Use and abuse of instance parameters in mathlib

Anne Baanen

Vrije Universiteit Amsterdam

ITP, 2022-08-10

What the title means

Lean is a theorem prover based on the calculus of constructions.
mathlib is the flagship Lean library.

What the title means

Lean is a theorem prover based on the calculus of constructions.
mathlib is the flagship Lean library.

Naming convention: typeclasses are a design pattern, implemented in Lean by the mechanism instance parameters.
Typeclasses in Coq work similarly.

Parameter kinds

```
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
```

(explicit parameters): supplied by user
{implicit parameters}: inferred through unification
[instance parameters]: inferred through synthesis

Instance synthesis algorithm

Lean finds instances through synthesis:
search through all declarations marked `@[instance]`,
until one unifies with the goal.

Instance synthesis algorithm

Lean finds instances through synthesis:
search through all declarations marked `@[instance]`,
until one unifies with the goal.

Instances can have instance parameters too.
These are also synthesized, resulting in depth-first search.
(Lean 4 brings a more efficient algorithm.)

Class declarations

Classes are record types declared with `class`:

```
class add_group (A : 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)
```

Dependent types mean classes can contain and depend on types, data and proofs in the same way.

Two inheritance patterns

Unbundled inheritance adds the superclass as instance parameter:

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

lemma neg_sub {A : Type}
  [add_group A] [add_comm_group A] (a b : A) :
  neg (sub a b) = sub b a := ...
```

Two inheritance patterns

Unbundled inheritance adds the superclass as instance parameter:

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

lemma neg_sub {A : Type}
  [add_group A] [add_comm_group A] (a b : A) :
  neg (sub a b) = sub b a := ...
```

Bundled inheritance provides superclass through instances:

```
instance add_comm_group.to_add_group (A : Type)
  [add_comm_group A] : add_group a := ...

lemma neg_sub {A : Type} [add_comm_group A]
  (a b : A) : neg (sub a b) = sub b a := ...
```

Mathlib's algebraic hierarchy

Mathlib uses bundled inheritance for the algebraic hierarchy:

```
class semigroup (G : Type) := ...

class comm_semigroup (G : Type)
  extends semigroup G := ...

class monoid (M : Type)
  extends semigroup M := ...

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

Mathlib's algebraic hierarchy

Mathlib uses bundled inheritance for the algebraic hierarchy:

```
class semigroup (G : Type) := ...

class comm_semigroup (G : Type)
  extends semigroup G := ...

class monoid (M : Type)
  extends semigroup M := ...

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

Multiple inheritance and overlapping instances are common.
Rule against definitionally unequal diamonds:
all solutions for a synthesis goal should unify.

Multi-parameter classes

Lean supports multi-parameter classes:

```
class module (R M : Type)
  [semiring R] [add_comm_monoid M] := ...
```

Vector spaces are expressed as
`[field K] [add_comm_group V] [module K V]`.

Multi-parameter classes

Lean supports multi-parameter classes:

```
class module (R M : Type)
  [semiring R] [add_comm_monoid M] := ...
```

Vector spaces are expressed as
`[field K] [add_comm_group V] [module K V]`.

Parameters to instances must be determined from the goal, so `module` requires unbundled inheritance: an instance `module R M → add_comm_monoid M` would leave `R` unspecified.
A linter in mathlib automatically warns for this situation.

Bundled morphisms

Mathlib uses bundled morphisms: structures containing a map and proofs showing it is a homomorphism.

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

structure ring_hom (R S : Type)
  [semiring R] [semiring S]
  extends monoid_hom R S := ...
```

Lean uses instances to coerce these tuples to functions.

Multiplicative explosion

Since `monoid_hom R S ≠ ring_hom R S`, proofs do not generalize automatically:

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

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
```

Multiplicative explosion

Since `monoid_hom R S ≠ ring_hom R S`, proofs do not generalize automatically:

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

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
```

There are many structures extending `monoid_hom` and many monoid operations in mathlib, resulting in multiplicatively many lemmas.

Morphism classes

My solution: generalize from monoid_hom M N to all types G with a monoid_hom_class G M N instance:

```
class monoid_hom_class (F M N : Type)
  [monoid M] [monoid N] :=
(to_fun : F → M → N)
(map_one : ∀ (f : F), to_fun f 1 = 1)
(map_mul : ∀ (f : F) (x y : M),
  to_fun f (x * y) = to_fun f x * to_fun f y)

class ring_hom_class (F R S : Type)
  [semiring R] [semiring S]
  extends monoid_hom_class R S := ...
```

Morphism classes

My solution: generalize from `monoid_hom M N` to all types `G` with a `monoid_hom_class G M N` instance:

```
class monoid_hom_class (F M N : Type)
  [monoid M] [monoid N] :=
(to_fun : F → M → N)
(map_one : ∀ (f : F), to_fun f 1 = 1)
(map_mul : ∀ (f : F) (x y : M),
  to_fun f (x * y) = to_fun f x * to_fun f y)

class ring_hom_class (F R S : Type)
  [semiring R] [semiring S]
  extends monoid_hom_class R S := ...
```

```
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)
```

Forgetful inheritance

There are two natural `module ℕ ℕ` instances:

- `add_comm_monoid M → module ℕ M`
  (`k • n = n + ⋯ + n`, `k` times)
- `semiring R → module R R`
  (`k • n = k * n`)

Diamond rule: scalar multiplications should be definitionally equal.

Forgetful inheritance

There are two natural `module ℕ ℕ` instances:

- `add_comm_monoid M → module ℕ M`
  (`k • n = n + ⋯ + n`, `k` times)
- `semiring R → module R R`
  (`k • n = k * n`)

Diamond rule: scalar multiplications should be definitionally equal.

Forgetful inheritance pattern: inheritance cannot create new data.
Instead, define scalar multiplication in the superclass:

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

Ad hoc classes and instances

If n is a prime number, $\mathbb{Z}/n\mathbb{Z}$ is a field.
Instance synthesis can't (practically) prove primality,
so a class `nat.prime` does not make sense.

Ad hoc classes and instances

If n is a prime number, $\mathbb{Z}/n\mathbb{Z}$ is a field.
Instance synthesis can't (practically) prove primality,
so a class `nat.prime` does not make sense.

Instead, Mathlib uses `fact (nat.prime n)`:

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

instance zmod.field (n : ℕ) [fact (nat.prime n)] :
  field (zmod n)
```

Ad hoc classes and instances

If n is a prime number, $\mathbb{Z}/n\mathbb{Z}$ is a field.
Instance synthesis can't (practically) prove primality,
so a class `nat.prime` does not make sense.

Instead, Mathlib uses `fact (nat.prime n)`:

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

instance zmod.field (n : ℕ) [fact (nat.prime n)] :
  field (zmod n)
```

Lean maintains a cache of candidate instances.
The `letI` tactic inserts into this cache, providing ad hoc instances within a proof context.

Term growth

Unbundled inheritance results in a parameter for each superclass, including in the instances themselves:

```
instance prod.comm_monoid
  [has_one M] [has_one N] [has_mul M] [has_mul N]
  [semigroup M] [semigroup N] [monoid M] [monoid N]
  [comm_semigroup M] [comm_semigroup N]
  [comm_monoid M] [comm_monoid N] :
  comm_monoid (M × N)
```

Linear growth of types causes exponential growth of synthesized instances.
Thus, deep hierarchies require bundling.

Looping in synthesis

The simple depth-first algorithm used by Lean 3 can easily end up looping:

```
class inhabited (t : Type) := (default : t)
class subsingleton (t : Type) :=
(eq : ∀ (x y : t), x = y)
class unique (t : Type)
  extends inhabited t, subsingleton t

instance (t : Type) [inhabited t] [subsingleton t] :
  unique t
```

Depth first search will end up diverging along the path
`unique` → `inhabited` → `unique` → …

Mathlib has a linter checking that synthesis succeeds or fails quickly.

Conclusions

Typeclasses scale to a large library...

Conclusions

Typeclasses scale to a large library… if you are able to fix common classes of subtle errors involving dangerous instances, definitional equality and divergence…

Conclusions

Typeclasses scale to a large library... if you are able to fix common classes of subtle errors involving dangerous instances, definitional equality and divergence... and can keep the whole system running quickly enough.