LNCS. Springer, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

Assia Mahboubi and Enrico Tassi. “Canonical Structures for the Working Coq User”. In: *ITP ’13*. Vol. 7998. LNCS. Springer, 2013, pp. 19–34. DOI: 10.1007/978-3-642-39634-2_5.

Amokrane Saïbi. “Typing Algorithm in Type Theory with Inheritance”. In: *POPL ’97*. ACM, 1997, pp. 292–301. DOI: 10.1145/263699.263742.

The mathlib Community. “The Lean Mathematical Library”. In: *CPP ’20*. ACM, 2020, pp. 367–381. DOI: 10.1145/3372885.3373824.

The hierarchy of MathComp 1.7.0

eqType

choiceType

countType

finType

finGroupType zmodType

lmodType finZmodType ringType countZmodType

finLmodType lalgType finRingType unitRingType comRingType countRingType

finLalgType algType finUnitRingType finComRingType comUnitRingType countUnitRingType countComRingType

vectType finAlgType unitAlgType finComUnitRingType idomainType countComUnitRingType

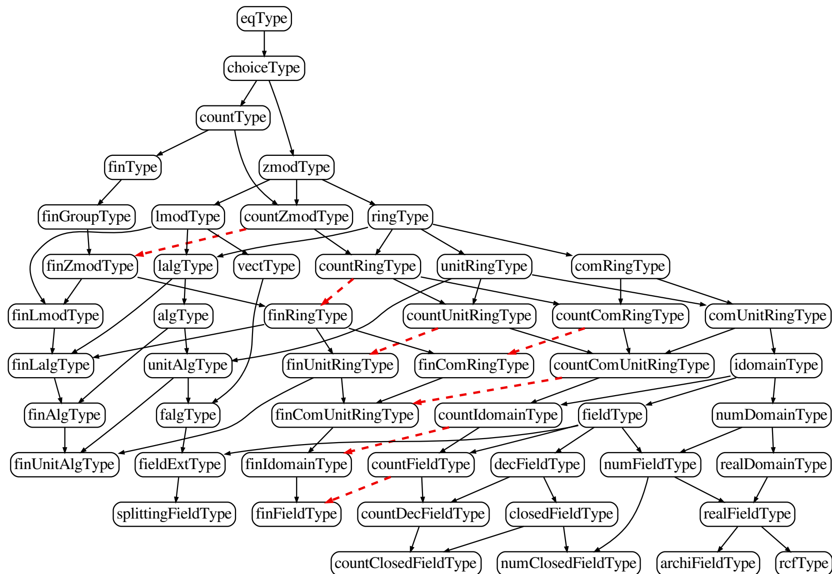
falgType finUnitAlgType finIdomainType fieldType countIdomainType numDomainType

fieldExtType finFieldType countFieldType decFieldType numFieldType realDomainType

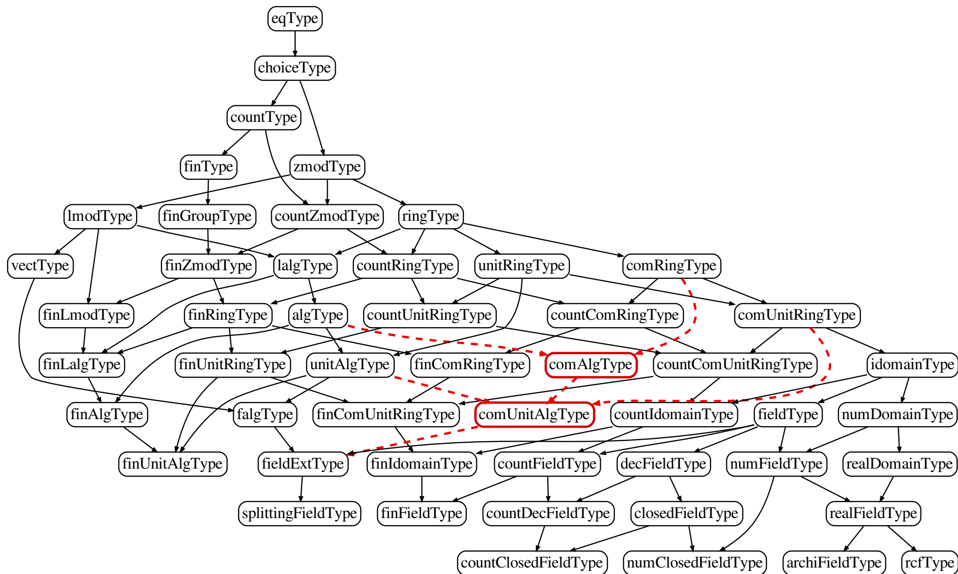
splittingFieldType countDecFieldType closedFieldType realFieldType

countClosedFieldType numClosedFieldType archiFieldType rcfType

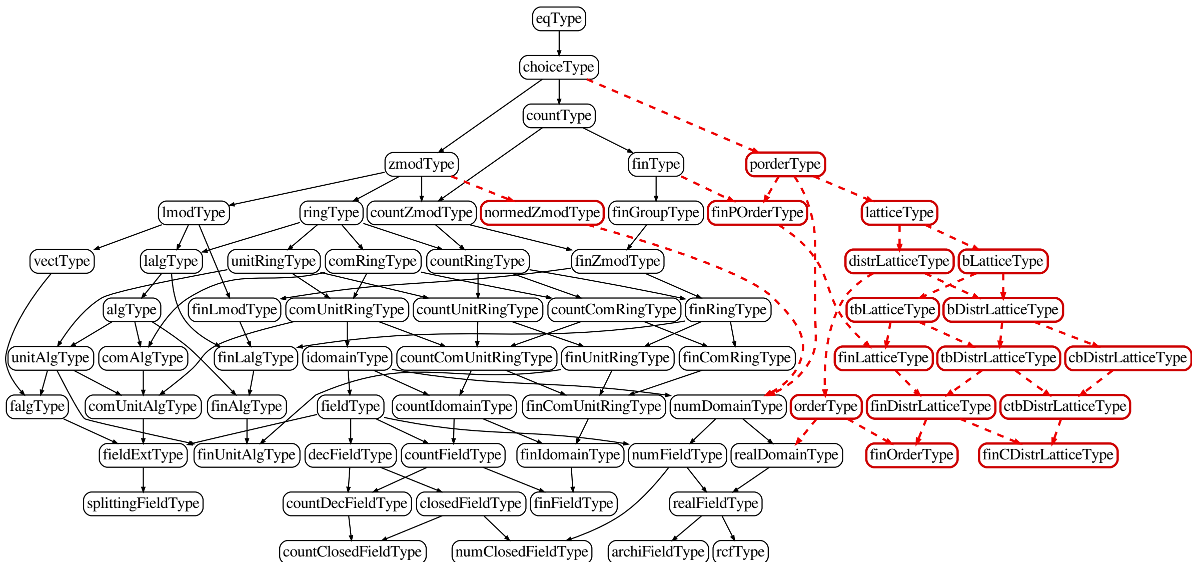
The hierarchy of MathComp 1.8.0 and 1.9.0



The hierarchy of MathComp 1.10.0



The hierarchy of MathComp 1.11.0



How to define structures? - Groups

```
Module Group.
 Record mixin_of (A : Monoid.type) := Mixin { opp : A → A; .. }.

 Record class_of (A : Type) := Class {
   base : Monoid.class_of A;
   mixin : mixin_of (Monoid.Pack A base) }.

 Structure type := Pack { sort : Type; class : class_of sort }.
End Group.

Coercion Group.sort : Group.type >-> Sortclass.

opp : ∀(A : Group.type), A → A.
```

How to define structures? - Rings

```
Module Ring.
 (* No mixin here. *)

 Record class_of (A : Type) := Class {
   base : Group.class_of A;
   mixin : Semiring.mixin_of (Monoid.Pack A (Group.base A base)) }.

 Structure type := Pack { sort : Type; class : class_of sort }.
End Ring.

Coercion Ring.sort : Ring.type >-> Sortclass.
```

Canonical structures

The `Canonical` command takes a definition with a body of the form

$$\lambda x_1 \ldots x_n, \{|p_1 := (f_1 \ldots); \ldots; p_m := (f_m \ldots)|\}$$

and then synthesizes unification hints between the projections $p_1, \ldots, p_m$ and the head symbols $f_1, \ldots, f_m$, respectively, except for unnamed projections.

Automated structure inference

In the `Ring` module:

```
Local Definition monoidType (cT : type) : Monoid.type :=
  Monoid.Pack (sort cT) (Group.base _ (base _ (class cT))).

Local Definition groupType (cT : type) : Group.type :=
  Group.Pack (sort cT) (base _ (class cT)).

Local Definition semiringType (cT : type) : Semiring.type :=
  Semiring.Pack (sort cT)
    (Semiring.Class _ (Group.base _ (base _ (class cT)))
                   (mixin _ (class cT))).

Local Definition semiring_groupType (cT : type) : Group.type :=
  Group.Pack (Semiring.sort (semiringType cT)) (base _ (class cT)).
```

A simplified formal model of hierarchies I

Definition (Hierarchy and inheritance relations)

A hierarchy $\mathcal{H}$ is a finite set of structures partially ordered by a non-strict (reflexive) inheritance relation $\rightsquigarrow^*$. We denote the corresponding strict (irreflexive) inheritance relation by $\rightsquigarrow^+$. $A \rightsquigarrow^* B$ and $A \rightsquigarrow^+ B$ respectively mean that B non-strictly and strictly inherits from A.

Definition (Common subclasses)

The (non-strict) common subclasses of $A, B \in \mathcal{H}$ are $\mathcal{C} := \{C \in \mathcal{H} \mid A \rightsquigarrow^* C \wedge B \rightsquigarrow^* C\}$. The minimal common subclasses of A and B are $\text{mcs}(A, B) := \mathcal{C} \setminus \{C \in \mathcal{H} \mid \exists C' \in \mathcal{C}, C' \rightsquigarrow^+ C\}$.

Definition (Well-formed hierarchy)

A hierarchy $\mathcal{H}$ is said to be well-formed if the minimal common subclasses of any two structures are unique; that is, $|\text{mcs}(A, B)| \leq 1$ for any $A, B \in \mathcal{H}$.

A simplified formal model of hierarchies II

Definition (Extended hierarchy)

An extended hierarchy $\bar{\mathcal{H}} := \mathcal{H} \dot{\cup} \{\top\}$ is a hierarchy $\mathcal{H}$ extended with $\top$ which means a structure that strictly inherits from all the structures in $\mathcal{H}$; thus, the inheritance relation is extended as follows:

$$
\begin{aligned}
A \mathrel{\bar{\rightsquigarrow}^*} \top &\iff \text{true}, & \\
\top \mathrel{\bar{\rightsquigarrow}^*} B &\iff \text{false} & (\text{if } B \neq \top), \\
A \mathrel{\bar{\rightsquigarrow}^*} B &\iff A \rightsquigarrow^* B & (\text{if } A \neq \top \text{ and } B \neq \top).
\end{aligned}
$$

Definition (Join)

The join is a binary operator on an extended well-formed hierarchy $\bar{\mathcal{H}}$, defined as follows:

$$
\mathrm{join}(A, B) = \begin{cases} C & (\text{if } A, B \in \mathcal{H} \text{ and } \mathrm{mcs}(A, B) = \{C\}), \\ \top & (\text{otherwise}). \end{cases}
$$

A simplified formal model of hierarchies III

Lemma

For any structures A, B, and C in an extended well-formed hierarchy $\bar{\mathcal{H}}$, C non-strictly inherits from $\text{join}(A, B)$ *if and only if C non-strictly inherits from both A and B:*
$\text{join}(A, B) \bar{\rightsquigarrow}^* C \iff A \bar{\rightsquigarrow}^* C \wedge B \bar{\rightsquigarrow}^* C.$

Theorem

The join operator on an extended well-formed hierarchy is associative, commutative, and idempotent; that is, an extended well-formed hierarchy is a join-semilattice.

Generating exhaustive assertions for join

Function `join`(A, B)**:**

  $T := (\{t \mid A \rightsquigarrow^* t\}) \cap (\{t \mid B \rightsquigarrow^* t\})$;

  /* $\mathcal{C}$ is the set of all the common subclasses of A and B. */

  foreach $t \in \mathcal{C}$ **do** $\mathcal{C} \leftarrow \{t' \in \mathcal{C} \mid \neg t \rightsquigarrow^+ t'\}$;

  if $\mathcal{C} = \emptyset$ **then return** $\top$; /* There is no join of A and B. */

  else if $\mathcal{C}$ *is singleton* $\{\mathsf{C}\}$ **then return** C; /* C is the join of A and B. */

  else fail; /* The join of A and B is ambiguous. */

foreach $A \in \mathcal{H}, B \in \mathcal{H}$ **do**

  C := `join`(A, B);

  if $A \neq B \wedge \mathsf{C} \neq \top$ **then print** "`check_join` A B C.";

end