Bundling in Dependent Type Theory

Anne Baanen

Vrije Universiteit Amsterdam

EuroProofNet/CLAS, 2022-09-24

What the title means

Bundling is a feature of dependent type theory allowing us to combine data and proofs in one object. In the unbundled version, we'd instead pass these in separate parameters.

```
-- Bundled:
structure units (M : Type) [monoid M] :=
(val : M) (inv : M)
(left_inv : inv * val = 1)
(right_inv : val * inv = 1)

-- Unbundled:
def is_unit {M : Type} [monoid M] : M → Prop :=
λ x : M, ∃ y : M, x * y = 1 ∧ y * x = 1
```

Translating between the two

```
-- From bundled to unbundled:
def is_unit' (x : M) := ∃ u : units M, u.val = x

-- From unbundled to bundled:
def units' (M) := Σ x : M, is_unit x
```

Translating between the two

```
-- From bundled to unbundled:
def is_unit' (x : M) := ∃ u : units M, u.val = x

-- From unbundled to bundled:
def units' (M) := Σ x : M, is_unit x
```

Many of the refactors I've worked on in the Lean mathematical library involved bundling or unbundling certain objects.

Translating between the two

```
-- From bundled to unbundled:
def is_unit' (x : M) := ∃ u : units M, u.val = x

-- From unbundled to bundled:
def units' (M) := Σ x : M, is_unit x
```

Many of the refactors I've worked on in the Lean mathematical library involved bundling or unbundling certain objects.

Bundling interacts with many theorem proving features, especially, in Lean, simplification and typeclasses.

Typeclasses in Lean

Mathlib writes "let M be a monoid and $x \in M$" as `{M : Type} [monoid M] (x : M)`: operations and proofs are bundled into a typeclass, but the carrier type is unbundled.

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

Typeclasses in Lean

Mathlib writes "let M be a monoid and $x \in M$" as
`{M : Type} [monoid M] (x : M)`: operations and proofs are bundled into a typeclass, but the carrier type is unbundled.

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

Compare this to canonical structures (e.g. MathComp in Coq), where we'd write `{M : Monoid} (x : M.carrier)`.

Typeclasses in Lean

Mathlib writes "let M be a monoid and $x \in M$" as `{M : Type} [monoid M] (x : M)`: operations and proofs are bundled into a typeclass, but the carrier type is unbundled.

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

Compare this to canonical structures (e.g. MathComp in Coq), where we'd write `{M : Monoid} (x : M.carrier)`.

Isabelle/HOL has locales (e.g. HOL-Groups) which can further unbundle `monoid`: `{M : Type} [monoid M (*) 1] (x : M)`.
Or you pass in hypotheses separately from data.

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.)

Two inheritance patterns

Unbundled typeclass inheritance adds the superclass as a parameter:

```
class comm_monoid (M : Type) [monoid M] :=
(mul_comm : ∀ (x y : M), x * y = y * x)

lemma mul_left_comm {M : Type}
  [monoid M] [comm_monoid M] (x y z : M) :
  x * (y * z) = y * (x * z) := ...
```

Two inheritance patterns

Unbundled typeclass inheritance adds the superclass as a parameter:

```
class comm_monoid (M : Type) [monoid M] :=
(mul_comm : ∀ (x y : M), x * y = y * x)

lemma mul_left_comm {M : Type}
  [monoid M] [comm_monoid M] (x y z : M) :
  x * (y * z) = y * (x * z) := ...
```

Bundled typeclass inheritance provides superclass through instances:

```
instance comm_monoid.monoid (M : Type)
  [comm_monoid M] : monoid M := ...

lemma mul_left_comm {M : Type} [comm_monoid M]
  (x y z : M) :
  x * (y * z) = y * (x * z) := ...
```

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.

Combinatorial explosion

A drawback of bundling is the combinatiorial explosion of definitions: we have `semigroup`, `monoid`, `group`, `ring`, etc. and `comm_semigroup`, `comm_monoid`, `comm_group`, `comm_ring`, etc.

Combinatorial explosion

A drawback of bundling is the combinatiorial explosion of definitions: we have `semigroup`, `monoid`, `group`, `ring`, etc. and `comm_semigroup`, `comm_monoid`, `comm_group`, `comm_ring`, etc.
And `ordered_comm_monoid`, `ordered_comm_group`, `ordered_comm_ring`, etc.

Combinatorial explosion

A drawback of bundling is the combinatiorial explosion of definitions: we have `semigroup, monoid, group, ring`, etc. and `comm_semigroup, comm_monoid, comm_group, comm_ring`, etc.

And `ordered_comm_monoid`, `ordered_comm_group`, `ordered_comm_ring`, etc.

And `linear_ordered_comm_monoid`, `linear_ordered_comm_group`, `linear_ordered_comm_ring`, etc.

Combinatorial explosion

A drawback of bundling is the combinatiorial explosion of definitions: we have `semigroup, monoid, group, ring`, etc. and `comm_semigroup, comm_monoid, comm_group, comm_ring`, etc.

And `ordered_comm_monoid`, `ordered_comm_group`, `ordered_comm_ring`, etc.

And `linear_ordered_comm_monoid`, `linear_ordered_comm_group`, `linear_ordered_comm_ring`, etc.

Mathlib is now unbundling some of the ring and order properties: `covariant_class M M * ≤` and `covariant_class M M * <` replace strict versions of ordered monoids.

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.

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.

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)
```

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 : M → N)
(map_one : to_fun 1 = 1)
(map_mul : ∀ x y,
  to_fun (x * y) = to_fun x * to_fun y)
```

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 : M → N)
(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 `F` with a `monoid_hom_class F 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 `F` with a `monoid_hom_class F 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)
```

Simplification and bundled properties

The simplifier doesn't do much proof search, so bundled lemmas are more useful.

```
-- useful @[simp] lemma
lemma units.mul_inv [monoid M] (a : units M) :
  a.val * a.inv = 1

-- less useful @[simp] lemma
lemma is_unit.mul_inv [division_monoid M] (a : M) :
  is_unit a → a * (inv a) = 1
```

Unification and bundling

The old mathlib definition for a basis was unbundled:

```
def is_basis [module R M] (b : I → M) :=
linear_independent R b ∧ spanning R b
```

Unification and bundling

The old mathlib definition for a basis was unbundled:

```
def is_basis [module R M] (b : I → M) :=
linear_independent R b ∧ spanning R b
```

Using choice, we can construct a coordinate for `x : M` for this basis:

```
noncomputable def is_basis.repr [module R M]
  {b : I → M} (hb : is_basis R b)
  (x : M) (i : ι) : R :=
sorry -- implementation omitted
```

Unification and bundling

Lean has proof irrelevance, so all `hb hb' : is_basis R b` are equal.
So it will replace a nice readable `std_basis.is_basis` with anything else that typechecks.
So you have no idea which basis vectors it's referring to.

Unification and bundling

Lean has proof irrelevance, so all `hb hb' : is_basis R b` are equal.
So it will replace a nice readable `std_basis.is_basis` with anything else that typechecks.
So you have no idea which basis vectors it's referring to.

A bundled definition doesn't have that (admittedly small) disadvantage:

```
structure basis (I R M) [module R M] :=
(vec : I → M)
(li : linear_independent R b)
(span : spanning R b)
```

Implementation hiding

I am an intuitionist. So using choice to define `basis.repr` is unsatisfying.

Implementation hiding

I am an intuitionist. So using choice to define `basis.repr` is unsatisfying.

Bundling `basis` means we can add more data so choice is not needed:

```
def basis (I R M) [module R M] :=
(repr : M ≃ₗ[R] R ^ I)

noncomputable def basis.mk [module R M] (b : I → M) :
  linear_independent R b → spanning R b → basis I R M
```

Implementation hiding

I am an intuitionist. So using choice to define `basis.repr` is unsatisfying.

Bundling `basis` means we can add more data so choice is not needed:

```
def basis (I R M) [module R M] :=
(repr : M ≃ₗ[R] R ^ I)

noncomputable def basis.mk [module R M] (b : I → M) :
  linear_independent R b → spanning R b → basis I R M
```

Pushing the axiom of choice into certain constructors means we can omit it elsewhere.

Equality and bundling

Carrying properties across equalities is generally annoying:

```
example [division_monoid M] (a b : M)
  (ha : is_unit a) (hab : a = b) : inv a * b = 1 :=
by rewrite [hab, is_unit.inv_mul a ha] -- error:
-- `ha : is_unit a` but expecting `is_unit b`

example [monoid M] (a : units M) (b : M)
  (hab : a.val = b) : a.inv * b = 1 :=
by rewrite [hab, units.inv_mul a] -- error:
-- no occurrence of `a.val` in `a.inv * b`
```

Equality and bundling

For bundled structures, mathlib often (manually) defines `ext`ensionality rules and `copy` constructors:

```
example [monoid M] (a : units M) (b : M)
  (hab : a.val = b) : a.inv * b = 1 :=
by rewrite [ ←units.copy_eq a b,
  -- (copy hab).inv * b = 1
            ←units.ext hab,
  -- (copy hab).inv * (copy hab).val = 1
            units.inv_mul (units.copy hab)]
```

Equality and bundling

For bundled structures, mathlib often (manually) defines `ext`ensionality rules and `copy` constructors:

```
example [monoid M] (a : units M) (b : M)
  (hab : a.val = b) : a.inv * b = 1 :=
by rewrite [ ←units.copy_eq a b,
  -- (copy hab).inv * b = 1
           ←units.ext hab,
  -- (copy hab).inv * (copy hab).val = 1
           units.inv_mul (units.copy hab)]
```

Equalities between bundled and unbundled definitions are still annoying.

Equality and bundling

Bundling causes a lot of nontrivial synonyms:

```
example : monoid_hom.id x = x :=
by rewrite [id_apply x] -- error:
-- given `monoid_hom.id` but expecting `id`
```

Equality and bundling

Bundling causes a lot of nontrivial synonyms:

```
example : monoid_hom.id x = x :=
by rewrite [id_apply x] -- error:
-- given `monoid_hom.id` but expecting `id`
```

Sometimes you can reduce synonyms with typeclass tricks:
`basis.constr (b : basis I R M) : (I → M') → (M → M')`
is R-linear only if R is commutative.
So we define it as a S-linear map, where R and S commute.
(You can always choose S = ℕ.)

Conclusions

Bundling definitely can help automation, including typeclasses and the simplifier.
It also can help intuitionists avoid the axiom of choice without getting in the way of classical mathematicians.

Bundling tends to cause duplication, and the equality story is unsatisfying.

Conclusions

Bundling definitely can help automation, including typeclasses and the simplifier.
It also can help intuitionists avoid the axiom of choice without getting in the way of classical mathematicians.

Bundling tends to cause duplication, and the equality story is unsatisfying.

Are there clever design patterns to fix disadvantages of bundling, or does better automation make bundling obsolete (for classical maths)?