Use and abuse of instance parameters in the Lean mathematical library

Anne Baanen

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands.

Corresponding author(s). E-mail(s): t.baanen@vu.nl;

Keywords: formalization of mathematics, dependent type theory, typeclasses, algebraic hierarchy, Lean prover

1 Introduction

An essential part of a mathematical library is a discipline for representing structures, such as an algebraic hierarchy including monoids, groups, rings and fields, or a hierarchy of spaces including topological spaces and metric spaces. Many design patterns have been proposed to enable the theory of a general structure (such as monoids) to be applied seamlessly to more specific structures (such as fields), including canonical structures in Coq [1], locales in Isabelle [2, 3], unification hints in Matita [4] and attributed types in Mizar [5].

The Lean mathematical library mathlib [6] has settled on the use of the *typeclass* [7] pattern for representing structures, implemented through Lean's mechanism of *instance parameters* [8, 9]. Typeclasses were invented by Wadler to provide ad hoc polymorphism in Haskell [7]. Similar mechanisms can now be found in programming languages including Idris [10], Rust [11, Chapter 6.11] and Scala [12], and interactive theorem provers including Agda [13], Coq [14] and Isabelle [15]. Class-based libraries of comparable complexity to mathlib have previously been developed by Hölzl, Immler and Huffman [16] for analysis in Isabelle/HOL and by Spitters and Van der Weegen [17] for an algebraic hierarchy in Coq. As of March 2023, mathlib contains over 900 classes and over 12000 instances.

In the various languages implementing a mechanism analogous to typeclasses, there is also a variety of syntax and semantics for the parts of this mechanism. In this paper, I will try to avoid confusion by using the terminology "instance parameters" when emphasising Lean 3's specific implementation, while "typeclass" refers to a design pattern that is implemented in Lean through instance parameters. This is analogous to the distinction drawn in the Scala documentation between its mechanism of "implicit parameters" and the typeclass pattern built with that mechanism.

This paper combines my original research with a survey of the mathlib community's experience in developing a class-based hierarchy, emphasising design patterns, issues arising from the use of classes and the trade-offs available for resolving these issues. Researching how to develop and maintain a large library of mathematical structures has led me to develop a number of typeclass-based patterns in Lean that have been added to mathlib. Given that mathlib is expected to upgrade soon from Lean 3 to Lean 4 [18], an upgrade that promises changes in Lean's support for typeclasses [19], now seems a good time to discuss what is achievable with the current mechanism. I have organized this paper around a representative example for each topic, based on mathlib source code. Unabridged, interactive versions of the listings are available at https://github.com/lean-forward/mathlib-classes.

This paper is an extended version of a paper published at ITP 2022 [20]. Apart from minor additions and updates spread throughout the text, this version contains three new sections:

- Section 5 explains mathlib's approach to instances with conflicting notation on the same type.
- Section 9 illustrates implementation complexities in the presence of *out* parameters.
- Section 13 discusses how typeclasses influence tactic design patterns in mathlib and vice versa.

2 Basic instance parameters in Lean 3

Lean provides the typeclass pattern through *instance parameters*, a mechanism and implementation closely resembling the same facilities in Coq [14]. Like Coq, Lean is a dependently typed language based on the calculus of constructions. Lean has a hierarchy of universes `Sort 0 : Sort 1 : Sort 2 : ...`, where `Sort 0` is more often written as `Prop` and `Sort (u + 1)` is written `Type u` or, leaving `u` implicit, `Type*`. The bottom universe `Prop` is an impredicative type of propositions that has definitional proof irrelevance.

Let us start with the following two Lean declarations, not found in mathlib, showing the main forms that parameters can take in Lean.

```
def sub {A : Type*} [add_group A] (a b : A) : A :=
add a (neg b)

lemma sub_eq_add_neg {A : Type*} [add_group A] (a b : A) :
  sub a b = add a (neg b) := by refl
```

The round brackets mark explicit parameters to be supplied when applying the lemma, the curly brackets mark implicit parameters inferred through unification, while square brackets mark the instance parameters (for which supplying a name is optional); these are used here to specify a typeclass constraint on the type `A`. Thus the term `sub a b` specifies only the `(a b : A)` parameters to `sub`, leaving the remaining parameters to be supplied by the elaborator. These parameters are then passed on to the calls to `neg` and `add` in the body of `sub`. There is no relevant distinction between the keywords `def` and `lemma` for our purposes, apart from indicating whether the declaration exists in a `Type` or the `Prop` universe.

The elaborator supplies values to instance parameters through *instance synthesis*: the parameters to the current declaration and all declarations in the global context which have been marked with the keyword `instance` are considered in turn as candidates. Each candidate instance is type-checked against the goal, and the first candidate where the types unify is returned. For example, defining `add_group ℤ` instance allows us to subtract two integers using the `sub` operator we defined above:

```
instance : add_group ℤ := sorry -- implementation omitted
lemma subtraction_example : (sub 42 37 : ℤ) = 5 := by refl
```

Instance parameters are considered a form of implicit parameters, and can thus be made explicit using the `@` operator:

```
#check sub -- sub : ?M_1 → ?M_1 → ?M_1
#check @sub -- sub : Π {A : Type*} [add_group A], A → A → A
```

Here `?M_1` stands for a free metavariable that the elaborator could not (yet) fill in.

Instance declarations can themselves have their own instance parameters. For example, the subsets of a monoid form a monoid under pointwise operations, which we can express as

```
instance pointwise_monoid {A : Type*} [monoid A] : monoid (set A) :=
{ mul := λ X Y, { (x * y) | (x ∈ X) (y ∈ Y) },
  mul_assoc := λ _ _ _, set.image2_assoc mul_assoc,
  ..sorry /- further fields omitted -/ }
```

When the synthesis of an instance of `monoid (set A)` tries to apply the `pointwise_monoid` instance, the elaborator will recurse and try to synthesize the `monoid A` dependency; if this instance is not found, search backtracks and continues with the next candidate `monoid (set A)` instance; if the entire search tree is exhausted, synthesis has failed. Since instances' types are unified with the goal, we can view the synthesis algorithm as performing recursion on the term structure of the goal.

Instance synthesis can be triggered by unification and trigger unification in turn. The order in which these these two mechanisms are interleaved is important in handling complicated situations, as discussed in Section 9.

2.1 Class definitions

The classes themselves are expressed as records, i.e. dependent tuples, with the class fields as projections taking instance parameters. Classes use the same syntax as record types in Lean, only differing in using the keyword `class` instead of `structure`. Dependent types mean data- and proof-carrying fields are expressed in the same way; Section 8 discusses some usage differences between data and proofs. Thus, a definition for `add_group` could look like:

```
class add_group (A : Type*) : Type* :=
(zero : A) (neg : A → A) (add : A → A → A)
(add_assoc : ∀ (x y z : A), add x (add y z) = add (add x y) z)
(zero_add : ∀ (x : A), add zero x = x)
(neg_add : ∀ (x : A), add (neg x) x = zero)
```

The projections of a class automatically use instance parameters, generating the declarations:

```
def add_group.zero {A : Type*} [h : add_group A] : A := h.1
def add_group.neg {A : Type*} [h : add_group A] : A → A := h.2
def add_group.add {A : Type*} [h : add_group A] : A → A → A := h.3

def add_group.add_assoc {A : Type*} [h : add_group A] :
  ∀ (x y z : A), add_group.add x (add_group.add y z) =
    add_group.add (add_group.add x y) z) := h.4
-- and so on.
```

The instance synthesis algorithm also allows instances for non-record types. In practice most classes in mathlib are record types and indeed Lean 4 will make the use of record types for classes obligatory to simplify the elaborator.

2.2 Subclassing

The mechanisms described above result in two patterns for subclass definitions, with important distinctions in semantics. *Unbundled subclasses* take superclasses as instance parameters to the class declaration. To define abelian groups as a subclass of additive groups, we write

```
class add_comm_group (A : Type*) [add_group A] : Type* :=
(add_comm : ∀ (x y : A), add x y = add y x)
```

Elaboration of the expression `add_comm_group A`, e.g. in a parameter `[add_comm_group A]`, requires the synthesis of an `add_group A` instance occurring as parameter to `add_comm_group`. We make this instance available by adding it as another instance parameter, so all results on abelian groups take the two instance parameters `[add_group A] [add_comm_group A]`; long inheritance chains cause long parameter lists. Similarly, declaring an `add_comm_group A` instance requires a previous declaration of an `add_group A` instance.

In contrast, *bundled subclasses* make use of instance synthesis to access the superclass, only requiring an instance parameter for the subclass. A bundled subclass contains an instance of its superclass as a record field. Lean's `extends` keyword provides syntax sugar for the construction:

```
class add_comm_group (A : Type*) extends add_group A :=
(add_comm : ∀ (x y : A), add x y = add y x)
```

This has analogous effects to writing

```
class add_comm_group (A : Type*) : Type* :=
(to_add_group : add_group A)
(add_comm : ∀ (x y : A),
  @add_group.add A to_add_group x y =
    @add_group.add A to_add_group y x)
-- Register the projection as an instance:
attribute [instance] add_comm_group.to_add_group
```

When we look at the synthesis of an `add_group A` instance, we can see the instance `add_comm_group.to_add_group` triggers a recursive search for an `add_comm_group A` instance; in this way subclass instances automatically provide access to declarations on the superclass. Both subclass patterns are used in mathlib; the following sections discuss reasons for choosing one over the other in a given situation.

2.3 Extensions of the typeclass pattern

Beyond expressing the basic typeclass patterns above, Lean's instance parameters provide a considerable amount of flexibility. Classes do not have to be parametrized over exactly one type, unlike what the phrase "typeclass" suggests, and the only restriction on `instance` declarations is that the head symbol of its return type is declared to be a class. In Haskell terminology, all the following extensions are allowed: constrained class method types, flexible contexts, flexible instances, incoherent instances, multi-parameter classes (including nullary classes), overlapping instances, quantified constraints, type synonym instances, undecidable instances. Most of these extensions find uses throughout mathlib.

A notable difference compared with Agda is that Lean allows overlapping instances, thus enabling the diamond inheritance pattern of Section 4. Compared with Isabelle, Lean permits multi-parameter classes, as we will see in Section 6; on the other hand Isabelle provides multi-parameter hierarchies through locales [3]. Compared with Coq, Lean adds a limited syntax for expressing functional dependencies [21], also discussed in Section 6.

Since Lean is a dependently typed language, parameters to classes are not restricted to types. For example, mathlib uses a typeclass parametrized over a natural number p to express the characteristic of a ring:

```
class char_p (R : Type*) [semiring R] (p : ℕ) : Prop :=
(cast_eq_zero_iff : ∀ (x : ℕ), (coe x : R) = 0 ↔ p | x)
```

The `char_p` class is an example of *a proof mixin*. Section 10 discusses this pattern further.

Since type-checking is used to match candidate instances with a synthesis goal, the synthesis algorithm works up to definitional equality. For example, since `2+2 : ℕ` is definitionally equal to `4`, Lean finds can solve the goal `char_p (zmod 4) (2 + 2)` using the instance `zmod.char_p 4 : char_p (zmod 4) 4`. Thus, typeclasses in Lean are coupled to a built-in syntactic notion of equality.

This combination of features means that instance parameters can be used for small-scale automation [22], since the instance synthesis mechanism provides a search tactic for definite Horn clauses: a clause of the form $C = (\bigwedge_i P_i(\vec{t})) \to Q(\vec{t})$, where the P_i

and Q are (not necessarily distinct) predicates and $\vec{t}$ a vector of terms, translates to an instance of the form `instance C [P_1 t_1 ... t_k] ... [P_n t_1 ... t_k] : Q t_1 ... t_k`. The branching depth-first nature of the synthesis algorithm has to be kept in mind during design in order to assure acceptable performance, as we will see in Section 12.

3 `has_mul`: notation typeclass

The typeclass pattern is used throughout mathlib for operator overloading, in much the same role that classes were originally introduced in Haskell. Generally, such a notation typeclass has one type parameter `α : Type*` and contains fields which carry only data. A basic example is the definition of the multiplication operator `*` in core Lean:

```
class has_mul (α : Type*) := (mul : α → α → α)
infix * := has_mul.mul
```

The `infix` command adds the notation `a * b` for `has_mul.mul a b`.

These notations are not directly coupled to the algebraic hierarchy: the `has_inv` class providing $^{-1}$ notation for the multiplicative inverse does not have any fields requiring a multiplicative group structure. However, in practice such notations are often provided through inheritance from an instance of a proof-carrying class in the algebraic hierarchy.

Lean uses classes to implement implicit coercions in the style of Saïbi [23]. Whenever the elaborator encounters a term `t : A` that is instead expected to have type `B`, it replaces `t` with `@coe A B _ t`, where the `_` marks an instance parameter of type `has_lift_t A B`. Similarly, when a term `f : F` produces a type error because it is expected to have a dependent function type, it is replaced with `coe_fn f` (where `coe_fn {F A} [has_coe_to_fun F A]` has type `Π (f : F), A f`), and when `t` is expected to be of the form `Sort u` (that is, either `Type v` if `u = v+1`, or `Prop` if `u = 0`), it is replaced with `coe_sort t` (where `coe_sort {A} [has_coe_to_sort A] : Sort u`). Such coercions are essential for mathlib's design of morphisms and subobjects, as we will see in Section 7.

4 `comm_monoid`: algebraic hierarchy class

The algebraic hierarchy in mathlib is built using typeclasses, based on the notation typeclasses discussed in the previous section. Similar class-based hierarchies exist in mathlib for topics including orders, topology and analysis, and all the hierarchies are connected throughout. As an example, the `comm_monoid` typeclass is implemented in mathlib essentially as follows:

```
set_option old_structure_cmd true -- explained below

class semigroup (G : Type*) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
```

```
class mul_one_class (M : Type*) extends has_one M, has_mul M :=
(one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)

class comm_semigroup (G : Type*) extends semigroup G :=
(mul_comm : ∀ a b : G, a * b = b * a)

class monoid (M : Type*) extends semigroup M, mul_one_class M

class comm_monoid (M : Type*) extends monoid M, comm_semigroup M
```

While `comm_monoid` is considered to sit low in the `mathlib` algebraic hierarchy, its definition already depends on seven ancestor classes in a complicated diamond inheritance pattern. Multiple inheritance paths result in two instances of `has_mul` for each `monoid` instance, thus requiring support for overlapping instances. We can also see that `mathlib` prefers bundled inheritance in the algebraic hierarchy, incorporating ancestor classes' fields rather than taking superclasses as instance parameters. This choice is further explained in Section 12.

The various hierarchies in `mathlib` are interwoven through multiple inheritance. Thus the hierarchy of order structures such as partial orders, linear orders and lattices (extending the notation typeclasses `has_le` providing the $\leq$ operator and `has_lt` providing $<$), is combined with the algebraic hierarchy into a hierarchy of ordered algebraic structures from partially ordered commutative monoids up to linearly ordered fields.

The first line `set_option old_structure_cmd true` switches between two representations of ancestors for `structure` and `class` declarations: under the default, "new" structure behaviour, `monoid M` would contain two fields, of type `semigroup M` and `mul_one_class M`, each of which carries its own distinct `has_mul` field. Thus the `mul_assoc` field inherited from `semigroup` would refer to a multiplication operation other than the multiplication of `one_mul` inherited from `mul_one_class`; the resulting class with two binary operators would not actually specify monoids. Indeed, Lean will detect such ambiguities and produce an error if a "new" structure inherits conflicting field names.

The "old" structure behaviour avoids this issue by copying all fields from the ancestor structure into the child structure, skipping duplicates, so that `monoid` only has one `mul` field. Compare the following two desugarings of `extends`:

```
class monoid_new (M : Type*) :=
(to_semigroup : semigroup M)
(to_mul_one_class : mul_one_class M)

class monoid_old (M : Type*) :=
(mul : M → M → M)
(mul_assoc : ∀ a b c : M, a * b * c = a * (b * c))
(one : M)
(one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)
```

Lean 4 only implements the "new" structure command to optimize the projection to ancestor structures, adding support for diamond inheritance through automatically inheriting from the common ancestor and copying the remaining fields.

In the terminology of Coq's Hierarchy Builder [24], the typeclasses are specified in terms of *mixins*: the packages of operations and properties available for a given structure. Like Hierarchy Builder provides for mixins, projections from a subclass to its immediate superclasses are automatically generated as instances. There is no explicit concept in Lean corresponding to Hierarchy Builder's *factories* or *builders*. To manually construct a subclass instance given a superclass or project a subclass into a superclass, users can apply the notation `.. s`, which extends a constructor's argument list by copying the relevant fields out of a tuple `s`.

In general, mathlib's hierarchy is extended when the mathematics requires it, so there are many parts of the hierarchy that do not form a boolean algebra. Thus there is no `comm_mul_one_class` forming the direct subclass of `comm_semigroup` and `mul_one_class`, nor is there a `comm_mul_class` that provides the `mul_comm` field by itself. Adding an intermediate class to the hierarchy is a straightforward process of moving over the fields and modifying the `extends` clauses, as recently happened with the addition of `mul_one_class`.

The relative ease of modification means the hierarchy does not need to be designed up front for all potential usages. This stands in contrast to the situation for packed classes, where refactoring the hierarchy involves a deep understanding of the details involved or the usage of a tool such as Hierarchy Builder, to ensure consistency such as the uniqueness of a join for any two structures applied to the same type [25]. Careful design is still needed for instances to avoid certain cases of drastic slowdowns, as seen in Section 12.

5 `multiplicative`: multiple instances on a single type

In addition to the above multiplicative hierarchy, mathlib includes an isomorphic additive hierarchy differing only in notation: the definition of `add_monoid` renames `(*)` and `1` to `(+)` and `0`. A metaprogram `to_additive` creates an appropriately renamed duplicate additive version of declarations. The duplicate notation is required for the definition of the `semiring` class in mathlib, since bundled class inheritance cannot express the fact that the additive and multiplicative structures of a semiring both form a monoid. In comparison to mathlib's ad hoc solution, Isabelle's locales support different notations automatically, since the operations of a target context can be renamed in sublocale declarations or instantiations [26].

Because the `monoid` and `add_monoid` classes exist in distinct hierarchies, there is no straightforward way to relate an additive structure with a multiplicative structure, for example to state that a group in additive notation (such as `zmod n`) is isomorphic to a group in multiplicative notation (such as `units (zmod p)`). Given a term of type `add_monoid M` we can easily construct a term of type `monoid M`, but this construction should not be declared as an instance because this could cause conflicting `monoid` instances on `M`, for example when it is a semiring.

To deal with incompatible instances of the same class on the same carrier type, mathlib uses *type synonyms* to manipulate unfolding [22]. For example, `multiplicative α` is declared to be a synonym of the type α onto which we can safely copy the `monoid` instance:

```
def multiplicative (α : Type*) := α
instance multiplicative.monoid {α : Type*}
  [add_monoid α] : monoid α :=
{ mul := λ (x y : α), (x + y : α),
  ..sorry } -- details omitted
```

Note that the definitional equality between `multiplicative α` and α allows us to view elements of one synonym as the other. Other type synonyms used in mathlib include `mul_opposite`, the multiplicative opposite where the left and right arguments to multiplications are swapped, `lex`, used to equip a product of partial orders with the lexicographic order (instead of pointwise comparison), and `order_dual`, the dual order where the left and right arguments to $<$ and $\leq$ are swapped.

The order library in MathComp instead distinguishes different structures by marking operators, specifically by adding a *display* parameter to the $\leq$ operator that specifies the expected order structure [27]. Using this technique, mathlib could redefine the $*$ operator as notation for `binop mul` and + notation for `binop add`, where `mul` and `add` are different constants of some unspecified type. Notation-agnostic declarations in Lean would have to carry around the display parameters explicitly, since leaving them implicit would block instance synthesis. It would be interesting to test whether Lean 4's automatic implicit parameters solve this drawback in all cases.

What allows us to distinguish `multiplicative α` from α during instance synthesis, at the same time as identifying the two types when defining the instance, is a subtlety in the unification mechanism. During unification, Lean will unfold definitions up to a specific *transparency* level, and this transparency level is restricted when unifying a candidate instance with the synthesis goal: only declarations explicitly marked as `reducible` and instances will be unfolded at that level. Thus `multiplicative α` can be unfolded to α when checking that `x + y : α` has type `multiplicative α`, but it will not be unfolded to α when checking that an instance of type `monoid α` solves the goal `monoid (multiplicative α)`. Unification triggered by instance search can always unfold instance declarations, which is required to support diamond inheritance.

Using the type `mul_equiv` of (multiplicative) group isomorphisms, `multiplicative` allows us to state the multiplicative subgroup of the field $\mathbb{Z}/p\mathbb{Z}$ is cyclic when p is prime, by stating it is isomorphic to $\mathbb{Z}/(p-1)\mathbb{Z}$:

```
def zmod.units_equiv_zmod (p : ℕ) (hp : nat.prime p) :
  mul_equiv (units (zmod p)) (multiplicative (zmod (p - 1))) :=
sorry -- proof omitted
```

The `to_additive` metaprogram and `multiplicative` type synonym work for complementary purposes: `to_additive` translates results on arbitrary monoids from one notation to another, while `multiplicative` translates a specific group from one notation to another. In particular, without the `to_additive` metaprogram, rewriting an expression involving both addition and multiplication would require casting back and forth between α and `multiplicative α`.

Declaring type synonyms to equal the original type has a disadvantage since it requires the user to know where to insert type ascriptions, such as `x + y : α`, based on a good understanding of the elaboration order. For example, defining multiplication on `multiplicative α` as `λ (x y : α), (x : α) + (y : α)` fails since Lean will instead search for a `has_add (multiplicative α)` instance. An alternative approach used in mathlib is to define a one-field structure wrapping the original type:

```
structure multiplicative (α : Type*) :=
of_add :: (to_add : α)
```

The implicit casts `α → multiplicative α` and `multiplicative α → α` are replaced by the explicit constructor `of_add` and projection `to_add` respectively. In exchange for the verbosity of having to insert these explicit operators everywhere, the type system will indicate missing conversions between additive and multiplicative notation.

6 `module`: multi-parameter classes

In algebra, a (*left semi-*)*module* is an additive commutative monoid `M` that is acted on by a semiring `R` through scalar multiplication, satisfying certain axioms; the concept generalizes vector spaces by replacing the field of scalars by an arbitrary semiring. Modules are available in the mathlib algebraic hierarchy in full generality as a multi-parameter typeclass depending on both `R` and `M`. Following the pattern of monoids, the base class introduces notation, and is subclassed to add the axioms of the structure:

```
class has_scalar (α β : Type*) : Type* :=
(smul : α → β → β)
infix · := has_scalar.smul

class mul_action (M A : Type*) [monoid M] extends has_scalar M A :=
(one_smul : ∀ (x : A), (1 : M) · x = x)
(mul_smul : ∀ (r s : M) (x : A), (r * s) · x = r · (s · x))

class distrib_mul_action (M A : Type*) [monoid M] [add_monoid A]
  extends mul_action M A :=
(smul_add : ∀ (r : M) (x y : A), r · (x + y) = r · x + r · y)
(smul_zero : ∀ (r : M), r · (0 : A) = 0)

class module (R M : Type*) [semiring R] [add_comm_monoid M]
  extends distrib_mul_action R M :=
(add_smul : ∀ (r s : R) (x : M), (r + s) · x = r · x + s · x)
(zero_smul : ∀ (x : M), (0 : R) · x = 0)
```

Compare this to the class-based analysis library in Isabelle/HOL, where the absence of multi-parameter classes means only real numbers appear as scalars [16]; Isabelle instead provides multi-parameter locales [3].

Vector spaces do not have a separate definition in mathlib since they only replace the ring axioms on the scalars with field axioms, while the fields of the module class are unchanged. Instead, a K-vector space V is denoted through parameters [field K] [add_comm_group V] [module K V]. In order to make vector spaces more discoverable for users, the mathlib community has been discussing a system of parameter-level abbreviations, so that [vector_space K V] expands into [field K] [add_comm_group V] [module K V].

6.1 Dangerous instances

We see here that the module hierarchy uses a mix of bundled and unbundled inheritance, unlike comm_monoid which solely uses bundled inheritance. This follows the rule of bundling only if the superclass has a superset of the subclass's parameters; otherwise the generated instance would be a *dangerous instance* where some parameters are undetermined. Namely, declaring that module R M extends add_comm_monoid M would generate the following instance:

```
instance module.to_add_comm_monoid {R M : Type*} [module R M] :
  add_comm_monoid M := sorry
```

Now instance synthesis for add_comm_monoid M will lead to a search for module ?M_1 M, where ?M_1 is a free metavariable. Since unification in the elaborator can call instance synthesis, without backtracking, finding the wrong instance for an underspecified goal may cause unification to fail where another instance would have worked. The general rule is that parameters to an instance should be inferrable through either unification with the goal type, or uniquely determined through instance synthesis. Section 9 illustrates that this constraint follows from instance declarations, not the class declaration.

Such dangerous instances with free variables in their constraints can be remedied in various ways. If, as above, the instance derives from a subclass constraint involving the extends keyword, the constraint is instead expressed through an instance parameter on the subclass; this implies no dangerous instance is generated to express inheritance. The main drawback is that this mix of unbundled and bundled inheritance is more confusing and less natural than the approach used in the MathComp library, where canonical structures allow bundling the additive monoid structure and the R-module structure [28].

If the free parameter is uniquely determined by the choice of the bound parameters, we can register this functional dependency with the out_param construction. Lean assigns all parameters as in-parameters, unless explicitly marked as out_param, in contrast to the automatic determination of the direction in Coq. For example, we can make R a functional dependency of M by instead defining:

```
class module (R : out_param Type*) (M : Type*) [semiring R] := -- etc.
```

The elaborator replaces all out parameters in the synthesis goal with a free metavariable, which is filled by unifying the goal with the type of the candidate instance. For module this functional dependency is not acceptable since each add_comm_monoid M has an instance add_comm_monoid.nat_module : module ℕ M reflecting the natural ℕ-module structure (see also Section 8), which would be incompatible with an

R-module structure for other semirings R. Moreover, the instance `add_comm_monoid.nat_module` provides a second reason that bundled inheritance is unsuitable for the subclass relation of `add_comm_monoid` and `module`: it would form a loop with the instance `module.to_add_comm_monoid`.

A final way to resolve dangerous instances is to remove the `instance` keyword so that it does not participate in synthesis. mathlib takes this approach when stating the theorem that any module over a ring has additive inverses:

```
def module.add_comm_monoid_to_add_comm_group (R M : Type*)
  [ring R] [add_comm_monoid M] [module R M] :
  add_comm_group M := sorry -- proof omitted
```

To provide `add_comm_group` instances when `R` is known, we can still make use of the instance `add_comm_monoid_to_add_comm_group` in a separate `instance` declaration.

7 `monoid_hom_class`: generic bundled morphisms

The representation of morphisms such as group homomorphisms or linear maps has changed repeatedly in mathlib, is still not unified and is still undergoing refactors. The main issue complicating the design is the trade-off between generality and ease of inference. The author of this paper has designed a pattern providing bundled morphisms with some of the advantages lost during the move from unbundled morphisms, by making theorems generic over types of morphisms. The same pattern works for subobjects, replacing “morphism” with “subobject” and “a map preserving an operation” with “a set closed under an operation”.

7.1 Unbundled morphisms

The original design of algebraic homomorphisms in mathlib did not bundle maps in the same structure as their properties, allowing any function `f : R → S` to be used as a ring homomorphism if an `is_monoid_hom f` instance was available. The `is_ring_hom` predicate stated `f` preserves the ring operations $*$, $+$, `1` and `0`. Instances were available for the common operations, except composition:

```
class is_monoid_hom {M N : Type*} [monoid M] [monoid N] (f : M → N) :
     Prop :=
(map_mul : ∀ x y : M, f (x * y) = f x * f y)
(map_one : f 1 = 1)

class is_ring_hom {R S : Type*} [semiring R] [semiring S] (f : R → S)
  extends is_monoid_hom f :=
(map_add : ∀ x y : R, f (x + y) = f x + f y)
(map_zero : f 0 = 0)

instance id.is_ring_hom (R : Type*) [semiring R] :
  is_ring_hom (id : R → R) := sorry -- details omitted
```

```
lemma comp.is_ring_hom {R S T : Type*} (f : R → S) (g : S → T)
  [semiring R] [semiring S] [semiring T] [is_ring_hom f] [is_ring_hom
    g] :
  is_ring_hom (g ∘ f) := sorry -- details omitted
```

Synthesis for the `is_ring_hom` class struggles with the resulting higher-order matching problems. In particular, there is no instance for composition since matching `is_ring_hom (?_g ∘ ?_f)` with a goal `is_ring_hom f` would result in setting `?_f` to `f` and `?_g` to the identity function `id`. Thus, making `comp.is_ring_hom` would lead to instance synthesis diverging along the path `is_ring_hom f → is_ring_hom (id ∘ f) → is_ring_hom (id ∘ id ∘ f) → ⋯`. In the formalization of Witt vectors, these issues led Commelin and Lewis to avoid classes and instead use a custom metaprogram for generating instances of their `is_poly` predicate [29].

Apart from the inability of instances on compositions to be synthesised, under this design rewriting tactics such as the simplifier cannot easily iterate over all subterms where the `map_mul` lemma can be applied: since every subterm of any term can potentially unify with a function application (such as a constant function), any subterm would cause an instance search. (In fact, Lean's simplifier rejects rewrite lemmas of this form altogether, as we will discuss in Section 13.) Finally, the collection of morphisms could not be as easily treated as an object in its own right, for example to put a group structure on the automorphisms of a field [6].

7.2 Bundled morphisms

For these reasons, mathlib was refactored to prefer bundled morphisms:

```
structure monoid_hom (M N : Type*) [monoid M] [monoid N] :=
(to_fun : M → N)
(map_mul : ∀ x y, to_fun (x * y) = to_fun x * to_fun y)
(map_one : to_fun 1 = 1)

structure ring_hom (R S : Type*) [semiring R] [semiring S]
  extends monoid_hom R S :=
(map_add : ∀ x y, to_fun (x + y) = to_fun x + to_fun y)
(map_zero : to_fun 0 = 0)

instance monoid_hom.has_coe_to_fun (M N : Type*) [monoid M] [monoid N]
    :
  has_coe_to_fun (monoid_hom M N) (λ _, M → N) :=
{ coe := monoid_hom.to_fun }

def monoid_hom.id (M : Type*) [monoid M] : monoid_hom M M :=
{ to_fun := id, .. } -- details omitted

def monoid_hom.comp {M N O : Type*} [monoid M] [monoid N] [monoid O]
  (f : monoid_hom M N) (g : monoid_hom N O) : monoid_hom M O :=
{ to_fun := g ∘ f, .. } -- details omitted
```

Lean uses the `has_coe_to_fun` instance to parse `(f : monoid_hom M N) x` as `(@coe_fn _ _ (monoid_hom.has_coe_to_fun M N) f : M → N) x`. Further examples of bundled morphisms available in mathlib include ring homomorphisms, linear maps, monotone functions (order homomorphisms) and the bijective versions of the above: group, ring and order isomorphisms and linear equivalences.

Bundled morphisms do not suffer from the composition, simplification and structure issues, at the cost of all morphisms needing to be declared as such ahead of time or needing lemmas to convert between bundled and unbundled forms. This is a drawback especially when the unbundled form has convenient notation, such as the additive group endomorphism of a ring given by multiplying by a constant `c`:

```
instance mul.is_add_monoid_hom {R : Type*} [ring R] (c : R) :
  is_add_monoid_hom ((*) c) := sorry -- details omitted

def add_monoid_hom.mul_left {R : Type*} [ring R] (c : R) :
  add_monoid_hom R R := { to_fun := (*) c, ..sorry } -- details
    omitted
```

In addition, it is no longer possible to use `monoid_hom` lemmas for a `ring_hom`: since `monoid_hom` and `ring_hom` are two different bundled types, ring homomorphisms can be viewed as monoid homomorphisms only through (manually) inserting coercions. Although the coercion could be supplied in some cases using unification hints, the support for unification hints in Lean 3 was not sufficient to do this in every case, and in anticipation of Lean 4's new unification system, unification hints were entirely removed from Lean 3 and mathlib.

Instead, to gain fully automatic simplification, all `monoid_hom` lemmas had to be copied over to `ring_hom` and all other structures extending `monoid_hom`. Thus mathlib ended up with many copies of lemmas such as `map_prod`:

```
lemma monoid_hom.map_prod (g : monoid_hom M N) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) := sorry -- proof omitted

lemma ring_hom.map_prod (g : ring_hom R S) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) :=
monoid_hom.map_prod s f g.to_monoid_hom

lemma mul_equiv.map_prod ...
lemma ring_equiv.map_prod ...
lemma alg_hom.map_prod ...
lemma alg_equiv.map_prod ...
```

This duplication is further multiplied by the amount of monoid operators in mathlib: a corresponding version of each `map_prod` lemma also exists for the product of a multiset and for the product of a list. Furthermore, monoid homomorphisms preserve multiplicative inverses, powers of elements, divisibility, nth roots, and so on. The end result is that the full set of lemmas grows proportionally to the number of structures extending `monoid_hom` times the number of operations preserved by a `monoid_hom`.

This copying happened manually and typically on an ad hoc basis, so that mathlib contributors often encountered lemmas that were missing for their specific choice of morphism, needing to switch contexts and add these mathematically trivial lemmas back in by hand, waiting for the dependencies to recompile before being able to continue with their proof. To address these shortcomings, the mathlib community on initiative of the author of this paper, switched to a third design pattern that automates the derivation of lemmas when a morphism type is extended.

7.3 Morphism classes

The cause of this duplication is that the pattern of "bundled morphism" was applied informally, with no unifying programmatic interface. The key insight was to follow object-oriented practice of programming to an interface rather than a concrete class, or in Lean terms: to program to a typeclass `monoid_hom_class` rather than a concrete type such as `monoid_hom`.

The first step in introducing this interface was a typeclass `fun_like` for *types* of bundled (dependent) functions, based on Eric Wieser's `set_like` class for types of bundled subobjects.[1]

```
class has_coe_to_fun (F : Type*) (α : out_param (F → Type*)) :=
(coe : Π x : F, α x)

class fun_like (F : Type*)
  (α : out_param Type*) (β : out_param (α → Type*))
  extends has_coe_to_fun F (λ _, Π a : α, β a) :=
(coe_injective' : function.injective coe)

-- A typical instance looks like:
instance monoid_hom.fun_like : fun_like (monoid_hom M N) M (λ _, N) :=
{ coe := monoid_hom.to_fun,
  coe_injective' := λ f g h, by { cases f, cases g, congr' } }
```

After defining the instance `monoid_hom.fun_like`, instance synthesis provides function application syntax, extensionality and congruence lemmas for monoid homomorphisms.

The next step in addressing the duplication is to introduce a class for the bundled morphism types that coerce to `monoid_hom`:

```
class monoid_hom_class (F : Type*) (M N : out_param Type*)
  [monoid M] [monoid N] extends fun_like F M (λ _, N) :=
(map_one : ∀ (f : F), f 1 = 1)
(map_mul : ∀ (f : F) (x y : M), f (x * y) = f x * f y)

instance : monoid_hom_class (monoid_hom M N) M N :=
sorry -- details omitted
```

[1] https://github.com/leanprover-community/mathlib/pull/6768

Note the difference between `is_monoid_hom f` and `monoid_hom_class F M N`: the former is a predicate on *morphisms*, the latter is a predicate on *types of morphisms*.

It is necessary to fully apply the morphism types before they can be used as a parameter to `monoid_hom_class`: since `monoid_hom` and `ring_hom` have different instance parameters, we are not able to write both `monoid_hom_class monoid_hom` and `monoid_hom_class ring_hom` type-correctly. This means the class requires parameters `M N`, which are `out_param`s so that the lemma application `map_one f` can leave these parameters implicit.

The types which extend `monoid_hom` should receive a `monoid_hom_class` instance, which we can do by subclassing `monoid_hom_class` and instantiating the subclass:

```
class ring_hom_class (F : Type*) (R S : out_param Type*)
  [semiring R] [semiring S]
  extends monoid_hom_class F R S :=
(map_zero : ∀ (f : F), f 0 = 0)
(map_add : ∀ (f : F) (x y : R), f (x + y) = f x + f y)

instance : ring_hom_class (ring_hom R S) R S := sorry -- details
    omitted
```

Now lemmas can be made generic by parametrizing over all the types of bundled morphisms, reducing the multiplicative amount of lemmas to an additive amount: each extension of `monoid_hom` should get a `monoid_hom_class` instance, and each operation preserved by `monoid_hom`s should get a lemma taking a `monoid_hom_class` parameter.

```
lemma map_prod {G : Type*} [monoid_hom_class G M N] (g : G) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) := sorry -- proof omitted
```

This design pattern has been applied in mathlib to morphisms (implemented as subclasses of `fun_like`) and subobjects (implemented as subclasses of `set_like`). The mathlib community has welcomed the morphism class design for reducing the amounts of duplication, manual work and missing lemmas, although not all usages have switched to the generic lemmas, and work is still ongoing to provide a suitable generic form of standard operations such as composition and identity maps.

8 `nsmul`: ensuring equality of instances

Each `add_comm_monoid M` structure naturally gives rise to an ℕ-module structure, where `n • x` is defined as `x + x + ⋯ + x`, `n` times. In addition, each `semiring R` structure naturally gives rise to an `R`-module structure on itself, where `x • y` is defined as `x * y`. These two actions are available in mathlib as instances `add_comm_monoid.nat_module` and `semiring.to_module` respectively. Note that setting `M = R = ℕ` results in two instances for `module ℕ ℕ`. The existence of multiple instances of the same type does not necessarily lead to problems in Lean. Indeed, diamond inheritance in the mathlib algebraic hierarchy exploits this possibility. Problems arise when the two instances are not definitionally equal, in cases such as a goal containing `add_comm_monoid.nat_module` in which we want to apply a lemma containing semiring.to_module. As an extra complication, the two instances result in the same syntax n • k, making incompatibilities hard to spot.

To resolve such issues, first we could ensure only one instance is found, for example by replacing the other instance with a def that is not considered during instance synthesis. However, both described above are mathematically useful in their respective context, and only cause an issue when this context overlaps, namely for the natural numbers. Modifying the order in which instances are considered will not work, since one instance is not merely a generalization of the other: when combining a lemma on add_comm_monoids with a lemma on semirings, both instances will still appear no matter the instance priorities.

When overlapping instances are required, the mathlib community ensures these are definitionally equal for all possible instantiations in the overlap. Note that Lean's implementation of diamond inheritance automatically provides definitional equality of all inheritance paths.

An advantage of the Prop-valued mixin classes discussed in Section 10 is that all instances are equal by proof irrelevance. For example, the mathlib community is considering replacing the data-carrying class fintype (α : Type*) : Type* containing a finite enumeration of the elements of a given type, with a proof-only class finite (α : Type*) : Prop non-constructively asserting the existence of an enumeration. Although fintype α is designed to be a subsingleton for all α, it is only a subsingleton up to propositional equality, meaning two different enumerations would still lead to unification issues. On the other hand, Prop-valued classes cannot be applied everywhere: the absence of data means it is incompatible with classes that provide notation such as scalar multiplication, and it is in general incompatible with intuitionistic logic. The class decidable_pred {α : Type*} (p : α → Prop) : Type* provides a decision algorithm for p, and is used in mathlib for small numeric computations. While we could define this to be Prop-valued by setting decidable_pred p := ∀ x, p x ∨ ¬ (p x), that would make it useless for actually performing this decision algorithm.

To make the two module ℕ ℕ instances definitionally equal, we ensure data-carrying fields of these instances are definitionally equal, using proof irrelevance for the proof-carrying fields [30]. In particular, the smul field of add_comm_monoid.module needs to be defined so that instantiated for ℕ, it equals multiplication on natural numbers nat.mul. While we could redefine nat.mul to be recursive on the left argument to match the action of left modules, this would violate the requirements of right modules, where multiplication by natural numbers must be right-recursive.

Instead, mathlib adds extra data to add_comm_monoid's ancestor add_monoid: a field nsmul : ℕ → M → M defines scalar multiplication by a natural number, and two proof fields assert it (propositionally) equals the left-recursive definition:

```
class add_monoid (M : Type*) extends add_semigroup M, add_zero_class M
     :=
(nsmul : ℕ → M → M)
(nsmul_zero : ∀ x, nsmul 0 x = 0)
(nsmul_succ : ∀ (n : ℕ) x, nsmul (n + 1) x = x + nsmul n x)
```

The `nsmul` field can be set to the usual $*$ operator for `add_comm_monoid ℕ`, and `mathlib` provides a generic implementation `nsmul_rec M : Type* [has_zero M] [has_add M] : ℕ → M → M` for instances where definitional equality is not a concern.

The same principle of providing a field for all definitional equalities generalizes the principle of *forgetful inheritance* [31] known also in Coq and Isabelle, that the instance creating a superclass from a subclass can only consist of projecting away fields. This rule is illustrated in `mathlib` by the class `metric_space` which extends `topological_space` [6, 32].

There is currently no mechanism available in `mathlib` for automatically detecting or resolving issues with definitional equality of instances. A linter [33] that warns for diamond issues would already be a useful improvement over the status quo of manual investigation. Even better would be a mechanism that can canonicalize instances of propositionally subsingleton classes to ensure equality also holds definitionally.

9 Instance parameters depending on out parameters

The `fun_like` pattern for morphisms does not just allow for "forgetful" inheritance from supertypes to subtypes, it also provides a small amount of automation for proving properties of morphisms, by allowing the user to define further instances. For example, a bijection that preserves multiplication also preserves an identity element in a monoid, which we express through the following instance:

```
class mul_equiv_class (F : Type*) (M N : out_param Type*)
  [has_mul M] [has_mul N] extends fun_like F M N :=
(to_fun : F → M → N)
(bijective : ∀ (e : F), function.bijective (to_fun e))
(map_mul' : ∀ (e : F) (x y : M), e (x * y) = e x * e y)

instance mul_equiv_class.monoid_hom_class (F : Type*) {M N : Type*}
  [monoid M] [monoid N] [mul_equiv_class F M N] :
  monoid_hom_class F M N :=
sorry -- proof omitted
```

Note that this instance has parameters `[monoid M] [monoid N]` depending on out parameters `M N` to `mul_equiv_class F M N`. This means the order of unification and instance synthesis are heavily interdependent: suppose Lean infers the type of the application `map_one e` where `e : mul_equiv M N`. This results in the synthesis goal with four metavariables, indicated by underscores: `@monoid_hom_class (mul_equiv M N) _ _ _ _`, since out parameters and their dependencies are replaced with metavariables at the start of synthesis.

Applying the candidate instance `mul_equiv_class.monoid_hom_class` unifies the goal and the return type of the instance, resulting in the creation of a new synthesis goal:

```
  @mul_equiv_class (mul_equiv M N) _ _
    (monoid.to_has_mul _) (monoid.to_has_mul _)
```

We next apply the candidate instance `mul_equiv.mul_equiv_class`. Unification assigns `M` and `N`, leaving unification goals `monoid.to_has_mul _ =?= M.has_mul` and `monoid.to_has_mul _ =?= N.has_mul`, and synthesis goals of type `monoid M` and `monoid N`. The only way to make progress in the unification goals is to first synthesize the `monoid` instances, then unfolding the instances and checking the multiplications coincide.

Lean can successfully infer the type of `map_one e`, by carefully postponing goals in the elaborator. This is not always possible in the same way: in particular when there is diamond inheritance in instance parameters dependent on out parameters. Concretely, the inheritance graph should not contain the following shape:

```
set_option old_structure_cmd true

class root (β : Type) : Type :=
(value : β)
class left (β : Type) extends root β
class right (β : Type) extends root β
class leaf (β : Type) extends left β, right β

class bottom (α : Type) (β : out_param Type) [root β]
class middle (α : Type) (β : out_param Type) [root β]
class top (α : Type) (β : out_param Type) [root β]

instance top.to_middle {α β} [left β] [top α β] :
  middle α β := ⟨⟩
instance middle.to_bottom {α β} [right β] [middle α β] :
  bottom α β := ⟨⟩

example {α β} [h1 : leaf β] [h2 : top α β] : bottom α β :=
by apply_instance -- Fails
```

In the example, applying `middle.to_bottom` results in the synthesis goal `middle α _`, where unification supplies an instance argument `right.to_root _`. The return type of `top.to_middle` instead contains `left.to_root`. This results in a synthesis goal `top α _` and a unification goal `right.to_root _ =?= left.to_root _`. The unification goal can be solved once the synthesis goal is solved, by synthesizing a `top α β` instance, since `h1.to_right.to_root` unifies with `h1.to_left.to_root`. However, a candidate instance must fully unify with the goal before its synthesis goals are processed, so here unification fails and the instance is not found.

Although the above example has been minified, the issue it illustrates arises in a more complicated form `mathlib` due to the extensive use of out parameters. In particular, this occurs in the declaration of `ring_seminorm_class` as subclass of `add_group_seminorm_class` under the assumption of an `ordered_semiring`, which is itself a subclass of `nonneg_hom_class` under the assumption of `linear_ordered_add_comm_monoid`; the assumptions have a common descendant `linear_ordered_semiring` which cannot be reliably synthesised due to this limitation. The workaround in this case was to declare `ring_seminorm_class` directly as subclass of `nonneg_hom_class`, so the problematic inheritance path can be avoided.

Alternatively, mathlib could adopt the rule that class declarations may not contain instance parameters that depend on out parameters. This would mean replacing the bundled inheritance of `monoid_hom_class` on `fun_like` with unbundled inheritance. A separate `fun_like` parameter would be synthesised first and supply the values of `M` and `N`, so that `monoid M` and `monoid N` instances can be synthesised before synthesis of `monoid_hom_class` starts, and unification remains unblocked throughout.

10 `unique`: proof-carrying mixin

The mathlib algebraic hierarchy is *semi-bundled*, meaning all operations and properties are passed in a single instance parameter. In contrast, mathlib also provides a large collection of mixins that can be added as separate instance parameters. For example, `subsingleton : Π (α : Type*), Prop` asserts the type α has at most one element. The subclass `unique α` of `subsingleton α` (constructively) asserts that α has exactly one element. This means `unique α` is also a subclass of `inhabited α`, which (constructively) specifies an element of α while also allowing for more. A theorem about trivial monoids will take these assumptions as separate parameters `[monoid M] [subsingleton M]`:

```
instance [monoid M] [subsingleton M] : unique (units M) :=
sorry -- proof omitted
```

In fact, `unique α` is equivalent to the conjunction of `subsingleton α` and `inhabited α`. However, the implication `∀ {α}, subsingleton α → inhabited α → unique α` cannot be added while keeping `subsingleton` and `inhabited` superclasses of `unique`, since that would result in an infinite loop `unique → subsingleton → unique → subsingleton →` $\cdots$ during instance synthesis. The tabled instance synthesis procedure in Lean 4 will ensure searches are performed only once per syntactically equal subgoal, resolving this specific issue [19]. The current version of mathlib still uses such conjunction classes even though instances cannot be automatically synthesized from conjuncts. Preferring a single instance parameter improves performance by reducing term size, as we will discuss in Section 12.

11 `fact`: interfacing between instances and non-instances

Suppose we want to create an instance reflecting the fact that $\mathbb{Z}/n\mathbb{Z}$ is a field if n is a prime number. This instance will take a number `n : ℕ` and a proof showing `n` is prime, and return a `field (zmod n)` instance. Given `n`, a proof that `n` is prime cannot be inferred through unification, so to make the instance synthesizable, the proof of primality must appear as an instance parameter. Thus, we could define a class `nat.prime n` asserting `n : ℕ` is a prime number, and take an instance of this class as a parameter of the `zmod.field` instance:

```
class nat.prime (n : ℕ) : Prop :=
(nontrivial : 2 ≤ n) (only_two_divisors : ∀ m ∣ n, m = 1 ∨ m = n)

def zmod : ℕ → Type
| 0     := ℤ
| (n+1) := fin (n+1)

instance zmod.field (n : ℕ) [nat.prime n] : field (zmod n) :=
sorry -- details omitted
```

Unfortunately, instances for `nat.prime` do not work well in their own right: it is impractical to check that a term `n` is a prime number by recursion on the term structure of `n`. In particular, `n` may contain free variables or be too large to reduce to a unary numeral. Splitting between a predicate `def nat.prime` whose proofs are passed as explicit parameters and the same predicate as a class declaration `class nat.prime_class` whose proofs are passed as instance parameters is not satisfying either, due to the large amount of duplication this would entail.

Instead mathlib provides a mechanism for ad hoc typeclass creation, by supplying a proposition to the `fact` class:

```
def nat.prime (n : ℕ) : Prop := 2 ≤ n ∧ (∀ m ∣ n, m = 1 ∨ m = n)

class fact (p : Prop) : Prop := (out : p)

instance zmod.field (n : ℕ) [fact (nat.prime n)] : field (zmod n) :=
sorry -- details omitted
```

In a similar way, the `fact` class is used for the assumption `x < y` when showing that the interval $[x, y] \subset \mathbb{R}$ is a manifold with boundary, to provide the assumption that a polynomial f splits in a field K when defining the natural inclusion of the splitting field of f into K, and to provide non-negativity or positivity assumptions in various contexts.

Along with the ad hoc class pattern provided by `fact`, there is an ad hoc instance pattern provided by the tactic `letI`. Instance synthesis considers declarations marked as `instance` and parameters to the current declaration, caching these before elaborating the type and body of the declaration. The `letI` tactic inserts new instances into this cache, providing this instance in the current proof context.

In addition, `letI` can resolve the dangerous instance issue of Section 6.1 in some cases: in a proof context where the ring of scalars `R` remains fixed, we can use `letI` to safely make `module.add_comm_monoid_to_add_comm_group` an instance within this context.

12 Performance and bundling

The pervasive use of typeclasses in mathlib means instance synthesis accounts for 10 to 25 percent of the build time of a typical mathlib file. Beyond the time taken for synthesis, the use of typeclasses has performance impacts on the entirety of the compilation process. For example, typeclasses tend to produce larger terms compared to those generated by canonical structure: Another factor complicating direct comparison with other mechanisms is the trade-off between upfront and repeating costs. For instance, although activating a locale in Isabelle is rather costly since it requires processing all declarations in the locale's dependency graph [3], declarations can be structured such that this cost only needs to be paid once for a group of declarations; instance synthesis is performed for each instance parameter in each term. Performance of typeclasses in Lean remains acceptable, though, thanks to instance caching, Lean's efficient synthesis implementation in C++ and mathlib's design patterns.

First of all, the mathlib algebraic hierarchy avoids unbundled subclasses that express superclass constraints through instance parameters, since these lead to exponential blowup of term sizes. An example of exponential blowup is discussed in detail in Ralf Jung's blog post [34]. This example concerns product type instances of an unbundled class such as the following modification to the `comm_monoid` class:

```
class semigroup (G : Type*) [has_mul G] := ...
class mul_one_class (M : Type*) [has_one M] [has_mul M] := ...
class comm_semigroup (G : Type*) [semigroup G] := ...
class monoid (M : Type*) [semigroup M] [mul_one_class M].
class comm_monoid (M : Type*) [monoid M] [comm_semigroup M].
```

Providing an instance for the natural numbers is straightforward, although it now involves instantiating each step in the hierarchy separately:

```
instance : semigroup ℕ := sorry -- details omitted
instance : mul_one_class ℕ := sorry -- details omitted
instance : comm_semigroup ℕ := sorry -- details omitted
instance : monoid ℕ := sorry -- details omitted
instance : comm_monoid ℕ := sorry -- details omitted
```

When we want to instantiate the commutative monoid structure on the product of two commutative monoids, we see that the length of types starts to grow noticeably:

```
instance prod.has_mul [has_mul G] [has_mul H] : has_mul (G × H) :=
{ mul := λ a b, (a.1 * b.1, a.2 * b.2) }
instance prod.semigroup [has_mul G] [has_mul H]
  [semigroup G] [semigroup H] : semigroup (G × H) :=
sorry -- details omitted
...
instance prod.comm_monoid
  [has_one M] [has_one N] [has_mul M] [has_mul N]
  [semigroup M] [semigroup N] [mul_one_class M] [mul_one_class N]
  [monoid M] [monoid N] [comm_semigroup M] [comm_semigroup N]
  [comm_monoid M] [comm_monoid N] :
  comm_monoid (M × N) :=
sorry -- details omitted
```

The linear growth in the types translates to an exponential growth in the term size of concrete instances, since each instance parameter implicit in `comm_monoid (ℕ × ··· × ℕ)` is filled with a term that has itself the same number of instance arguments.

The performance issue of unbundled classes is well known in the Coq community since Coq has a similar implementation of classes to Lean, and the first Coq library using classes for its algebraic hierarchy suffered from slowdowns due to this design issue [17]. The *packed classes* design pattern used for performant canonical structures [1] translates to bundled subclassing: in packed classes, the substructure relation is expressed by declaring the superstructure as an instance, instead of a parameter to the record. Similarly, mathlib prefers bundled classes, expressing the subclass relation through incorporating the superclass as an instance, instead of a parameter on the class's type.

In addition, the deprecation of the `old_structure_cmd` option results in improved performance for unification of instances in the presence of large inheritance chains. Since equality of structures is determined field-wise, incorporating a parent as a field means instances deriving from the same parent instances can be immediately verified to be equal, compared to the `old_structure_cmd` situation where this comparison has to be performed on the union of all fields of all ancestor structures, unfolding all intermediate projections.

An important source of slowdowns is failing instance searches since the entire search space has to be exhaustively checked before failing. If a user omits a hypothesis by mistake, the error message should not be a timeout but instead point out that the instance was not found. Thus, all instances in mathlib are checked by a `fails_quickly` linter, that checks that within an acceptable time (configurable in the linter) synthesis fails to synthesize a given instance when arguments are missing. The `fails_quickly` linter can also detect timeouts caused by looping or diverging synthesis, for example the loop `nonempty` → `has_bot` → `nonempty` in the following code:

```
-- ‘has_bot.bot‘ is notation for the minimum element of ‘α‘
class has_bot (α : Type*) := (bot : α)

instance has_bot_nonempty (α : Type*) [has_bot α] : nonempty α :=
⟨has_bot.bot⟩

-- The natural numbers are well-ordered.
instance nat.subtype.has_bot (s : set ℕ) [decidable_pred (∈ s)]
  [h : nonempty s] : has_bot s := sorry -- proof omitted
```

On the other hand, sometimes failing to synthesize instances should not cause an error, especially in tactics which can handle synthesis failures by switching to a less powerful procedure. The interaction between instances and tactics is further explored in Section 13.

The depth-first nature of instance synthesis means it is advantageous to try instances that succeed or fail fast before ones that require traversing a full tree before determining their success. The `priority` attribute of instances controls the order in which instances of the same class are considered: higher priorities are tried before lower priorities. The rule of thumb used in mathlib states to assign a low priority to *blanket instances*: those where all explicit parameters to the class are free variables. In particular, all subclass instances are automatically assigned a lower priority. A subtler case involved unification of quotient types: the instance con.quotient.decidable_eq states equality is decidable on the quotient of a type M by any decidable multiplicative congruence relation of type con M.

```
instance con.quotient.decidable_eq {M : Type*} [has_mul M] (c : con M)
  [∀ (a b : M), decidable (c a b)] : decidable_eq (quotient c) :=
sorry -- proof omitted
```

A value for the instance parameter [has_mul M] has to be synthesized when con. quotient.decidable_eq is considered as candidate instance, meaning this instance will cause a search through all instances of all subclasses of has_mul. Since mathlib makes extensive use of other quotient types such as multiset α, the quotient of list α modulo permutations, specialized instances such as multiset.decidable_eq : decidable_eq α $\to$ decidable_eq (multiset α) are assigned higher priority than the expensive instance con.quotient.decidable_eq.

While the above design guidelines have allowed the growth of mathlib's class hierarchy, the fact they often need to be verified and applied manually shows that performance is a key consideration in the further growth of the library.

13 Instances and tactics

The preceding sections have focussed on terms in the surface language, but Lean also provides a tactic framework for producing terms using built-in and user-defined procedures. The pervasive nature of instances in mathlib means that tactics are developed with the instance framework in mind, to guarantee correctness and performance. Conversely, the demands of comprehensively and performantly implementing tactic procedures have influenced typeclass design decisions.

The first, surface-level, way in which tactics and typeclasses interact in Lean, is that tactics are represented by elements in the tactic monad, and monadic syntax, including do notation, is implemented in Lean using the monad typeclass.

Lean also offers a built-in tactic tactic.mk_instance : expr $\to$ tactic expr , that takes a goal type and performs synthesis, returning the resulting instance (or monadically reports an error if synthesis fails). Tactics can also call the default elaboration procedures, that take expressions with metavariables and attempt to supply suitable values to the metavariables, thus indirectly interfacing with instance synthesis. The converse is not true: the instance synthesis algorithm is not able to invoke tactics to supply missing parameters. Making this possible would allow for a much more powerful variety of instances, such as replacing the [fact (nat.field n)] parameter to the zmod.field instance with a call to the norm_num tactic that can prove primality, replacing manual local instance declarations involving letI.

An important issue in the interaction of tactics and instances is that tactics do not have to be pure: they are allowed to alter various parts of the prover's state, including adding new (class and instance) declarations, or even replace the environment of declarations and attributes altogether. As a consequence, cache-based optimization strategies employed by the elaborator are not reliable within tactics, and it seems unlikely tactics can be called safely in the depths of the synthesis algorithm.

Tactics can produce very large terms, and many of the functions in these terms can take instance parameters, so instance synthesis ends up being a notable part of a tactic's runtime. A tactic typically performs its reasoning involving only a few kinds of mathematical structure, instantiated on a single type: the tactic `ring` for solving equalities in the language of commutative (semi)rings only needs an instance of `comm_ring R` and a few of its subclasses, but this instance appears hundreds of times in the resulting proof term. To optimize this common pattern, mathlib provides a `instance_cache` structure that allows tactic programmers to explicitly store common instances. Given a type (such as `R`), an element of the type `instance_cache` stores a map sending the name of a class to an instance of this class on `R`. When an `instance_cache` is used to construct a term, all instance parameters are supplied through a lookup in the cache, or synthesized and stored if missing. Of course, this construction is only needed because Lean 3's built in instance cache does not reliably work for tactics. Lean 4 promises a better caching mechanism that avoids the need to reimplement a cache.

Especially in the context of tactics, failing instance synthesis needs to behave well, since a tactic can use synthesis to detect if a more effective specialized procedure can be used, or that there is not enough structure an a more general procedure is required. In particular, the simplification procedure used instance synthesis to determine whether the type of a subterm had a `subsingleton` instance, which would allow more powerful rewriting in the presence of dependencies between subterms. This meant the simplification tactic searched for `subsingleton` instances for each subterm that another subterm depended on, in the worst case spending more than half its time on failing instance synthesis calls. Modifying the simplification procedure to instead rely on user-supplied congruence lemmas[2] resulted in a speedup of approximately 15 percent over the entirety of mathlib.

Many tactics in mathlib use a form of the *keyed matching* [35, 36] pattern: to find a function application of the form `f x y ... z`, the head term `f` should have literally the same syntax while the arguments `x y ... z` should merely unify with the expected terms. By unifying expected instance arguments with the supplied instance, keyed matching allows for the presence of overlapping, yet definitionally equal, instances. In particular, it is common in Lean 3 to use data structures indexed on the declaration name appearing at the head of a term. For example, the simplifier traverses all subterms of the goal, maps the head of the subterm to a list of all known rewrite rules for a term with that specific head, and uses unification to identify which rewrite rule, if any, applies. Thus, all rewrite rules supplied to the simplifier must have a name as head term, instead of a free variable.

Because this form of keyed matching is used, the algebraic hierarchy must involve notational typeclasses. Suppose that we unbundle the operation and identity element from the `monoid` class:

```
class monoid (M : Type*) (op : M → M → M) (id : M) :=
(id_op : ∀ x, op id x = x)
```

[2] https://github.com/leanprover-community/lean/pull/665

```
(op_id : ∀ x, op x id = x)
(op_assoc : ∀ x y z, op (op x y) z = op x (op y z))
```

In this design, the simplifier would be unable to apply `op_id` as a rewrite rule, since the head of the left hand side, `op x id` does not start with the name of a declaration. On the other hand, the simplifier does use `mul_one : ∀ x, x * 1 = x` from the actual algebraic hierarchy, since * (or more precisely `has_mul.mul`) is a declaration name.

14 Conclusion

The pervasive use of class-based patterns throughout mathlib demonstrates that the instance parameter mechanism scales to a large interconnected library of mathematical structures. In addition to algebraic, order and topological hierarchies, classes have proved useful for representing morphisms and subobjects. Still, the choice between bundled and unbundled subclassing and the duplication in the additive and multiplicative hierarchy are drawbacks of the use of typeclasses that do not appear in canonical structures or locales. For newcomers, notation for classes such as vector spaces is surprising. Even worse, dangerous instances, definitional equality and divergence are regular sources of errors that require a good understanding of the synthesis mechanism to resolve, and keeping the whole system performant is a permanent source of concern.

Lean 4 brings tabled instance synthesis to improve performance, and linters are able to report both dangerous instances and divergence. Future work should build on these improvements by providing a user friendly way of resolving definitional equality issues, be it a linter or a way to incorporate equality into the synthesis mechanism. In addition, macros that transform binder lists can address some of the unfamiliar notations. Another avenue to address the drawbacks of typeclasses is the integration of classes with another hierarchy mechanism, such as the combination of classes and locales in Isabelle.

Acknowledgements. I want to thank the anonymous reviewers and Jeremy Avigad, Jasmin Blanchette, Bryan Gin-Ge Chen, Johan Commelin, Sander Dahmen, Manuel Eberl, Rob Lewis, Assia Mahboubi, Filippo A. E. Nuccio, Kazuhiko Sakaguchi, Enrico Tassi and Eric Wieser for their helpful comments on earlier versions on this manuscript.

Declarations.

Funding

The author was funded by Nederlandse Organisatie voor Wetenschappelijk Onderzoek under Vidi grant No. 016.Vidi.189.037, Lean Forward.

Conflicts of interest

The author has no competing interests to declare that are relevant to the content of this article.

Code availability

Unabridged, interactive versions of the listings in this paper are available online at https://github.com/lean-forward/typeclass-paper.

References

[1] Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_23

[2] Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.) TPHOLs'99. LNCS, vol. 1690, pp. 149–166. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48256-3_11 . https://doi.org/10.1007/3-540-48256-3_11

[3] Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014) https://doi.org/10.1007/s10817-013-9284-7

[4] Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: Hints in unification. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 84–98. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_8

[5] Grabowski, A., Kornilowicz, A., Schwarzweller, C.: On algebraic hierarchies in mathematical repository of Mizar. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) FedCSIS 2016. Annals of Computer Science and Information Systems, vol. 8, pp. 363–371. IEEE, New York City, USA (2016). https://doi.org/10.15439/2016F520 . https://doi.org/10.15439/2016F520

[6] The mathlib Community: The Lean mathematical library. In: Blanchette, J., Hriţcu, C. (eds.) CPP 2020. Certified Programs and Proofs, pp. 367–381. ACM, New York City, USA (2020). https://doi.org/10.1145/3372885.3373824

[7] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Principles of Programming Languages. POPL '89, pp. 60–76. ACM, New York City, USA (1989). https://doi.org/10.1145/75277.75283

[8] Moura, L., Kong, S., Avigad, J., Doorn, F., Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. LNCS, vol. 9195, pp. 378–388. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-319-21401-6_26

[9] Moura, L., Avigad, J., Kong, S., Roux, C.: Elaboration in dependent type theory. CoRR **abs/1505.04324** (2015) 1505.04324

[10] Brady, E.C.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013) https://doi.org/10.1017/S095679681300018X

[11] team, T.: The Rust Reference 1.57.0. Accessed 2021-12-22. (2021). https://doc.rust-lang.org/1.57.0/reference/index.html

[12] Oliveira, B.C.d.S., Moors, A., Odersky, M.: Type classes as objects and implicits. SIGPLAN Not. **45**(10), 341–360 (2010) https://doi.org/10.1145/1932682.1869489

[13] Devriese, D., Piessens, F.: On the bright side of type classes: Instance arguments in Agda. SIGPLAN Not. **46**(9), 143–155 (2011) https://doi.org/10.1145/2034574.2034796

[14] Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_23 . https://doi.org/10.1007/978-3-540-71067-7_23

[15] Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs'97. LNCS, vol. 1275, pp. 307–322. Springer, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0028402 . https://doi.org/10.1007/BFb0028402

[16] Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 279–294. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_21 . https://doi.org/10.1007/978-3-642-39634-2_21

[17] Spitters, B., Weegen, E.: Developing the algebraic hierarchy with type classes in Coq. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 490–493. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_35 . https://doi.org/10.1007/978-3-642-14052-5_35

[18] Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE-28. LNCS, vol. 12699, pp. 625–635. Springer, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-79876-5_37 . https://doi.org/10.1007/978-3-030-79876-5_37

[19] Selsam, D., Ullrich, S., Moura, L.: Tabled typeclass resolution. CoRR **abs/2001.04301** (2020) 2001.04301

[20] Baanen, A.: Use and abuse of instance parameters in the Lean mathematical library. In: Andronick, J., Moura, L. (eds.) ITP 2022. LIPIcs, vol. 237, pp. 4–1420. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.4 . https://doi.org/10.4230/LIPIcs.ITP.2022.4

[21] Jones, M.P.: Type classes with functional dependencies. In: Smolka, G. (ed.) Programming Languages and Systems, pp. 230–244. Springer, Berlin, Heidelberg

(2000). https://doi.org/10.1007/3-540-46425-5_15

[22] Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. J. Funct. Program. **23**(4), 357–401 (2013) https://doi.org/10.1017/S0956796813000051

[23] Saïbi, A.: Typing algorithm in type theory with inheritance. In: Principles of Programming Languages. POPL '97, pp. 292–301. ACM, New York City, USA (1997). https://doi.org/10.1145/263699.263742 . https://doi.org/10.1145/263699.263742

[24] Cohen, C., Sakaguchi, K., Tassi, E.: Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In: Ariola, Z.M. (ed.) FSCD 2020. LIPIcs, vol. 167, pp. 34–13421. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl (2020). https://doi.org/10.4230/LIPIcs.FSCD.2020.34 . https://doi.org/10.4230/LIPIcs.FSCD.2020.34

[25] Sakaguchi, K.: Validating mathematical structures. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12167, pp. 138–157. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-51054-1_8 . https://doi.org/10.1007/978-3-030-51054-1_8

[26] Ballarin, C.: Exploring the structure of an algebra text with locales. J. Autom. Reason. **64**(6), 1093–1121 (2020) https://doi.org/10.1007/s10817-019-09537-9

[27] Allamigeon, X., Canu, Q., Cohen, C., Sakaguchi, K., Strub, P.-Y.: Design patterns of hierarchies for order structures. Submitted to ITP 2023. Version 1 (February 2023). (2023). https://hal.inria.fr/hal-04008820

[28] Mahboubi, A., Tassi, E.: The Mathematical Components Libraries. Zenodo, Genève, Switzerland (2017). https://doi.org/10.5281/zenodo.4457887

[29] Commelin, J., Lewis, R.Y.: Formalizing the ring of Witt vectors. In: Hriţcu, C., Popescu, A. (eds.) CPP '21. Certified Programs and Proofs, pp. 264–277. ACM, New York City, USA (2021). https://doi.org/10.1145/3437992.3439919 . https://doi.org/10.1145/3437992.3439919

[30] Wieser, E.: Scalar actions in Lean's mathlib. CoRR **abs/2108.10700** (2021) 2108.10700

[31] Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Sakaguchi, K.: Competing inheritance paths in dependent type theory: A case study in functional analysis. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12167, pp. 3–20. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-51054-1_1 . https://doi.org/10.1007/978-3-030-51054-1_1

[32] Buzzard, K., Commelin, J., Massot, P.: Formalising perfectoid spaces. In: CPP '20. Certified Programs and Proofs, pp. 299–312. ACM, New York City, USA

(2020). https://doi.org/10.1145/3372885.3373830

[33] Doorn, F., Ebner, G., Lewis, R.Y.: Maintaining a library of formal mathematics. In: Benzmüller, C., Miller, B.R. (eds.) CICM 2020. LNCS, vol. 12236, pp. 251–267. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53518-6_16 . https://doi.org/10.1007/978-3-030-53518-6_16

[34] Jung, R.: Exponential blowup when using unbundled typeclasses to model algebraic hierarchies. Accessed 2022-02-01 (2019). https://www.ralfj.de/blog/2019/05/15/typeclasses-exponential-blowup.html

[35] Gonthier, G., Tassi, E.: A language of patterns for subterm selection. In: Beringer, L., Felty, A. (eds.) ITP 2012, pp. 361–376. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_25

[36] Sakaguchi, K.: Reflexive tactics for algebra, revisited. In: Andronick, J., Moura, L. (eds.) ITP 2022. LIPIcs, vol. 237, pp. 29–12922. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.29 . https://drops.dagstuhl.de/opus/volltexte/2022/16738