Validating Mathematical Structures

Kazuhiko Sakaguchi*

Abstract

Packed classes [1] method is a general design pattern to define and combine algebraic structures in the Coq system. One can enable subtypings of classes and automated structure inference by combining some coercions and canonical projections [2] with packed classes. The MathComp library [4] uses these methods ubiquitously to define 49 algebraic structures, and declares more than 400 coercions and more than 800 canonical projections to implement their inheritances (Fig. 1). Declaring such a huge number of coercions and canonical projections correctly could be a very difficult task. We have implemented a validation mechanism for packed classes and their inheritances as a part of the Coq system and an external tool.

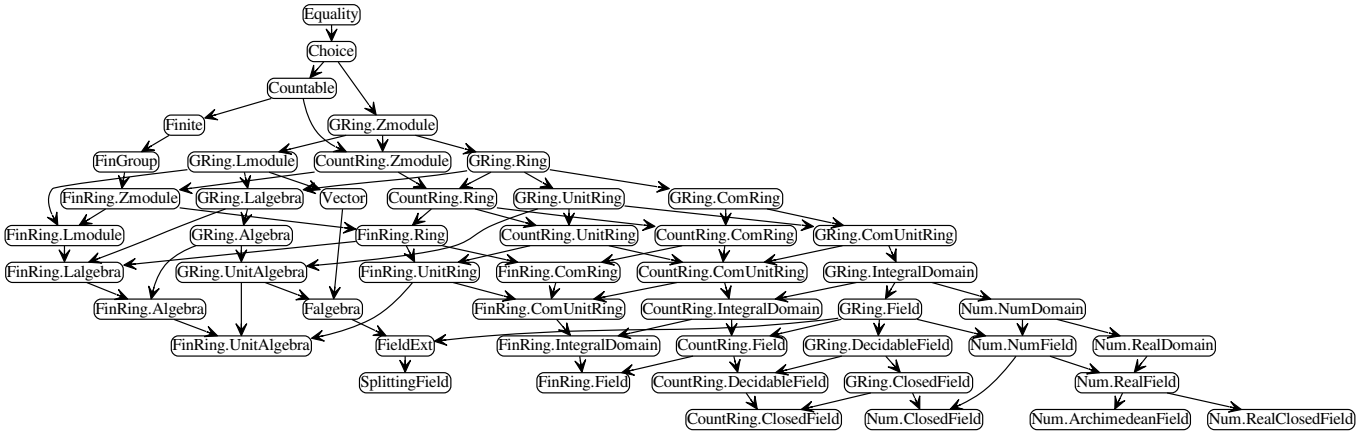

Figure 1: The hierarchy of mathematical structures in the MathComp library 1.9.0

Declaring coercions and canonical projections to implement inheritances Let us show how coercions and canonical projections work with the hierarchy consisting of rings, commutative rings, rings with units, and commutative rings with units (Fig. 2). Implementations of these structures and their description can be found in the `ssralg` library and [1, Sec. 3.2]. Therefore here we focus on their inheritances.

In Fig. 2, each edge `A.type` ⟶ `B.type` means that the structure `B.type` (directly) inherits the structure `A.type`. For any (direct or transitive) inheritance `A.type` $\overset{+}{\longrightarrow}$ `B.type`, `B.type` should be coercible to `A.type` as follows.

```
[Ring.sort] : Ring.type >-> Sortclass
[ComRing.sort] : ComRing.type >-> Sortclass
[UnitRing.sort] : UnitRing.type >-> Sortclass
[ComUnitRing.sort] : ComUnitRing.type -> Sortclass

[ComRing.ringType] : ComRing.type >-> Ring.type
[UnitRing.ringType] : UnitRing.type >-> Ring.type
[ComUnitRing.ringType] : ComUnitRing.type >-> Ring.type
[ComUnitRing.comRingType] : ComUnitRing.type >-> ComRing.type
[ComUnitRing.unitRingType] : ComUnitRing.type >-> UnitRing.type
```

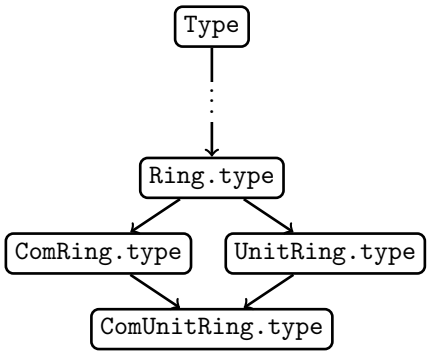

Figure 2: The hierarchy diagram of the ring structures

For any inheritance `A.type` $\overset{+}{\longrightarrow}$ `B.type`, there should be a canonical projection `B.type <- A.type` to solve a unification problem `A.sort` $?_1 \sim$ `B.sort` $?_2$ (and symmetric one) by instantiating $?_1$ with `B.aType` $?_2$, where `B.aType` is the coercion from `B.type` to `A.type`. Therefore, the last 5 coercions should also be canonical projections.

```
ComRing.sort <- Ring.sort ( ComRing.ringType )
UnitRing.sort <- Ring.sort ( UnitRing.ringType )
ComUnitRing.sort <- Ring.sort ( ComUnitRing.ringType )
ComUnitRing.sort <- ComRing.sort ( ComUnitRing.comRingType )
ComUnitRing.sort <- UnitRing.sort ( ComUnitRing.unitRingType )
```

There is one more missing canonical projection to solve a unification problem `ComRing.sort` $?_1 \sim$ `UnitRing.sort` $?_2$. Both `ComRing.type` and `UnitRing.type` do not inherit each other, but their greatest common subclass (hereinafter,

*University of Tsukuba, Japan, sakaguchi@coins.tsukuba.ac.jp

referred to as "join") `ComUnitRing.type` should be inferred from this unification problem. A solution of this unification problem is instantiating $?_1$ and $?_2$ with `ComUnitRing.comRingType` $?_3$ and `ComUnitRing.com_unitRingType` $?_3$ respectively, where $?_3$ is a fresh unification variable of type `ComUnitRing.type` and `ComUnitRing.com_unitRingType` is the canonical projection here we need.

```
ComUnitRing.com_unitRingType : comUnitRingType -> unitRingType
ComRing.sort <- UnitRing.sort ( ComUnitRing.com_unitRingType )
```

Join canonical projections could be defined in the opposite direction. Generally, one of them should be defined, and the direction does not matter.

Validating coercions by a relaxed ambiguous path condition In the above example of ring structures, the following inheritance paths have the same source and target classes `ComUnitRing.type >-> Ring.type`.

```
[ComUnitRing.ringType]
[ComUnitRing.comRingType; ComRing.ringType]
[ComUnitRing.unitRingType; UnitRing.ringType]
```

The function compositions of the above paths are convertible (judgementally equal) to each other thanks to the packed classes method. Inheritances are considered to be broken if there is an inconvertible path because one would see mysterious type mismatches on structures, but Coq had no mechanism to detect inconvertible paths until version 8.9. Therefore we have refined the warning of ambiguous inheritance paths to report only inconvertible ones. Convertibility checking of two inheritance paths is non-trivial in general. However, we have discovered and implemented a simple checking procedure to do this for inheritance paths satisfying uniform inheritance condition [3].

Validating canonical projections We have implemented an external tool `hierarchy.ml` in OCaml to validate two kinds of canonical projections on mathematical structures: inheritances and joins. Our tool takes the list of canonical projections by using the `Print Canonical Projections` Vernacular command, filters out entries other than inheritances, checks the transitivity of inheritance relation, and generates exhaustive assertions for automatic structure inference as a `.v` file. The following assertions can be generated from the hierarchy of Fig. 2 by `hierarchy.ml`.

```
check_join Ring.type Ring.type Ring.type.
check_join Ring.type ComRing.type ComRing.type.
check_join Ring.type UnitRing.type UnitRing.type.
check_join Ring.type ComUnitRing.type ComUnitRing.type.
check_join ComRing.type Ring.type ComRing.type.
check_join ComRing.type ComRing.type ComRing.type.
check_join ComRing.type UnitRing.type ComUnitRing.type.
check_join ComRing.type ComUnitRing.type ComUnitRing.type.
check_join UnitRing.type Ring.type UnitRing.type.
check_join UnitRing.type ComRing.type ComUnitRing.type.
check_join UnitRing.type UnitRing.type UnitRing.type.
check_join UnitRing.type ComUnitRing.type ComUnitRing.type.
check_join ComUnitRing.type Ring.type ComUnitRing.type.
check_join ComUnitRing.type ComRing.type ComUnitRing.type.
check_join ComUnitRing.type ComUnitRing.type ComUnitRing.type.
check_join ComUnitRing.type UnitRing.type ComUnitRing.type.
```

`check_join` is implemented as a tactic notation, and `check_join t1 t2 t3` asserts that the join of `t1` and `t2` is `t3`. These assertions are generated by finding the join for all the ordered combinations of two structures using the following `join` procedure.

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

$T := (\{t \mid t_1 \xrightarrow{+} t\} \cup \{t_1\}) \cap (\{t \mid t_2 \xrightarrow{+} t\} \cup \{t_2\})$; /* T is the set of all the common subclasses of t_1 and t_2. */
foreach $t \in T$ **do** $T \leftarrow \{t' \in T \mid t \not\xrightarrow{+} t'\}$;
if $T = \emptyset$ **then return** None; /* There is no join of t_1 and t_2. */
else if T *is a singleton set* $\{t\}$ **then return** t; /* t is the join of t_1 and t_2. */
else fail; /* There are more than two join structures of t_1 and t_2. A join should not be ambiguous. */

Evaluation We have applied our validation mechanism to the MathComp 1.7.0 library, found no bugs on coercions, and found the following bugs on canonical projections. 1. The joins of `countType` and some algebraic structures (`zmodType`, `ringType`, etc.) are ambiguous because `finalg` has no inheritances from `countalg`. 2. There are 7 missing join canonical projections excluding ambiguous ones. 3. The inferred join of `finType` and `countType` is `extremal_group` because of its incorrect canonical `finType` instance implementation.

The current implementation of validation is incomplete. At least, we cannot check the hierarchies of morphisms and sub-structures with our tool. Although, the above results sustain that our tool can detect some inheritance bugs.

Acknowledgement We appreciate the support of the Marelle project-team, INRIA Sophia Antipolis. In particular, we want to thank Cyril Cohen and Enrico Tassi for providing help to understand packed classes and canonical structures well and to implement our validation mechanisms.

References

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

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

[3] Kazuhiko Sakaguchi. *Relax the ambiguous path condition of coercion*. URL: https://github.com/coq/coq/pull/9743.

[4] The Mathematical Components project. *The Mathematical Components repository*. URL: https://github.com/math-comp/math-comp.