



HAL
open science

Design patterns of hierarchies for order structures

Xavier Allamigeon, Quentin Canu, Cyril Cohen, Kazuhiko Sakaguchi,
Pierre-Yves Strub

► **To cite this version:**

Xavier Allamigeon, Quentin Canu, Cyril Cohen, Kazuhiko Sakaguchi, Pierre-Yves Strub. Design patterns of hierarchies for order structures. 2023. hal-04008820

HAL Id: hal-04008820

<https://inria.hal.science/hal-04008820v1>

Preprint submitted on 28 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design patterns of hierarchies for order structures

Xavier Allamigeon ✉

Inria and CMAP, CNRS, École Polytechnique, IP Paris, France

Quentin Canu ✉

Université Paris-Saclay, Inria and CMAP, CNRS, École Polytechnique, IP Paris, France

Cyril Cohen ✉🏠 

Université Côte d’Azur, Inria, France

Kazuhiko Sakaguchi ✉ 

Inria, France

Pierre-Yves Strub ✉

France

Abstract

Using order structures in a proof assistant naturally raises the problem of working with multiple instances of a same structure over a common type of elements. This goes against the main design pattern of hierarchies used for instance in Coq’s `MathComp` or Lean’s `mathlib` libraries, where types are canonically associated to at most one instance and instances share a common overloaded syntax.

We present new design patterns to leverage these issues, and apply them to the formalization of order structures in the `MathComp` library. A common idea in these patterns is *underloading*, i.e., a disambiguation of operators on a common type. In addition, our design patterns include a way to deal with duality in order structures in a convenient way. We hence formalize a large hierarchy which includes partial orders, semilattices, lattices as well as many variants.

We finally pay a special attention to order substructures. We introduce a new kind of structure called *prelattice*. They are abstractions of semilattices, and allow us to deal with finite lattices and their sublattices within a common signature. As an application, we report on significant simplifications of the formalization of the face lattices of polyhedra in the `Coq-Polyhedra` library.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics; Mathematics of computing → Mathematical software

Keywords and phrases formalization of mathematics, hierarchies of mathematical structures, packed classes, order structures, lattices

1 Introduction

A common problem in the formalization of algebraic structures is how to handle multiple instances on the same “carrier” type. Although this is a general problem, it arises very early in the formalization of order structures, since it is natural for a given carrier type to have several order relations. Indeed, every partial order \leq over a type T induces a dual order \leq^d , where $x \leq^d y$ if and only if $y \leq x$. Moreover, some data types have several notions of ordering, and none of them is canonical. For instance, natural numbers can at least be ordered using the “lesser than” relation and the divisibility relation. Similarly, lists and Cartesian products of ordered types can be ordered using the lexicographic and pointwise orders, etc. Many applications require the simultaneous manipulation of different order relations over the same ground set. To mention just a few, it appears in computer algebra with Gröbner bases of (multivariate) polynomial ideals, where the bases depend on the order chosen over the degrees of monomials; this collection of monomial ordering gives rise to the notion of *Gröbner fan* [26]. In economics, generalizations of stable matching problems and Gale–Shapley algorithm also lead to find antichains (collections of incomparable elements) in the sense of several orders; see [2, 10].

Having multiple instances of a given structure over the same carrier type actually conflicts with the overloading of definitions and notations that are traditionally exploited in the formalization of algebraic structures. On the one hand, in the context of interactive theorem proving, the user needs to keep the ability to distinguish between the different order relations over a common type, despite overloading. On the other hand, theorems should be unique and generic enough to apply to several order relations, no matter how the latter are distinguished. This discrepancy between operators and theorems is usually not accounted for in the usual formalization of algebraic hierarchies in modern proof assistants.

Contributions. We introduce new design patterns that bring the possibility to manipulate multiple instances on the same carrier type, while still retaining the genericity of the theories of algebraic structures. One of the ingredients of our approach is a way to perform overloading disambiguation, which we call *underloading*. We present the implementation of this approach on the formalization of a variety of order structures in the proof assistant `Coq` [27] and its library `MathComp` [33]. The resulting hierarchy includes partial orders, meet- and join-semilattices, lattices, all with some variants like the existence of bottom or top elements, and distributivity of meet and join operators, etc.

In more details, we introduce a first layer of order structures, referred to as the *order relation hierarchy*, where inference of instances is guided by the order relation rather than a carrier type. This hierarchy relies on semi-bundled canonical structures, i.e., dependent records where the carrier type is a parameter. We also deal with the case where the simultaneous use of distinct orders over the same type is only occasional, and the traditional type-as-carrier paradigm is still relevant. We show that the previous design pattern is compatible with this situation, by defining atop of the first hierarchy a second layer of order structures where the inference is based on the carrier type. We refer to it as the *ordered type hierarchy*. The novelty here is the addition of display phantom types to carry out disambiguation of the notations by the system. We make sure generic theorems from the order relation hierarchy apply to the statements using the ordered type hierarchy as well.

We integrate duality of order relations at the core of the order hierarchies. To this aim, we design the axioms of order structures to get convenient features such as convertibility between a structure and the dual of its dual, or easy interaction between dual orders and product orders.

Dealing with order substructures is another motivation to have structures sharing the same carrier type. We make an extra step by designing a library for finite lattices that is well-suited to the manipulation of sublattices. This relies on the introduction of an original order structure, which we call *prelattice*, that serves as an ambient structure providing a common carrier type, order relation, and a uniform way to construct meet and join operators for all finite lattices within. This ensures that the meet and join operators are consistent between two comparable sublattices. We show the benefit of this way to handle finite lattices on the formalization of polyhedra and their faces.

Organization of the paper. In Section 2 we recall the main ingredients on how algebraic canonical structures hierarchies are usually designed. Section 3 deals with the new design patterns for order hierarchies, including the ordered type and the order relation hierarchies and the treatment of duality. In Section 4, we present the formalization of finite lattices atop of prelattices, and their application to the face lattice of polyhedra. We finally discuss related work in Section 5.

The source code of this work is provided with the submission as supplementary material. The modules of the source code are referred to throughout the paper. Further details can be found in the `README` file at the root of the archive.

2 Hierarchy of mathematical structures in type theory

2.1 Structures as dependent records

This section reviews some known approaches to define mathematical structures in dependent type theory. A structure is a set of objects endowed with additional features such as operations, relations, and axioms. Such a structure can be encoded as a dependent record type that bundles its components. For example,

```
Structure eqType := EqType { eq_sort : Type; eq_op : rel eq_sort }.
```

represents a type (`eq_sort`) equipped with an equality comparison function (`eq_op`), where

- `rel T` is a shorthand for $T \rightarrow T \rightarrow \text{bool}$, and stands for the type of relations over T ;
- `eqType` is the name of the record type;
- `EqType` is the only constructor of `eqType` of type $\forall (T : \text{Type}), \text{rel } T \rightarrow \text{eqType}$;
- `eq_sort` and `eq_op` are the name of the fields, which also work as record projections, i.e., they have type `eqType` \rightarrow `Type` and $\forall E : \text{eqType}, \text{rel } (\text{eq_sort } E)$ respectively, and they return the corresponding component of a given record instance.

Although we omit the axioms here, we can add them as record fields as well, e.g.,

```
eq_op_refl :  $\forall x, \text{eq\_op } x \ x = \text{true}$ ;
```

to state that `eq_op` should be reflexive.

Since a record type may have parameters, we can move some components from fields to parameters as follows.

```
Class eqClass (sort : Type) := EqClass { eq_op : rel sort }.
```

The choice of which components to have as parameters or fields is called a *bundling*. Records like `eqType` are called (*fully*) *bundled* structures [11, 12, 20]. In contrast, records like `eqClass` are referred to as *semi-bundled* structures [34, Sect. 4.1.1], because they bundle all the components except the carrier type.

2.2 Overloading and structure inference

Canonical structures [22, 19, 31] enable overloading of record projections in Coq through a higher-order unification algorithm [36] extended with hints [5]. For example, the `eqType` record in Section 2.1 can be seen as a database relating types `eq_sort` to their canonical comparison functions `eq_op`. In order to relate a type, say natural numbers `nat`, to its comparison function `eqn` of type `rel nat`, we declare an instance of this record type:

```
Canonical nat_eqType : eqType := { | eq_sort := nat; eq_op := eqn | }.
```

where `{ | ... | }` is an alternative syntax for record instance construction that emphasizes the correspondence between projections and fields, i.e., the body of `nat_eqType` is equivalent to `EqType nat eqn`.

The above declaration `nat_eqType` allows us to typecheck `eq_op 1 2` where the first argument of `eq_op` is made implicit and `1` and `2` have type `nat`. Since `eq_op` has type

$$\forall T : \text{eqType}, \text{eq_sort } T \rightarrow \text{eq_sort } T \rightarrow \text{bool},$$

solving a type equation `eq_sort ?T $\hat{=}$ nat` suffices to typecheck this term, where `?T` is a unification variable referring to the unknown first argument of `eq_op`. The above declaration instructs Coq to solve this unification problem by instantiating `?T` with `nat_eqType`.

As a rule of thumb, for each field `proj := f ...` in its body, a canonical instance declaration `I` instructs Coq to solve unification problems of the form `proj ?I $\hat{=}$ f ...` by instantiating `?I` with `I`.

We can also declare a canonical `eqType` instance for a polymorphic type, e.g., the Cartesian product `T1 * T2` of any `eqType` instances `T1` and `T2`, by parameterizing the instance.

```
Canonical prod_eqType (T1 T2 : eqType) : eqType := { |
  eq_sort := eq_sort T1 * eq_sort T2;
  eq_op x y := eq_op x.1 y.1 && eq_op x.2 y.2 | }.
```

Thanks to this canonical declaration, Coq will solve any type equation of the form `eq_sort ?T ≐ T1 * T2` by generating two unification problems `eq_sort ?T1 ≐ T1` and `eq_sort ?T2 ≐ T2`, and using its solutions to instantiate `?T` with `prod_eqType ?T1 ?T2`.

2.3 Inheritance and coherence in packed classes

Mathematical structures form an inheritance hierarchy. If any instance of a structure A forms an instance of another structure B , we say that A *inherits* from B , B is *poorer* than A , and A is *richer* than B , e.g., fields inherit from rings and rings inherit from groups.

The *packed classes* discipline [11] used in the `MathComp` library is a uniform way to implement such an inheritance hierarchy, while allowing multiple inheritance and enabling instance resolution by canonical structures.

In packed classes, the definition of each structure is split into (at least) 2 records, called a *class* and a *structure*. We redefine the `eqType` structure as a packed class below.

```
Record eqClass (sort : Type) := EqClass { eq_op : rel sort }.
Structure eqType := EqType { eq_sort : Type; eq_class : eqClass eq_sort }.
```

In a fully-bundled hierarchy where the inference is guided by the carrier type, a class record, e.g., `eqClass` above, is always a semi-bundled record that takes the carrier type as a parameter and bundles all the other components of the structure. The structure record, e.g., `eqType` above, is a fully-bundled record that bundles the carrier type with its class instance.

As an example of inheritance from `eqType`, we define (partially) ordered types.

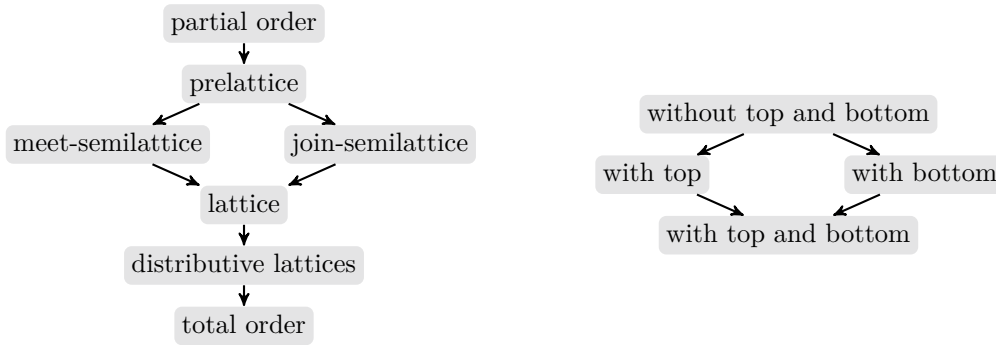
```
Record ordClass (sort : Type) :=
  OrdClass { ord_eqClass : eqClass sort; le_op : rel sort }.
Structure ordType := OrdType { ord_sort : Type; ord_class : ordClass ord_sort }.
```

Inheritance in packed classes should always be done by including all the components of the class record of a poorer structure, e.g., `eqClass`, in that of a richer structure, e.g., `ordClass`. In the above definition, the latter includes the former as the first field `ord_eqClass`. A subtyping function from `ordType` to `eqType` can be defined by “rebundling” the given `ordType` instance as an `eqType` instance using only constructors and projections of records, since the former includes all the components of the latter.

```
Canonical ord_eqType (T : ordType) : eqType :=
  { | eq_sort := ord_sort T; eq_class := ord_eqClass (ord_class T) | }.
```

The principle according to which structure inheritance and subtyping are respectively done by inclusion and erasure of extra components, is called *forgetful inheritance* [1]. It plays a crucial role in ensuring the coherence [21, 23, 1]—the property that instances of the same structure on the same carrier type obtained by several inheritance constructions are convertible—of the hierarchy.

The above subtyping function `ord_eqType` also instructs Coq to solve unification problems of the form `eq_sort ?E ≐ ord_sort ?O` by instantiating `?E` with `ord_eqType ?O`, and enables the corresponding inheritance in structure inference.



■ **Figure 1** The inheritance hierarchy of the order structures. Left: Overview of the hierarchy without variants. Right: an abbreviate hierarchy of the variants of every structure.

3 Hierarchies of order relations and ordered types

We recall that a partial order \leq over a set S is a relation that is reflexive, antisymmetric and transitive. The top and the bottom, denoted by \top and \perp , of the partially ordered set (S, \leq) are the largest and smallest elements, i.e., $\perp \leq x$ and $x \leq \top$ for any $x \in S$. The partially ordered set (S, \leq) turns into a *meet-semilattice* when it has a binary operator \wedge on S , called *meet*, such that $x \wedge y$ is the greatest lower bound of x and y , i.e., $z \leq x \wedge y$ if and only if $z \leq x$ and $z \leq y$ for any $x, y, z \in S$. *Join-semilattices* and its binary operator \vee , called *join*, are defined dually. When S is a meet- and join-semilattice, it is called a *lattice*. It is said to be *distributive* when the meet distributes over the join (or, equivalently, the join distributes over the meet).

Figure 1 illustrates the inheritance hierarchy of order structures we implemented. This hierarchy roughly consists of partial order, meet- and join-semilattices, lattices, distributive lattices, and total order as shown in the left side. Prelattices are an intermediate structure between partial order and semilattices, discussed in Section 4. Each of those order structures except prelattices has four variants: without top and bottom, with top, with bottom, and with both top and bottom, as described in right side of Figure 1. Therefore, every inheritance relation described in the left side of Figure 1 actually has to be broken down into several relations taking the top/bottom elements into account. As described in the introduction, the hierarchy actually consists of two layers, namely the order relation and ordered type hierarchies. We point out that every order structure listed in Figure 1 is implemented in the two hierarchies. These hierarchies can be respectively found in `ssreflect/reorder.v` and `ssreflect/order.v` in the `MathComp` archive provided with the submission.

This section explains the key implementation idea of the order hierarchies. We start by giving an overview of the ordered type hierarchy in Section 3.1 and the use of underloading, as the design pattern (fully-bundled structures, type aliases) is easier to grasp on first reading. We then explain in Section 3.2 the implementation of the order relation hierarchy based on another kind of underloading, and how the ordered type hierarchy builds on it. We finally deal with duality in Section 3.3 in both hierarchies.

3.1 Multiple instances in the fully-bundled ordered type hierarchy

The canonical structure mechanism has a limitation that at most one instance can be associated to a pair of a projection and a head constant that appears in unification problems, e.g., `ord_sort ?o ≐ nat`, which conflicts with the need for multiple instances. Nevertheless,

6 Design patterns of hierarchies for order structures

we can circumvent this limitation by defining an alias of the head constant [14, Section 2.3]. For example, we can associate the usual order relation `leq` over natural integers to the standard type `nat`, as well as the divisibility relation `dvdn` to an alias of the natural number type denoted by `natdvd`, and that reduces to `nat`, as follows:

```
Structure ordType := OrdType { ord_sort : Type; le : rel ord_sort }.
```

```
Definition natdvd := nat.
```

```
Canonical nat_ordType := { | ord_sort := nat; le := leq |}.
```

```
Canonical natdvd_ordType := { | ord_sort := natdvd; le := dvdn |}.
```

This way, solving the equations `ord_sort ?0 ≐ nat` and `ord_sort ?0 ≐ natdvd` respectively gives the solutions `?0 := nat_ordType` and `?0 := natdvd_ordType`.

In the presence of such multiple instances, we need a mechanism to distinguish overloaded operators applied to distinct instances in printing. However, we cannot properly fine-tune printing by defining notations specialized to specific instances, given that the notation mechanism of Coq is purely syntactic, and syntactically different yet definitionally the same instances may coexist [1]. In order to address this issue, we introduce the *display* parameter to the order structures, to which we can specialize notations. A display is a term of type `disp_t`, whose implementation is explained in Section 3.3. Each group of order structure instances that are supposed to share the same set of order notations in printing should have a dedicated display defined as a opaque constant of type `disp_t`. For example, we define display constants `nat_display` and `dvd_display` to define two sets of order notations specialized for `nat_ordType` and `natdvd_ordType` respectively. Order structures now take a display as a type parameter:

```
Structure ordType (d : disp_t) := OrdType { ord_sort : Type; le : rel ord_sort }.
```

and their instances are redefined to take the corresponding display constant:

```
Canonical nat_ordType : ordType nat_display := { | ord_sort := nat; le := leq |}.
```

```
Canonical natdvd_ordType : ordType dvd_display :=
  { | ord_sort := natdvd; le := dvdn |}.
```

Since `le` now has type $\forall (d : \text{disp_t}) (O : \text{ordType } d), \text{rel } (\text{ord_sort } O)$, we can define notations for `le` specialized for each display.

```
Notation "x ≤ y" := (@le nat_display _ x y).
```

```
Notation "x %| y" := (@le dvd_display _ x y).
```

The display constants `nat_display` and `dvd_display` are made opaque and thus not convertible. Therefore, they prevent us from accidentally simplifying one of the displays to the other display and confusing the specialized order notations. In Section 3.3, we explain how the use of displays fits with duality of order structures.

3.2 Semi-bundled order relation hierarchy

While the combination of aliasing and underloading presented in Section 3.1 provides a workaround to the limitation of canonical structures, it is admittedly restricted to occasional uses of multiple instances over the same carrier type.

In more details, the user may have to carefully annotate their definitions using type aliases, which requires an extra effort. Moreover, a type alias can be accidentally unfolded and may trigger an unexpected inference. As an example, suppose `Total.sort` is the carrier projection of the total order structure. While `nat` in Section 3.1 would have instances of both partial and total order instances, `natdvd` would have only an instance of the former, which means we do not register any solution for `Total.sort ?T ≐ natdvd`. However, Coq

actually solves this equation by turning it into $\text{Total.sort } ?_T \hat{=} \text{nat}$ by unfolding natdvd in the RHS [14, Section 2.3]. Since this issue stems from the convertibility between natdvd and nat and making them inconvertible has a drawback of inserting explicit type casts between them, we argue that it is not specific to Coq and canonical structures.

The order relation hierarchy we describe now is a more robust solution to the multiple instances issue, based on semi-bundled order structures that allow us to infer instances from operators (order relations, meet, join, etc) instead of the carrier type.

Inferring instances from operators. The design pattern of the order relation hierarchy is first illustrated on the semi-bundled partial order structure below:

```
Module RelOrder.
Module Partial.

Record class_of (T : Type) (le : rel T) := Class {
  lexx : reflexive le;      (* := forall x : T, le x x *)
  le_anti : antisymmetric le; (* := forall x y : T, le x y && le y x -> x = y *)
  le_trans : transitive le;  (* := forall y x z : T, le x y -> le y z -> le x z *)
}.

Structure order (T : Type) := Pack { le : rel T; class : class_of le }.

End Partial.
Notation pOrder := Partial.order.
Notation le := Partial.le.
```

In contrast to fully-bundled structures, the structure record `order` above takes the carrier type T as a parameter and bundles the order relation `le` with its class instance. Therefore, the class record `class_of` takes the relation as a parameter and bundles the three axioms of non-strict partial order. The example of multiple instances from Section 3.1 can be redefined as `pOrder` instances as follows.

```
Canonical leq_pOrder := { | Partial.le := leq; Partial.class := ... |}.
Canonical dvdn_pOrder := { | Partial.le := dvdn; Partial.class := ... |}.
```

These instances can be queried by their relations `leq` and `dvdn` instead of the carrier type nat , by solving equations $\text{Partial.le } ?_p \hat{=} \text{leq}$ and $\text{Partial.le } ?_p \hat{=} \text{dvdn}$, respectively.

As an example of inheritance from the `pOrder` structure, we define the semi-bundled meet semilattice structure `meetOrder`.

```
Module Meet.

Record class_of (T : Type) (le : rel T) (meet : T → T → T) := Class {
  base : Partial.class_of le;
  mixin : ∀ x y z, le x (meet y z) = le x y && le x z }.

Structure order (T : Type) :=
  Pack { le : rel T; meet : T → T → T; class : class_of le meet }.

End Meet.
Notation meetOrder := Meet.order.
Notation meet := Meet.meet.
```

The `meetOrder` structure introduces the meet operator `meet`. It appears as a new parameter of the class record and a new field of the structure record.

The following subtyping function `meet_pOrder` from `meetOrder` to `pOrder` allows us to solve equations of the form $\text{Partial.le } ?_p \hat{=} \text{Meet.le } ?_m$ by $?_p := \text{meet_pOrder } ?_m$.

```
Canonical meet_pOrder (T : Type) (ord : meetOrder T) :=
  { | Partial.le := Meet.le ord; Partial.class := Meet.base (Meet.class ord) |}.
```



```
Coercion meet_pOrder : meetOrder  $\rightarrow$  pOrder.
End RelOrder.
```

A generic lemma about the `pOrder` structure, e.g., reflexivity:

```
RelOrder.lexx :  $\forall$ (T : Type) (ord : pOrder T), reflexive (RelOrder.le ord),
```

applies to any abstract or concrete order declared above, as follows.

```
Example lexx_overloading (T : Type) (ord : meetOrder T) (x : T) (n : nat) :
  RelOrder.le ord x x && leq n n && dvdn n n.
(* 'RelOrder.lexx' applies to each element of the conjunction: *)
Proof. by rewrite !RelOrder.lexx. Qed.
```

To avoid confusing several operators bundled in an order structures, e.g., `le` and `meet` of the `meetOrder` structure, we do not declare these projections as implicit coercions and keep them explicit. Therefore, we always use the projection from the poorest structure introducing it to avoid having several constants for the same operator. For example, to talk about the order relation of a `meetOrder` instance, we use `RelOrder.le` and avoid using `RelOrder.Meet.le`.

Underloading semi-bundled operators by fine-tuning simplification. As we have just seen, `RelOrder.le` unifies with any order relation declared, and generic lemmas about semi-bundled order structures automatically apply to them. From the “underloading” point of view, it is natural to use concrete order relations, e.g., `leq` and `dvdn`, in definitions and statements when applicable, and expect `RelOrder.le` applied to a concrete instance, e.g., `leq_pOrder` and `dvdn_pOrder`, to unfold to a concrete order relation by the `simpl` and `cbn` tactics [32], which simplify Coq terms by conversion while keeping the readability by avoiding too much unfolding. However, as we have previously argued, `RelOrder.le` should subsume the role of `RelOrder.Meet.le` in definitions, statements, and goals, and thus, `RelOrder.le` (`meet_pOrder ord`) should not simplify to `RelOrder.Meet.le ord`.

To sum up, `RelOrder.le` should unfold only when the given instance is concrete. Since the subtyping functions such as `meet_pOrder` can be chained, whether the given semi-bundled order structure instance `ord` is concrete or not can be determined as follows:

- if `ord` is a subtyping function applied to another instance `ord'`, `ord` is a concrete instance if and only if `ord'` is a concrete instance, and
- otherwise, `ord` is a concrete instance if and only if it reduces to a constructor application.

In fact, we can encode the above criteria to `Arguments` declarations [29], that allow us to control the simplification tactics of Coq. Since unfolding any subtyping function immediately gives us a constructor application, we instruct them to unfold only when the given instance unfolds to a constructor application. It can be done by using the `!` flag, which exactly instructs the simplification tactics to unfold the definition in question only when the arguments marked with `!` unfolds to constructor applications.

```
Arguments RelOrder.meet_pOrder T !ord.
```

Therefore, the overloaded operators should unfold only when the given instance unfolds to a constructor application as well.

```
Arguments RelOrder.le T !ord x y.
Arguments RelOrder.meet T !ord x y.
```

With the above declarations, `RelOrder.le` simplifies to the underlying order relation of the instance, e.g., `leq`, when applied to a concrete instance, e.g., `leq_pOrder` and `meet_pOrder leq_meetOrder`, while it does not simplify when applied to abstract instances, e.g., `meet_pOrder ord` where `ord` is a variable.

In practice, the `simpl` tactic has a bug that it does not respect the above criteria in the presence of the above `Arguments` commands. For now, we workaroud this issue by

implementing a simplification tactic called `rosimp1` that locally calls the `cbn` tactic for the operators of semi-bundled order structures.

Building fully-bundled structures on top of semi-bundled structures. While the two order hierarchies have their own use cases, the difference of bundling should not prevent us from using results proved in one hierarchy in another. We explain here how to build the ordered type hierarchy on top of that of order relation, and how to enable sharing of lemmas between these hierarchies.

Our approach consists in turning a relatively unbundled structure into a bundled structure by defining a new record type. Here, we define the fully-bundled partial order structure by turning the class record for the `pOrder` structure into its fully-bundled correspondence. We can define the fully-bundled meet semilattice structure following the same pattern.

```
(* In 'Module Order' *)
Module Partial.

Record class_of (T : Type) :=
  Class { le : rel T; rel_class : RelOrder.Partial.class_of le }.

Structure type (d : disp_t) := Pack { sort : Type; class : class_of sort }.

End Partial.
Notation porderType := Partial.type.

Definition le d (T : porderType d) : rel (Partial.sort T) :=
  Partial.le (Partial.class T).
```

The following canonical instance allows us to unify `RelOrder.le` with `Order.le` defined above, and thus enables sharing of lemmas between the order relation and ordered type hierarchies by making lemmas from the former applicable to the latter.

```
Canonical porderType_pOrder d (T : porderType d) := { |
  RelOrder.le := @Order.le _ T;
  RelOrder.Partial.class := Partial.rel_class (Partial.class T) |}.
```

Although applying a lemma from the order relation hierarchy may turn an order type operator, e.g., `Order.le`, into an order relation operator, e.g., `RelOrder.le (porderType_pOrder _)`, in the goal, we can simplify the latter back to the former by the underloading technique previously discussed. To do so, `porderType_pOrder` has to be considered as a concrete instance even if the given `porderType` instance is abstract.

3.3 Duality

A notable case of multiple instances of order structures is that any order \leq has its dual \geq . If \leq is a partial or total order, its dual \geq is a partial or total order as well, respectively. If a set forms a meet-semilattice with respect to \leq , it also forms a join-semilattice with respect to the dual order \geq . Thanks to the duality, many lemmas about order, e.g., facts about join, can be deduced from its dual, e.g., facts about meet.

This section explains how to internalize involutivity of dual constructions in type theory, i.e., to make any order instance T convertible the dual of its dual $(T^d)^d$. While the same methodology has been used in formalizations of category theory [15, 35, 17], we explain its connection to forgetful inheritance (Section 2.3), and demonstrate that it also allows us to make $(T_1 \times T_2)^d$ convertible with $T_1^d \times T_2^d$ where \times denotes the product order.

While we first explain the duality in the case of the order relation hierarchy, we then show how the same technique applies to the ordered type hierarchy by relaxing the opaqueness of displays.

Definitionally involutive duals. We first add the dual versions of all the axioms to the definition of an order structure [15, 35, 17]. For example, the definition of partial order structure includes reflexivity, antisymmetry, and transitivity of the dual relation \geq in addition to those of \leq , as follows.

```
Module Partial.

Record class_of (T : Type) (le : rel T) := Class {
  lexx : reflexive le;                dlexx : reflexive (dual_rel le);
  le_anti : antisymmetric le;       dle_anti : antisymmetric (dual_rel le);
  le_trans : transitive le;          dle_trans : transitive (dual_rel le) }.

Structure order (T : Type) := Pack { le : rel T; class : class_of le }.

End Partial.
Notation porder := Partial.order.
```

where `dual_rel le` is the dual of `le`, i.e., `dual_rel le x y := le y x`. Note that `dual_rel (dual_rel le)` is convertible with `le` since Coq supports η -conversion of functions.

For a given `porder` instance `ord`, its dual `dual_porder ord` can be defined as follows:

```
Definition dual_porderClass (T : Type) (le : rel T) (c : class_of le) := {
  lexx := dlexx c; dlexx := lexx c;
  le_anti := dle_anti c; dle_anti := le_anti c;
  le_trans := dle_trans c; dle_trans := le_trans c }.

Canonical dual_porder (T : Type) (ord : porder T) : porder T :=
  { | le := dual_rel (le ord); class := dual_porderClass (class ord) |}.
```

where the prefix `Partial` is omitted. The first definition `dual_porderClass` replaces the axioms with their duals in the given instance, and applying it twice to a class instance `c` gives its η -expanded version `{ | lexx := lexx c; dlexx := dlexx c; ... |}`. The same discussion applies to the second definition `dual_porder`. By turning `Partial.class_of` into a primitive record [28, 35], we can enable this η -conversion.

The above dualization can also be explained in terms of forgetful inheritance. Forgetful inheritance allows us to recover the contents of a poorer structure instance from a richer structure instance by including the components of the former in the definition of the latter. If we see dual constructions as inheritance, it leads to the idea that we can recover any instance of an order structure from its dual by including all the dual axioms in the definition of the structure. In fact, we can ensure further coherence properties involving dual constructions and usual structure inheritance by including the dualized axioms of a poorer order structure in the definition of a richer order structure.

Interaction of dual and product. This section presents a `porder` instance for the product order such that the dual of the product $(T_1 \times T_2)^d$ is convertible with the product of two dual orders $T_1^d \times T_2^d$.

Firstly, we define the product order relation `prod_le` on the Cartesian product `T1 * T2` from partial order instances `ord1` on `T1` and `ord2` on `T2`, and show that it holds the three axioms of partial order. Note that the computational contents of these proofs are irrelevant here, and thus they can be made opaque, i.e., these proofs can be closed by the `Qed` command.

Section ProdPOrder.

```
Context (T1 T2 : Type) (ord1 : porder T1) (ord2 : porder T2).
```

```
Definition prod_le (le1 : rel T1) (le2 : rel T2) (x y : T1 * T2) :=
  le1 x.1 y.1 && le2 x.2 y.2.
```

```
Fact prod_lexx : reflexive (prod_le (Partial.le ord1) (Partial.le ord2)).
```

```

Fact prod_le_anti : antisymmetric (prod_le (Partial.le ord1) (Partial.le ord2)).
Fact prod_le_trans : transitive (prod_le (Partial.le ord1) (Partial.le ord2)).

End ProdPOrder.

```

Since the dual of `prod_le` is equal to `prod_le` itself where `ord1` and `ord2` are replaced with their duals, the above facts where `ord1` and `ord2` are replaced with their duals gives the reflexivity, antisymmetry, and transitivity of the dual of `prod_le`. Therefore, the `porder` instance for the product order can be defined as follows.

```

Section ProdPOrder.
Context (T1 T2 : Type) (ord1 : porder T1) (ord2 : porder T2).
Context (ord1d := dual_porder ord1) (ord2d := dual_porder ord2).

Definition prod_porderClass :=
  Partial.Class (@prod_lexx _ _ ord1 ord2) (@prod_lexx _ _ ord1d ord2d)
    (@prod_le_anti _ _ ord1 ord2) (@prod_le_anti _ _ ord1d ord2d)
    (@prod_le_trans _ _ ord1 ord2) (@prod_le_trans _ _ ord1d ord2d).

Canonical prod_porder : porder (T1 * T2) := { |
  Partial.le := prod_le (Partial.le ord1) (Partial.le ord2);
  Partial.class := prod_porderClass |}.

End ProdPOrder.

```

In `prod_porderClass` above, each dual pair of axioms, e.g., `lexx` and `dlexx`, uses the same fact, e.g., `prod_lexx`, instantiated with the given orders `ord1` and `ord2`, and their duals `ord1d` and `ord2d`, respectively. Therefore, replacing `ord1` and `ord2` with their dual orders in `prod_porderClass` has the same effect as swapping each pair of dual axioms, i.e., applying `dual_porderClass`, and the equation in question holds definitionally.

Interaction of dual and displays. To enable definitionally involutive duals in the ordered type hierarchy, we have to relax the opaqueness of displays, e.g., any display `d` should be convertible with `dual_display (dual_display d)` where `dual_display d` is the display of the dual of an instance with display `d`. To do so, we define `disp_t` as a primitive record bundling a dualized pair of two units.

```

Record disp_t := Disp { d1 : unit; d2 : unit }.
Definition dual_display (d : disp_t) := { | d1 := d2 d; d2 := d1 d |}.

```

4 Finite lattices and sublattices

In this section, we deal with the formalization of substructures of order structures, with a particular focus on finite sublattices. While the design patterns introduced in Section 3 allow us to handle several orders over a common carrier type, the formalization of sublattices raises additional issues, since the meet and join of every sublattice of a lattice L must be consistent with the meet and join of L . This situation is reminiscent of that of the formalization of finite groups and subgroups, or of vector spaces and subspaces. In the case of groups, the approach developed in [13, 18] consists in introducing an ambient structure that provides a set of operators (multiplicative law, inverse, unit element) shared by all groups and subgroups. Groups are then defined as sets closed under the operators, and all groups can be somehow thought of as subgroups. We take inspiration from this approach. However, one major difficulty in our case is that, even if they share a partial order relation, two distinct lattices

may not have the same meet and join in general,¹ so that these operators cannot be fixed as the meet and join of an ambient structure. This is what leads us to introduce a new order structure called *prelattice*, whose purpose is to provide a common way to build meet and join operators for all finite lattices, on top of a common order relation. The Coq code discussed in this section can be found in the file `finlattice.v` of the project `Order`.

4.1 Prelattices

The Coq formal structure of *prelattice* inherits from partial orders, two operators called `premeet` and `prejoin`, and consistency axioms.

```
Module Prelattice.
Record mixin_of (T : porderType) := Mixin {
  premeet : {fset T} → T → T → T;
  prejoin : {fset T} → T → T → T;

  premeet_min  : ∀S x y, x ∈ S → y ∈ S → premeet S x y ≤ x ∧ premeet S x y ≤ y;
  premeet_inf  : ∀S x y z, x ∈ S → y ∈ S → z ∈ S →
    z ≤ x → z ≤ y → z ≤ premeet S x y;
  premeet_incr : ∀S U x y, S ⊂ U → x ∈ S → y ∈ S →
    premeet U x y ∈ S → premeet S x y ≤ premeet U x y;


  prejoin_max  : (* omitted *) ;
  prejoin_sup  : (* omitted *) ;
  prejoin_decr : (* omitted *) ;
}.

Record class_of (T : Type) := Class {
  base : Order.POrder.class_of T;
  mixin : mixin_of (@POrder.Pack disp_tt base);
}.

Structure type (d : disp_t) := Pack { sort; _ : class_of sort }.
End Prelattice.
```

Notation "prelatticeType d" := (Prelattice.type d).

The two operators `premeet` `prejoin` : {fset T} → T → T → T are parametrized by a finite subset of points; we recall that the type {fset T}, provided by the `finmap` library [7], represents the finite sets over the type T. The operator `premeet` has to satisfy the properties `premeet_min`, `premeet_inf` and `premeet_incr`. Intuitively, given a finite set `S` : {fset T}, the function `premeet S` is intended to play the role of a meet operator over `S`. Indeed, the first two properties `premeet_min` and `premeet_inf` corresponds to the fact that, for all `x`, `y` ∈ `S`, `premeet S x y` is precisely the meet of `x` and `y` in `S`, as long as `premeet S x y` belongs to `S`. The property `premeet_incr` can be thought as a generalization of the constraint that the meet operator of a lattice `S` must coincide with the meet operator of a lattice `U` when `S` is a sublattice of `U`. More precisely, assume that `S`, `U` satisfies `S ⊂ U`, and that `S` is closed under the meet operator associated with `U`, meaning that for all `x`, `y` ∈ `S`, we have `premeet U x y` ∈ `S`. Then, the two properties `premeet_min` and `premeet_inf` ensure that

¹ For instance, in the partial order with Hasse diagram , consider the two lattices respectively formed by the black, blue and green elements, and the black, blue and orange elements. The join of the two blue elements is the green element in the former lattice, while it is the orange element in the latter.

$\text{premeet } U \ x \ y \leq \text{premeet } S \ x \ y$. The purpose of the property `premeet_incr` is to provide the converse inequality, so that the two operators `premeet S` and `premeet U` coincide over `S`. In other words, we can use `premeet S` and `premeet U` interchangeably, as expected. The `prejoin` operator is axiomatized in a dual way to `premeet`. Prelattices are preserved by duality, as exchanging the operators `premeet` and `prejoin` yields a prelattice over the dual order. We use the design pattern described in Section 3.3 in order to support definitionally involutive duals.

While the existence of `premeet` and `prejoin` operators can appear as a strong assumption, we claim that prelattices actually encompass semilattices. For instance, a meet-semilattice (T, \leq, \wedge) yields a prelattice in which the `premeet` operator is simply given by the ambient meet operator \wedge (independently of the subset `S`), and the `prejoin` operator `prejoin S x y` is defined the meet of the (finitely many) elements $z \in S$ satisfying $z \geq x$ and $z \geq y$. Symmetrically, every join-semilattice gives rise to a prelattice. Since prelattices are closed under Cartesian product, they encompass any finite product of meet- or join-semilattices (which show that prelattices strictly contain semilattices).

4.2 Finite sublattices of a prelattice

In this setting, a finite lattice over a prelattice `T : prelatticeType disp` is formalized as a nonempty subset `elts : {fset T}` that is closed under the operations `premeet elts` and `prejoin elts`:

```
Record finLattice := FinLattice {
  elts          : {fset T};
  premeet_closed : ∀ x y, x ∈ elts → y ∈ elts → premeet elts x y ∈ elts;
  prejoin_closed : ∀ x y, x ∈ elts → y ∈ elts → prejoin elts x y ∈ elts;
  fl_inhabited   : elts ≠ fset0 }.
```

While we use the term *finite lattices*, structures of type `finLattice` can actually thought of as sublattices of the prelattice. Moreover, as in the works [13, 18] on finite groups, there is no distinction between the type of finite lattices and that of sublattices. For convenience, given a finite lattice `S : {finLattice T}`, we introduce notation `\meet_S` and `\join_S` for `premeet S` and `prejoin S` respectively, and `\top_S` and `\bot_S` for the top and bottom elements (resp. defined as the `\meet_S` and `\join_S` of all elements in `S`).

Every term `S : {finLattice T}` is equipped with a lattice structure with top and bottom in the sense of the ordered type hierarchy, over the finite type `fset_sub_type S` of elements of the set `S`.² In this way, we recover all the theory of lattices developed in `order.v`, for instance, statements like

```
Lemma meetACA (x y z t : fset_sub_type S):
  meet (meet x y) (meet z t) = meet (meet x z) (meet y t).
```

where `meet` refers here to the meet operator of the type `fset_sub_type S`. Such statements are strongly bound to the considered lattice `S`, since they take as input terms of the corresponding finite type. From a practical perspective, our goal is to manipulate several finite lattices at the same time. However, casting elements from one finite type to another would quickly become tedious. The rationale behind having all lattices sharing a common prelattice is to manipulate elements `x, y, ...` over the same carrier type `T`, possibly with the assumption that they belong to some lattice (thought of as a finite set). This is why we duplicate the

² The coercion `fset_sub_type` from finite sets to finite types is equivalent to $\{x \mid x \in S\}$, provided by `finmap`, inserted automatically and not displayed. We make it explicit here for the sake of readability.

statements of the theory of `tbLatticeType` from `order.v` and put them into a form that is more adapted to this need, for example:

```
Lemma fmeetACA (S : {finLattice T}) x y z t :
  x ∈ S → y ∈ S → z ∈ S → t ∈ S →
    \meet_S (\meet_S x y) (\meet_S z t) = \meet S (\meet_S x z) (\meet_S y t).
```

The latter statements can be proved in a straightforward way from the corresponding ones in the theory of `tbLatticeType`.

A finite lattice $S : \{\text{finLattice } T\}$ gives rise to a dual finite lattice over the dual prelattice. The latter is denoted as $S^{\text{fd}} : \{\text{finLattice } T^{\text{d}}\}$, and is simply obtained by exchanging the fields `premeet_closed` and `prejoin_closed`. Like for the structures described in Section 3, we exploit duality to get straightforward proofs of some statements from the symmetric ones.

We introduce the relation `sublattice S U` which states that the finite lattice S is a sublattice of the finite lattice U . This definition only requires that S and U are finite sets:

```
Definition sublattice (S U : {fset T}) :=
  [∧ S ⊂ U, {in S & S, ∀ x y, premeet U x y ∈ S}
    & {in S & S, ∀ x y, prejoin U x y ∈ S}].
```

The property `sublattice S U` is equivalent to the fact that the operators `\meet_S` and `\meet_U` (resp. `\join_S` and `\join_U`) coincide over elements of S . Therefore, the fact that `sublattice S U` holds for a certain $U : \{\text{fset } T\}$ is a sufficient condition for nonempty $S : \{\text{fset } T\}$ to be a finite lattice (see `Lemma premeet_closed_sub` and `prejoin_closed_sub` in `finlattice.v`).

A notable class of sublattices are lattice intervals. A *lattice interval* consists of the elements of a lattice ranging between two of its elements. We use the notation $[\langle a; b \rangle]_S$ where $a, b : T$ and $S : \{\text{finLattice } T\}$ for such intervals. In order to show that such intervals are finite lattices, we simply prove that $[\langle a; b \rangle]_S$ is closed under the operators `\meet_S` and `\join_S`, which corresponds to `sublattice [\langle a; b \rangle]_S S`. Intervals also enjoy some composability properties. For instance, under mild assumptions, the interval $[\langle a; b \rangle]_{[\langle A; B \rangle]_S}$ of an interval $[\langle A; B \rangle]_S$ of a finite lattice S is equal to the interval of S with the same bounds a and b :

```
Lemma mono_itv (S : {finLattice T}) (A B a b : T) : A ∈ S → B ∈ S → A ≤ B →
  a ∈ [\langle A; B \rangle]_S → b ∈ [\langle A; B \rangle]_S → a ≤ b → [\langle a; b \rangle]_{[\langle A; B \rangle]_S} = [\langle a; b \rangle]_S.
```

or that the dual of an interval of S is the reversed interval of the dual finite lattice:

```
Lemma dual_itv (S : {finLattice T}) a b : ([\langle a; b \rangle]_S)^{\text{fd}} = [\langle b; a \rangle]_{(S^{\text{fd}})}.
```

A remarkable feature of our design pattern is to reduce the proof of such equality statements between finite lattices to that of the equality of the underlying sets (which is a routine verification in the case of the two lemmas `mono_itv` and `dual_itv`). Indeed, the equality of the finite sets suffices to make the join and meet operators of the two lattices coincide: both are given by the operators `premeet` and `prejoin` applied to the same set. Moreover, the properties `premeet_closed` and `prejoin_closed` can be equivalently expressed as Boolean predicates (quantified variables range over a finite set). In this way, they enjoy proof irrelevance.

We finally define morphisms and isomorphisms of finite lattices. We skip their description as the formalization is standard.

4.3 Application to the face lattice of polyhedra

A *polyhedron* \mathcal{P} is the set of points of \mathbb{R}^n satisfying a finite system of affine linear inequalities, i.e., of the form $\sum_{j=1}^n a_{ij}x_j \leq b_i$ for $i = 1, \dots, m$, where the a_{ij} and b_i are real. A *face* of \mathcal{P} is a set of the form $\{x \in \mathcal{P} : \forall i \in I, \sum_{j=1}^n a_{ij}x_j = b_i\}$ where I is a subset of $\{1, \dots, m\}$. The faces of \mathcal{P} constitute a finite lattice, in which the partial order is the set inclusion, the meet

operator is the set intersection, and the top and bottom elements are \mathcal{P} and \emptyset . Sublattices are ubiquitous when manipulating polyhedra and their faces, as the face lattice of a face of a polyhedron is an interval sublattice of the face lattice of the polyhedron. A formalization of the basic properties of faces have been carried out in the project `Coq-Polyhedra` [4]. However, the original design of lattices introduced unnecessary complications in proofs. Indeed, every lattice had to be associated to its own type (following the type-as-carrier design). This caused an extensive use of subtypes and casts which are not relevant from a mathematical perspective. Moreover, this often led to duplicated statements, some ranging over finite sets of polyhedra underlying to face lattices, and others over the lattices themselves.

These issues have been solved by rewriting the formalization using the structures of prelattices and finite lattices previously described (cf. `poly_base.v` in the project `Coq-Polyhedra`). The ambient prelattice over the type `'poly_n` of polyhedra in \mathbb{R}^n simply arises from the semilattice structure induced by the set inclusion, the set intersection, and the top element \mathbb{R}^n . Given a polyhedron P , we prove that the set `face_set P` of its faces are closed under the `premeet` and `prejoin` operators, which only exploits that faces are closed under set intersection. In this way, we associate to every $P : 'poly_n$ the term `face_lattice P` : `{finLattice 'poly_n}`. Several basic properties of faces then easily write in the formalism of finite lattices, e.g.,

Lemma `face_lattice_of_face` ($P Q : 'poly[R]_n$) :
 $Q \in \text{face_lattice } P \rightarrow \text{face_lattice } Q = [\langle [\text{poly}0]; Q \rangle]_{(\text{face_lattice } P)}$.

where `[poly0]` stands for the empty polyhedron. This corresponds to the aforementioned property on the face lattice of a face.

A central property of polytopes (bounded polyhedra) is that their face lattices are closed under taking intervals, i.e., any interval of the face lattice of a polytope is the face lattice of another polytope (up to isomorphism). This is stated as:

Theorem `closed_by_interval` ($P : 'poly_n$) $F F'$:
 $\text{bounded } P \rightarrow F \in \text{face_lattice } P \rightarrow F' \in \text{face_lattice } P \rightarrow F \subset F' \rightarrow$
 $\text{exists2 } Q : 'poly_n, \text{ bounded } Q \ \& \ \text{isof } [\langle F; F' \rangle]_{(\text{face_lattice } P)} (\text{face_lattice } Q)$.

where `isof S S'` means that the finite lattices S and S' are isomorphic. The proof of this statement is now much closer to the usual hand-and-paper proof thanks the new design of finite lattices. It relies on the application of the following inductive principle over finite lattices. Consider a property $A : \{\text{finLattice } T\} \rightarrow \text{Prop}$, and suppose that

$A_incr : \forall S, \forall x, \text{atom } S \ x \rightarrow A \ S \rightarrow A \ [\langle x; \backslash\text{ftop}_S \rangle]_S$.
 $A_decr : \forall S, \forall x, \text{coatom } S \ x \rightarrow A \ S \rightarrow A \ [\langle \backslash\text{fbot}_S; x \rangle]_S$.

where `atom S x` (resp. `coatom S x`) stands for the fact that x is an atom (resp. a coatom) of S , meaning that there is no element between `\fbot_S` and x (resp. between x and `\ftop_S`). Then, it can be proved (see `Lemma itv_induction` in `finlattice.v`) that the property A holds for any interval of S ; the proof is done by first assuming that A_incr , and dealing later with A_decr using duality. In the case of `Theorem closed_by_interval`, we use the following property $A \ S := \text{exists2 } Q : 'poly_n, \text{ bounded } Q \ \& \ \text{isof } S (\text{face_lattice } Q)$. In order to show A_decr , we use the fact that any coatom x corresponds to a face Q' of Q , so that the interval `[\langle \backslashbot_S; x \rangle]_S` is isomorphic to `[\langle [\text{poly}0]; Q' \rangle]_{(\text{face_lattice } Q)}`. The latter is exactly `face_lattice Q'` owing to `Lemma face_lattice_of_face`. In other words, the polytope Q' is a witness of $A \ [\langle \backslashbot_S; x \rangle]_S$. The proof of A_incr follows from the vertex figure construction formalized in [4].

5 Related work

To our knowledge, this is the first work mixing different bundlings in order to change the indexation key (carrier or operator), based on the local preferences. This is also a step towards solving Problem #10 from Tom Hales’s “A Review of the Lean Theorem Prover” [16], which states in particular that formalization should not commit to a particular bundling, and that overlapping instances should be distinguishable. We now discuss the choices made in other systems or library based on dependent type theory.

The `MathComp` library already has structures indexed on operators and other based on the carrier type, but they do not overlap. Indeed, structures where inference is based on the operators are all monoids, while carrier type inference based structures start with finite groups and abelian groups. As such, introducing a hierarchy where each structure can be inferred on either is a contribution of our work, as well as the mechanisms for unloading.

The other existing library for which inference can be done either on the type or the operators of a structure is the `Math Classes` library [25], which uses fully unbundled type-classes [24, 30] which are parametrized on both the carrier and the operations and contain axioms as fields. This leads to a formalization style with long contexts, and commits the user to using this particular bundling style for inference to go through.

In `Lean/mathlib` [34, 6], the primary definition for each structure is a class on the carrier, and in some classes the structure is even split in several bits, for example a module is an additive abelian group which also has an action from a ring. Order relations are declared through a similar mechanism and inference is based on the carrier type. Bundling the carrier is also possible, but rather meant to declare the type of the object in the category associated with the structure, since typeclass inference cannot deal with fully bundled structures.

In the standard library of `Agda` [9], the two same bundling are in use, but inference is based on the opening of partially instantiated module or in some cases, typeclass inference, which is based on the carrier type. Other work [3], which target `Agda` for their experimentation also focus on having multiple ways of packaging structures seamlessly. This work seems complementary to ours since they design a language for not committing to a particular bundling. However, they do not address the interplay between inference and modular bundling.

Concerning the formalization of finite lattices and sublattices, we already compared our approach with the one used for finite groups in `MathComp` in Section 4. In `Lean/mathlib` [6], subobject are handled in the same way as morphisms, in a bundled way, which they underline is crucial, but losing inference in the context of typeclass resolution. In comparison, with canonical structures, we retain both bundling and inference.

6 Conclusion

The two-layer design pattern opens the way to more generic treatment of overloading in the presence of multiple instances of the same structure.

However, this design pattern introduces even more boilerplate to an already heavy setup. The next step to integrate these two layers and automate the boilerplate is to generate it through a domain specific language. `Hierarchy Builder` [8] provides such a language, and could be adapted with little extra information to compile to the design pattern we propose in this paper instead. This would systematize the work done in this paper to all structures, not just order structures, and provide the flexibility of handling multiple instances everywhere.

Moreover, with the integration to `Hierarchy Builder`, we can envision changing the flow of inference in order to make inference pick up display information as well, hence enabling a seamless integration between different bundlings.

References

- 1 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 *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020. doi:10.1007/978-3-030-51054-1_1.
- 2 Ron Aharoni and Tamás Fleiner. On a lemma of scarf. *J. Comb. Theory, Ser. B*, 87(1):72–80, 2003. doi:10.1016/S0095-8956(02)00028-X.
- 3 Musa Al-hassy, Jacques Carette, and Wolfram Kahl. A language feature to unbundle data at will (short paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019*, page 14–19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3357765.3359523.
- 4 Xavier Allamigeon, Ricardo D. Katz, and Pierre-Yves Strub. Formalizing the face lattice of polyhedra. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/lmcs-18(2:10)2022.
- 5 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009. doi:10.1007/978-3-642-03359-9_8.
- 6 Anne Baanen. Use and abuse of instance parameters in the Lean mathematical library. In *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.4.
- 7 Cyril Cohen and Kazuhiko Sakaguchi. A finset and finmap library, 2015–. URL: <https://github.com/math-comp/finmap>.
- 8 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.FSCD.2020.34.
- 9 Agda development team. Agda 1.7.2 standard library. URL: <https://github.com/agda/agda-stdlib>.
- 10 Tamás Fleiner. A fixed-point approach to stable matchings and some applications. *Math. Oper. Res.*, 28(1):103–126, 2003. doi:10.1287/moor.28.1.103.14256.
- 11 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009. doi:10.1007/978-3-642-03359-9_23.
- 12 Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. A constructive algebraic hierarchy in Coq. *J. Symb. Comput.*, 34(4):271–286, 2002. doi:10.1006/jsco.2002.0552.
- 13 Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2007. doi:10.1007/978-3-540-74591-4_8.
- 14 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *J. Funct. Program.*, 23(4):357–401, 2013. doi:10.1017/S0956796813000051.
- 15 Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant category-theory library in Coq. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July*

- 14-17, 2014. *Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2014. doi:10.1007/978-3-319-08970-6_18.
- 16 Thomas Hales. A review of the lean theorem prover. Blog post, 2018. URL: <https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/>.
- 17 Jason Z. S. Hu and Jacques Carette. Formalizing category theory in Agda. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 327–342. ACM, 2021. doi:10.1145/3437992.3439922.
- 18 Assia Mahboubi. The rooster and the butterflies. In *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-39320-4_1.
- 19 Assia Mahboubi and Enrico Tassi. Canonical structures for the working Coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2013. doi:10.1007/978-3-642-39634-2_5.
- 20 Robert Pollack. Dependently typed records in type theory. *Formal Aspects Comput.*, 13(3-5):386–402, 2002. doi:10.1007/s001650200018.
- 21 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 292–301. ACM Press, 1997. doi:10.1145/263699.263742.
- 22 Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 1999. URL: <https://tel.archives-ouvertes.fr/tel-00523810>.
- 23 Kazuhiko Sakaguchi. Validating mathematical structures. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2020. doi:10.1007/978-3-030-51054-1_8.
- 24 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.
- 25 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.
- 26 Bernd Sturmfels. *Gröbner Bases and Convex Polytopes*. Number 8 in Univ. Lectures Series, Providence, Rhode Island, 1996. American Mathematical Society, 1996.
- 27 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2022. the PDF version with numbered sections is available at <https://doi.org/10.5281/zenodo.7313584>. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/>.
- 28 The Coq Development Team. *Section 2.1.8 “Record types”*. In [27], 2022. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/language/core/records>.
- 29 The Coq Development Team. *Section 2.2.5 “Setting properties of a function’s arguments”*. In [27], 2022. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/language/extensions/arguments-command>.
- 30 The Coq Development Team. *Section 2.2.7 “Typeclasses”*. In [27], 2022. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/addendum/type-classes>.
- 31 The Coq Development Team. *Section 2.2.8 “Canonical Structures”*. In [27], 2022. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/language/extensions/canonical>.

- 32 The Coq Development Team. *Section 3.1.3 “Reasoning with equalities”*. In [27], 2022. URL: <https://coq.inria.fr/distrib/V8.16.0/refman/proofs/writing-proofs/equality#reasoning-with-equalities>.
- 33 The Mathematical Components project. The mathematical components repository, 2015–. URL: <https://github.com/math-comp/math-comp>.
- 34 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.
- 35 Amin Timany and Bart Jacobs. Category theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPICs*, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.FSCD.2016.30.
- 36 Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *J. Funct. Program.*, 27:e10, 2017. doi:10.1017/S0956796817000028.