Validating Mathematical Structures

Kazuhiko Sakaguchi

University of Tsukuba

The Coq Workshop 2019 @ Portland

Packed classes

Packaging Mathematical Structures [Garillot et al. 2009]

Packed classes are generic design patterns to define and combine algebraic structures, which support:

- multiple inheritance,
- maximal sharing of notations and theories, and
- automated structure inference.

Packed classes

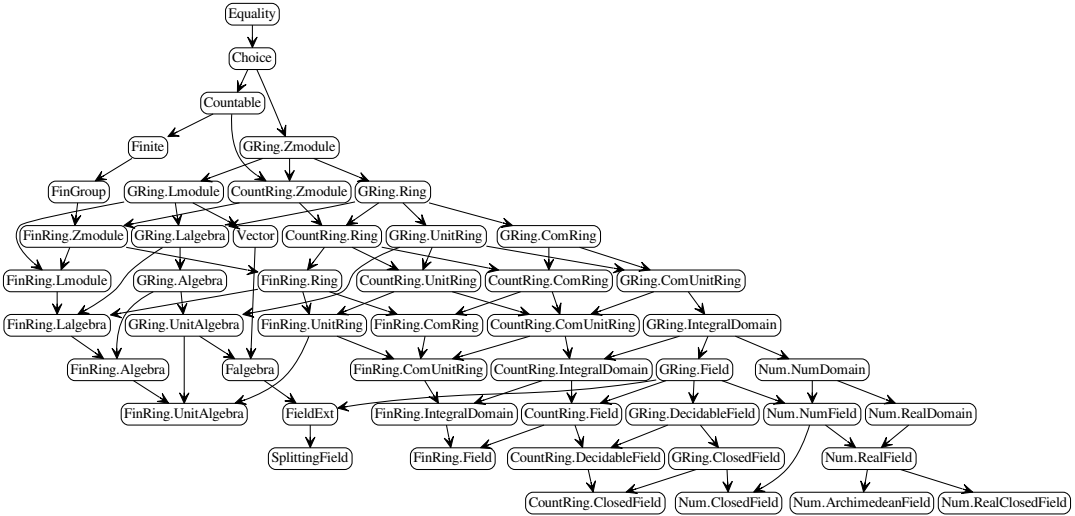
Packaging Mathematical Structures [Garillot et al. 2009]

Packed classes are generic design patterns to define and combine algebraic structures, which support:

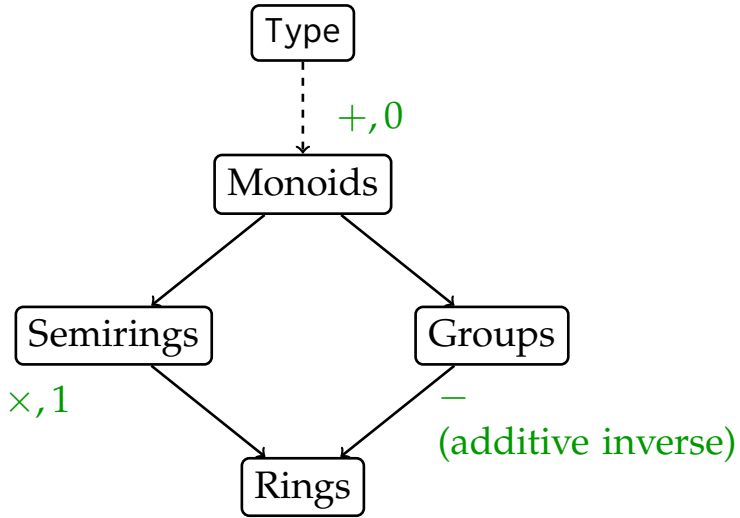
- multiple inheritance,
- maximal sharing of notations and theories, and
- automated structure inference:
  - requires a lot of unification hints in the form of canonical projections [Mahboubi et al. 2013] .
  - E.g., MathComp 1.9.0:
    49 structures, 547 implicit coercions, 711 canonical projections.

Packed classes

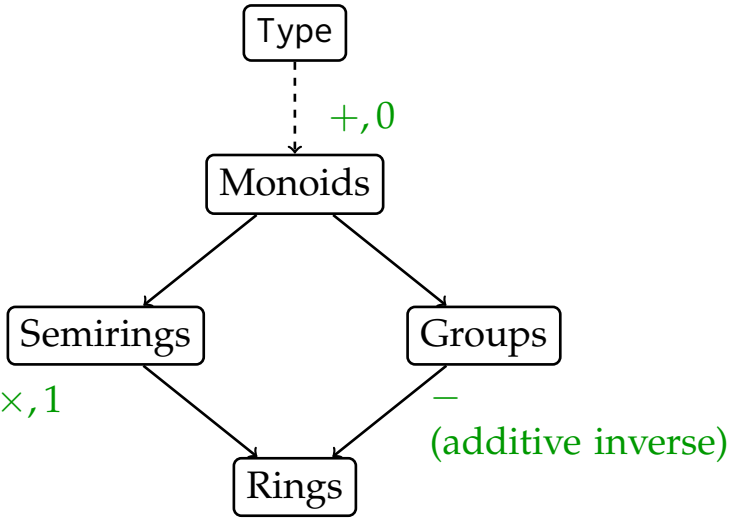
Packaging Mathematical Structures [Garillot et al. 2009]



A hierarchy with multiple inheritance



A hierarchy with multiple inheritance



less axioms,
more instances,
larger structures

more axioms,
less instances,
smaller structures

How to define a structure?

```
Module Monoid.

Record class_of (A : Type) := Class { (* set of operators and axioms *) }.

Structure type := Pack { sort : Type; _ : class_of sort }.

Local Definition class (cT : type) :=
  match cT as cT' return class_of (sort cT') with Pack _ c ⇒ c end.

End Monoid.
```

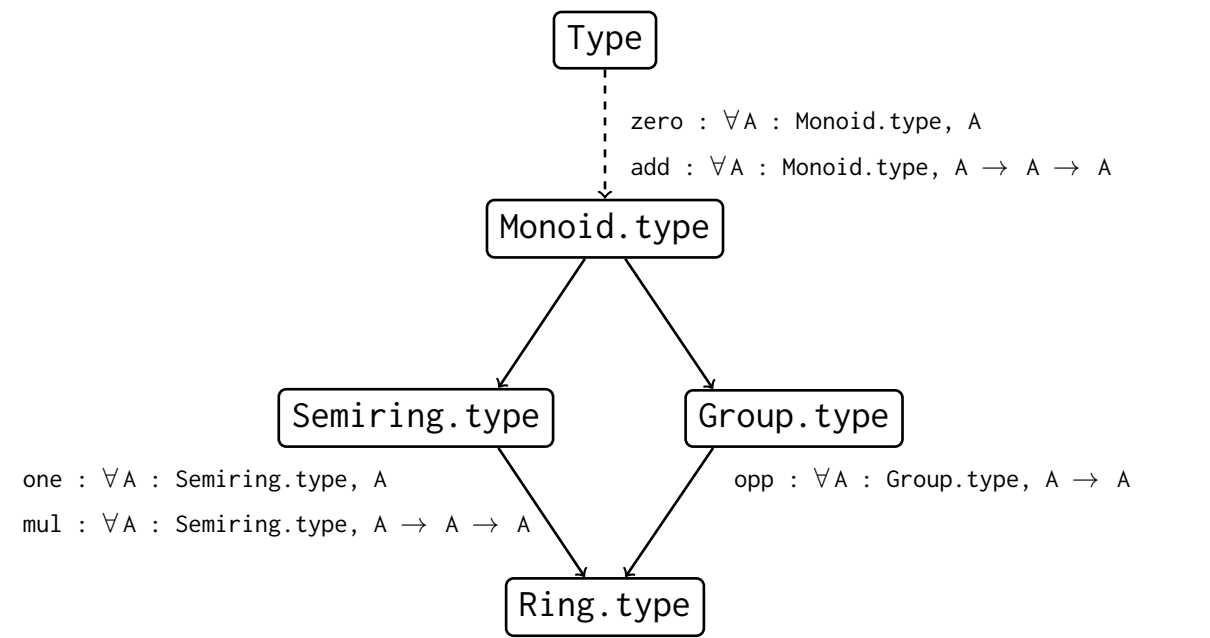
How to define a structure?

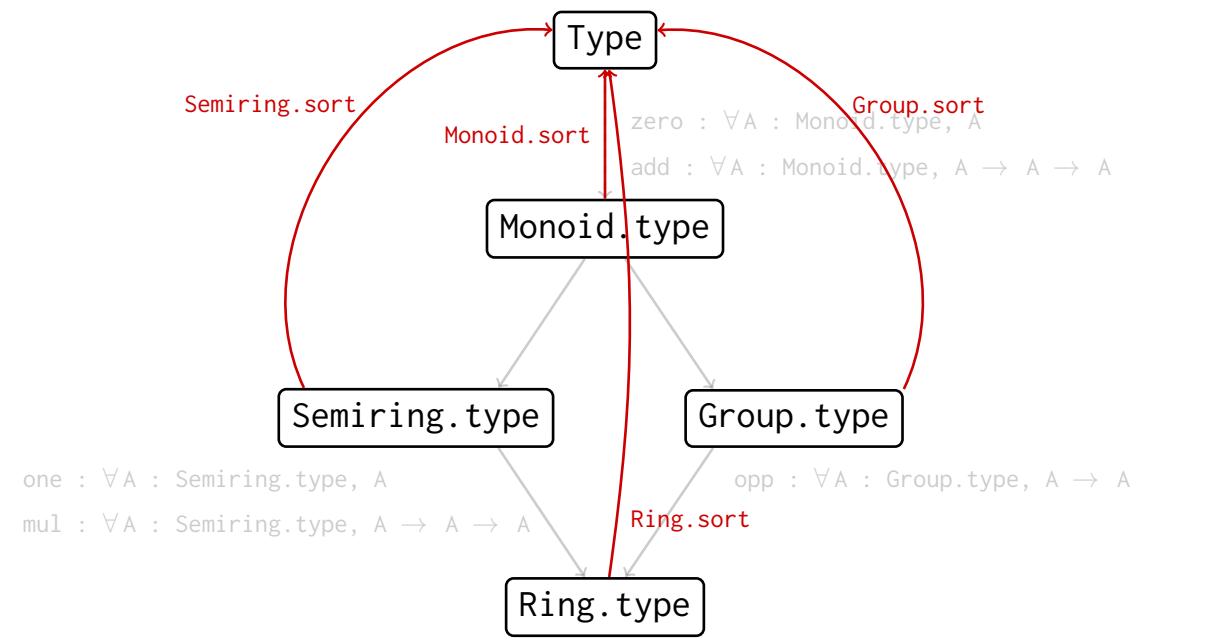
```
Module Monoid.

Record class_of (A : Type) := Class { (* set of operators and axioms *) }.

Structure type := Pack { sort : Type; _ : class_of sort }.

Local Definition class (cT : type) :=
  match cT as cT' return class_of (sort cT') with Pack _ c ⇒ c end.

End Monoid.
```

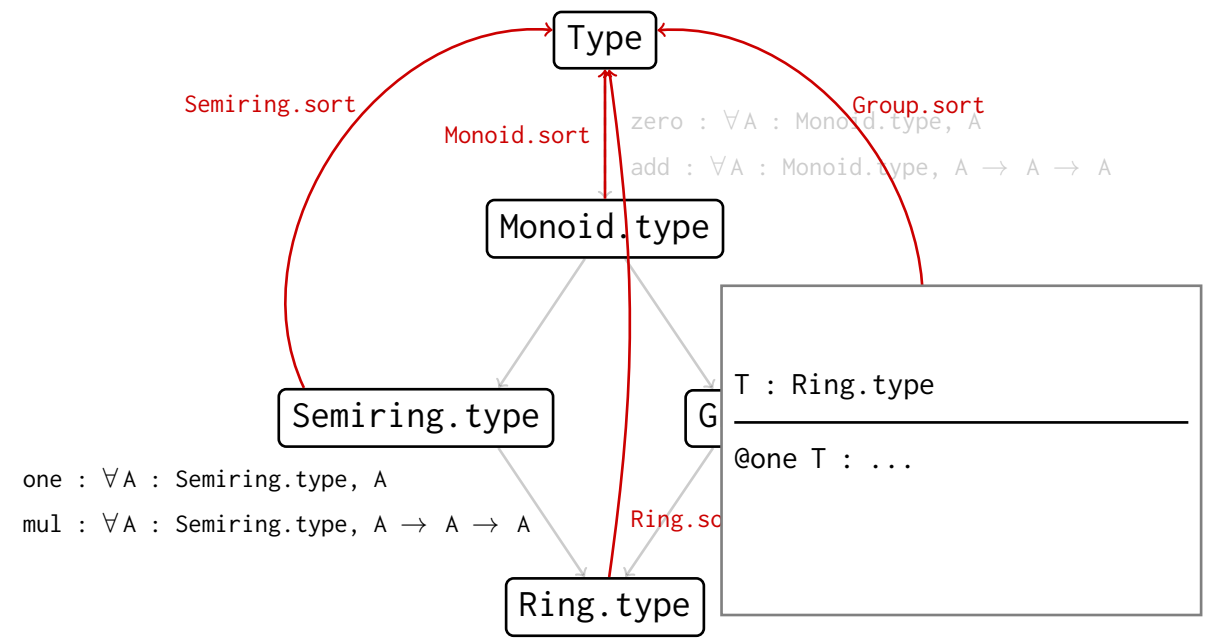
```
add  : ∀A : Monoid.type, Monoid.sort A → Monoid.sort A → Monoid.sort A
zero : ∀A : Monoid.type, Monoid.sort A
```

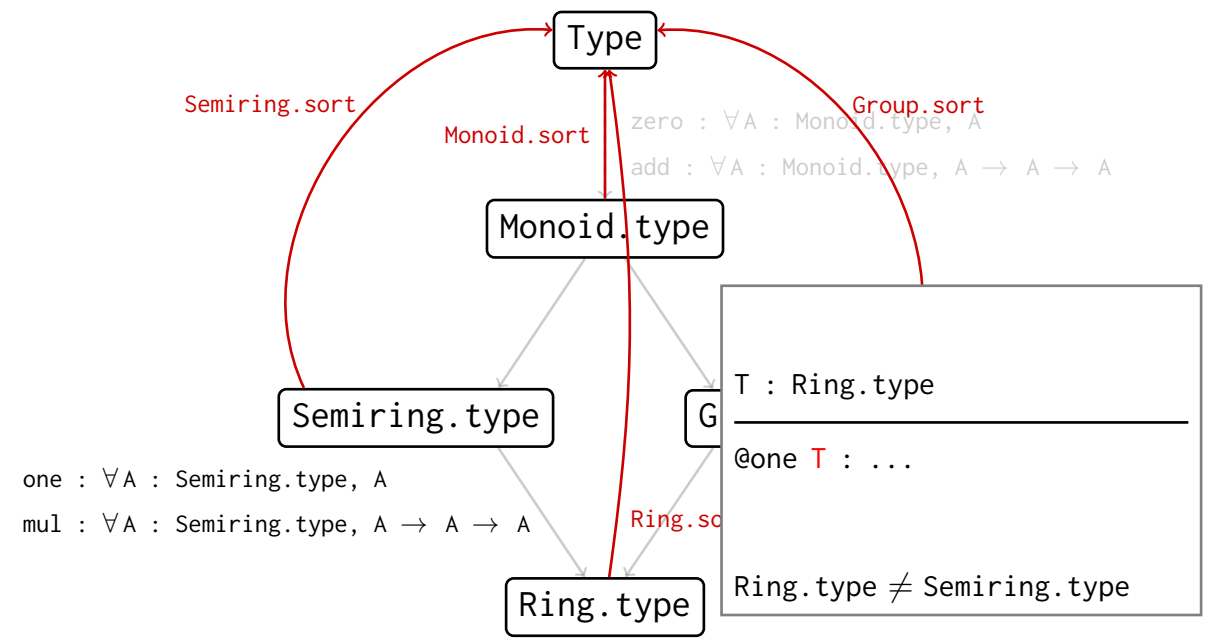
How to define a structure?

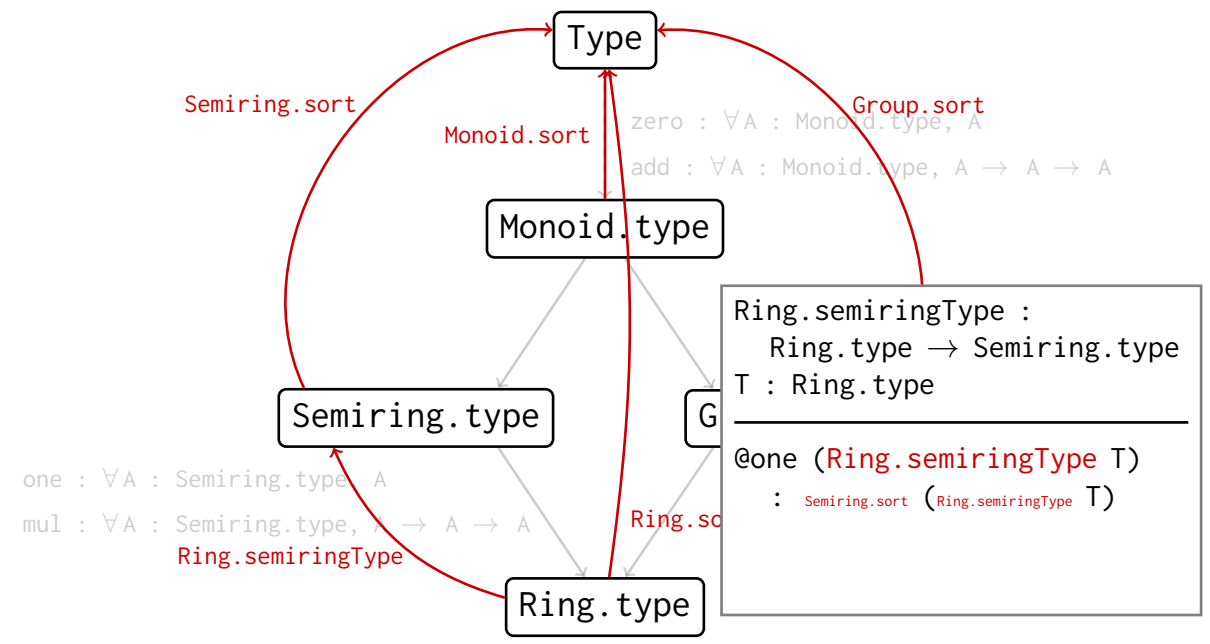
```
Module Monoid.

Record class_of (A : Type) := Class { (* set of operators and axioms *) }.

Structure type := Pack { sort : Type; _ : class_of sort }.

Local Definition class (cT : type) :=
  match cT as cT' return class_of (sort cT') with Pack _ c ⇒ c end.

End Monoid.

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


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

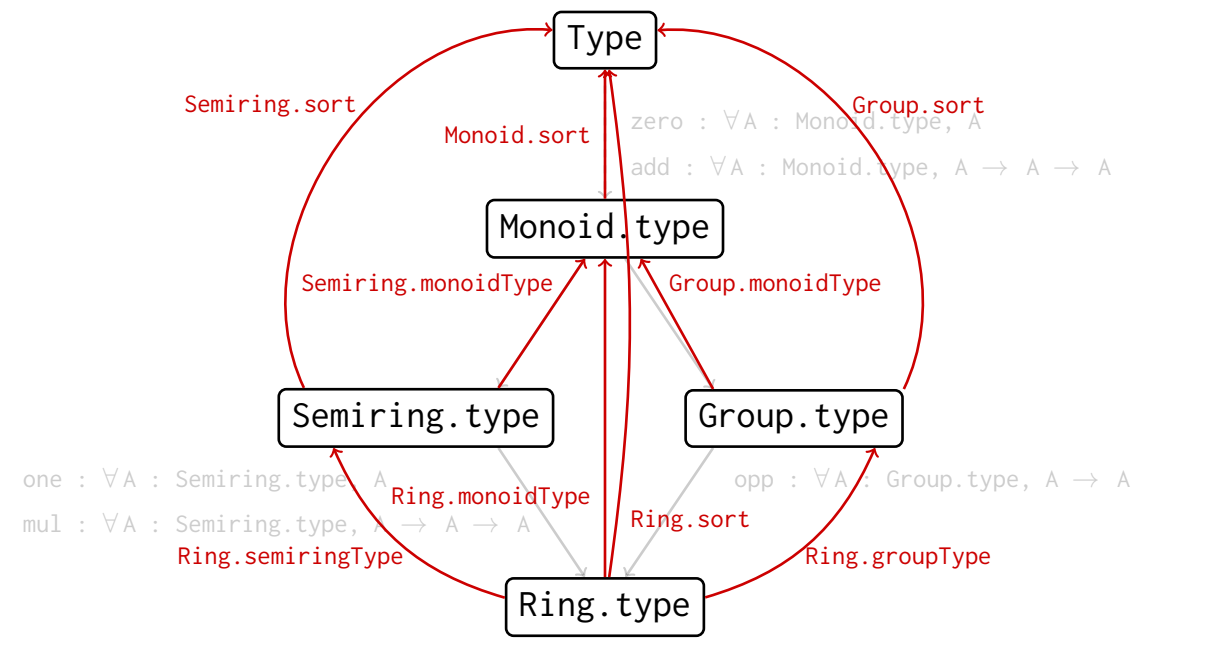

Type
zero : ∀A : Monoid.type, A
add : ∀A : Monoid.type, A → A → A
Monoid.type
Semiring.type
Group.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
opp : ∀A : Group.type, A → A
Ring.type


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
G
T : Ring.type
@one T : ...
Ring.type ≠ Semiring.type
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
Ring.sc
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
Ring.semiringType :
Ring.type → Semiring.type
T : Ring.type
@one (Ring.semiringType T)
: Semiring.sort (Ring.semiringType T)
one : ∀A : Semiring.type, A
mul : ∀A : Semiring.type, A → A → A
Ring.semiringType
Ring.sc
Ring.type


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
Ring.monoidType
Ring.sort
opp : ∀ A : Group.type, A → A
Ring.semiringType
Ring.groupType
Ring.type

Implicit coercions

- If a structure `B` (transitively) inherits a structure `A`, we should declare an implicit coercion `B >-> A`.
  - Transitive ones can be automatically computed by Coq, but we declare them explicitly to mitigate performance issues.

Implicit coercions

- If a structure `B` (transitively) inherits a structure `A`, we should declare an implicit coercion `B >-> A`.
  - Transitive ones can be automatically computed by Coq, but we declare them explicitly to mitigate performance issues.
- There are ambiguous paths for multiple inheritances, e.g.,

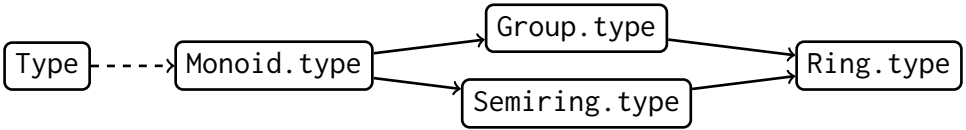
```
[Ring.monoidType] : Ring.type >-> Monoid.type.
[Ring.groupType; Group.monoidType] : Ring.type >-> Monoid.type.
[Ring.semiringType; Semiring.monoidType] : Ring.type >-> Monoid.type.
```

- If we use packed classes correctly, these ambiguous paths will be convertible with each other. If they aren't, the hierarchy is broken.

Implicit coercions

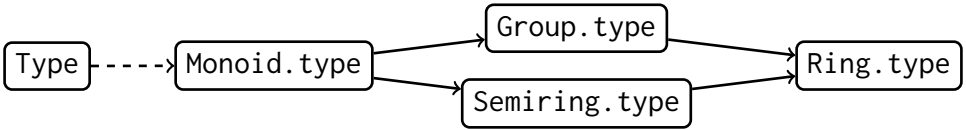
- We had no systematic way to detect inconvertible ambiguous paths in Coq 8.9, because it reports all inheritance paths as ambiguous that have the same classes as existing ones.
- We have relaxed the condition of ambiguous paths by convertibility checking. Coq 8.10 will report only inconvertible ones as ambiguous: `coq/coq#9743`.
  - This ambiguity checking is nontrivial in general, but we found that it's easy for **uniform inheritances**.

Automated structure inference



$\vdash$ @add $?_1$ (@zero $?_1$) (@one $?_2$) : ...

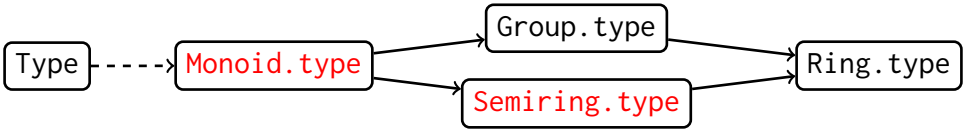
Automated structure inference



$$\frac{\vdots}{\texttt{@add}\ ?_1\ (\texttt{@zero}\ ?_1) : \texttt{Monoid.sort}\ ?_1 \to \texttt{Monoid.sort}\ ?_1} \qquad \frac{\vdots}{\texttt{@one}\ ?_2 : \texttt{Semiring.sort}\ ?_2}$$

$$\vdash \texttt{@add}\ ?_1\ (\texttt{@zero}\ ?_1)\ (\texttt{@one}\ ?_2) : \ldots$$

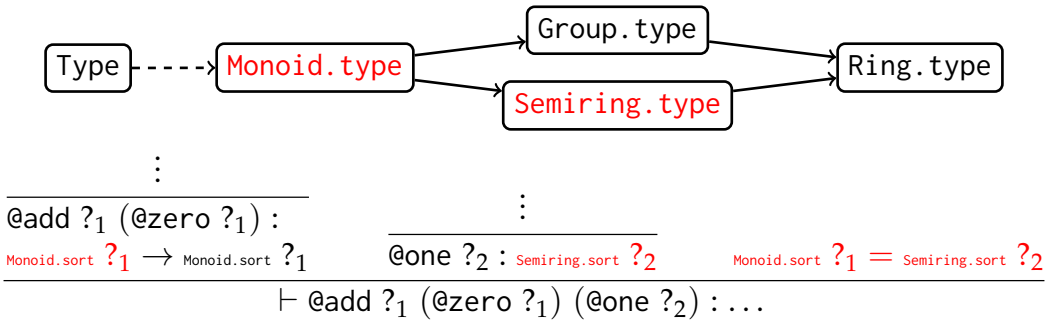
Automated structure inference



$$\frac{\vdots}{\texttt{@add}\ ?_1\ (\texttt{@zero}\ ?_1) : \texttt{Monoid.sort}\ ?_1 \to \texttt{Monoid.sort}\ ?_1} \qquad \frac{\vdots}{\texttt{@one}\ ?_2 : \texttt{Semiring.sort}\ ?_2} \qquad \texttt{Monoid.sort}\ ?_1 = \texttt{Semiring.sort}\ ?_2$$

$$\vdash \texttt{@add}\ ?_1\ (\texttt{@zero}\ ?_1)\ (\texttt{@one}\ ?_2) : \ldots$$

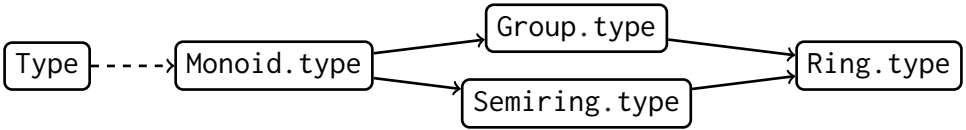
Automated structure inference



The canonical solution is:

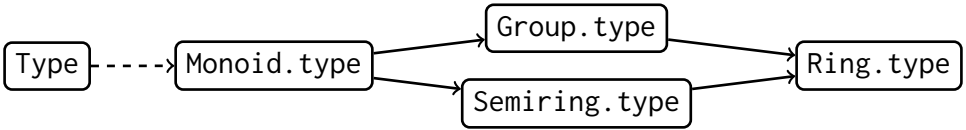
$$?_1 := \texttt{Semiring.monoidType}\ ?_2.$$

Automated structure inference



$\vdash$ @opp ?$_3$ (@one ?$_4$) : ...

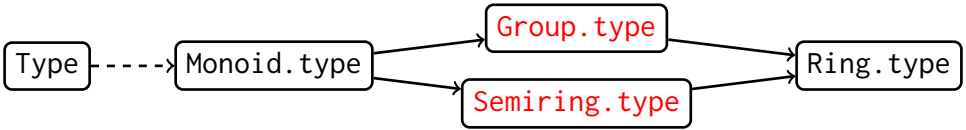
Automated structure inference



$$\frac{\vdots}{\texttt{@opp}\ ?_3 : \texttt{Group.sort}\ ?_3 \to \texttt{Group.sort}\ ?_3} \qquad \frac{\vdots}{\texttt{@one}\ ?_4 : \texttt{Semiring.sort}\ ?_4}$$

$$\vdash \texttt{@opp}\ ?_3\ (\texttt{@one}\ ?_4) : \ldots$$

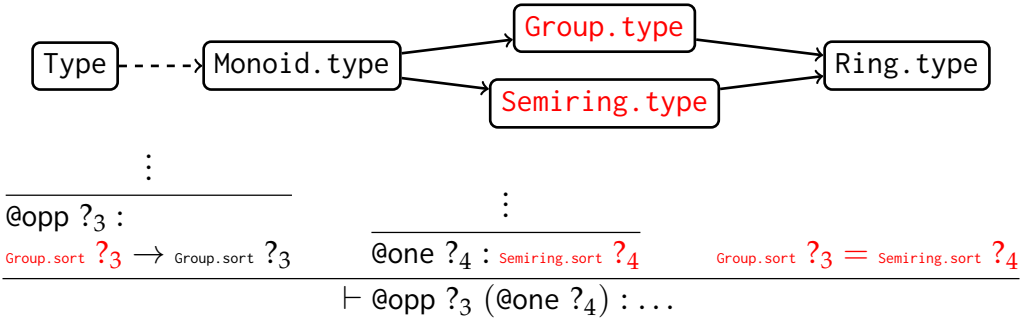
Automated structure inference



$$\frac{\vdots}{\texttt{@opp}\ ?_3 : \texttt{Group.sort}\ ?_3 \to \texttt{Group.sort}\ ?_3} \qquad \frac{\vdots}{\texttt{@one}\ ?_4 : \texttt{Semiring.sort}\ ?_4} \qquad \texttt{Group.sort}\ ?_3 = \texttt{Semiring.sort}\ ?_4$$

$$\vdash \texttt{@opp}\ ?_3\ (\texttt{@one}\ ?_4) : \ldots$$

Automated structure inference



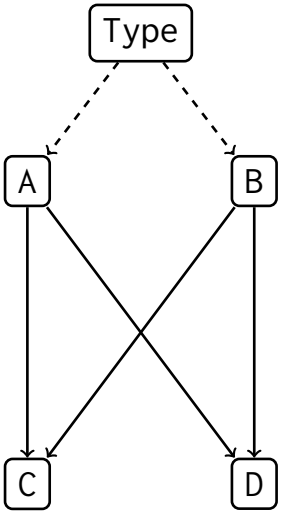
The canonical solution is:

$$\begin{aligned}&\text{fresh } ?_5 : \texttt{Ring.type},\\ &\quad ?_3 := \texttt{Ring.groupType } ?_5,\\ &\quad ?_4 := \texttt{Ring.semiringType } ?_5.\end{aligned}$$

Automated structure inference

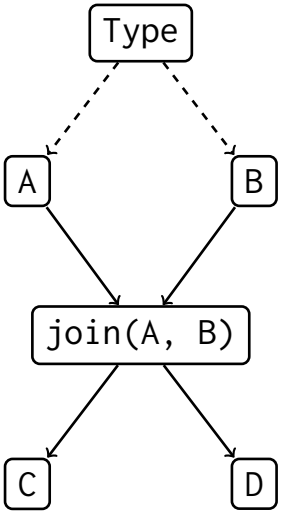
- To solve a unification problem `A.sort _` = `B.sort _`, we need to find the **join** structure `C` of `A` and `B`.
  - `C` must be a maximal common subclass of `A` and `B`.
  - `C` is `A` if `A` inherits `B`, and is `B` if `B` inherits `A`.
  - `C` is undefined if `A` and `B` have no common subclass.
- For any two structures, their **join must be unique**.
  - For a given inheritance diagram, we can validate this property and generate a set of assertions which state that "`C` is the join of `A` and `B`".
  - Generated assertions can be checked by Ltac script.

Why joins must be unique?



- The structures `A` and `B` have two maximal common subclasses `C` and `D` which inherit both `A` and `B` and have extra axioms.
- Which structure should be inferred from `A.sort _` = `B.sort _`?
  - If `C` is inferred as their join, it can never be `D`, and vice versa.

Why joins must be unique?



- The structures `A` and `B` have two maximal common subclasses `C` and `D` which inherit both `A` and `B` and have extra axioms.
- Which structure should be inferred from `A.sort _` = `B.sort _`?
  - If `C` is inferred as their join, it can never be `D`, and vice versa.
- We must disambiguate it by declaring an intermediate structure that just inherits both `A` and `B` without no extra axioms.

Assertions

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

Assertions

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

- Line 4 asserts that “the join of groups and semirings are rings.”

Assertions

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

- Line 4 asserts that “the join of groups and semirings are rings.”
- Lines 2 and 5 are symmetric ones.
  We need both of them because unification is asymmetric in Coq!

Generating exhaustive assertions for join

Function `join`(t_1, t_2)**:**

$T := (\{t \mid t_1 \to^* t\}) \cap (\{t \mid t_2 \to^* t\})$;

/* T is the set of all the common subclasses of t_1 and t_2. */

foreach $t \in T$ **do** $T \leftarrow \{t' \in T \mid \neg t \to^+ t'\}$;

if $T = \emptyset$ **then return** None; /* There is no join of t_1 and t_2. */

else if T *is singleton* $\{t\}$ **then return** Some t; /* t is the join of t_1 and t_2. */

else fail; /* A join must not be ambiguous. */

foreach $t_1 \in H, t_2 \in H$ **do**

if `join`(t_1, t_2) *is* (Some t_3) **then print** (`check_join` t_1 t_2 t_3);

end

$\to^*$ and $\to^+$ are the reflexive transitive and transitive closures of the inheritance relation respectively.

Hunting inheritance bugs with our tool

- We found and fixed many inheritance bugs in MathComp.
  - finalg structures didn't inherit countalg structures. It made many joins ambiguous.
  - The `finType` instance of `extremal_group` wrongly overwrites the join of `finType` and `countType`.

Future work

- Automating structure/inheritance declarations by meta-programming in coq-elpi
(with Enrico Tassi and Cyril Cohen).

References I


François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *TPHOLs '09*. Vol. 5674. LNCS. Springer, 2009, pp. 327–342.


Assia Mahboubi and Enrico Tassi. "Canonical Structures for the Working Coq User". In: *ITP '13*. Vol. 7998. LNCS. Springer, 2013, pp. 19–34.

Uniform inheritance condition

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

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

Uniform inheritance condition

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

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