Generating morphism types using parametricity and Trocq

Cyril Cohen, Assia Mahboubi, Vojtěch Štěpančík

Objective

We rarely care about structures and not their morphisms. Yet formalizing morphisms feels repetitive, mechanical, boring, …

Objective

We rarely care about structures and not their morphisms. Yet formalizing morphisms feels repetitive, mechanical, boring, *automatable*.

Objective

We rarely care about structures and not their morphisms. Yet formalizing morphisms feels repetitive, mechanical, boring, *automatable*.

The user writes a `Record Str (A : □) := { (* ops, sorts, axioms, ... *) }`

Objective

We rarely care about structures and not their morphisms. Yet formalizing morphisms feels repetitive, mechanical, boring, *automatable*.

The user writes a `Record Str (A : □) := { (* ops, sorts, axioms, ... *) }`

Then we want to generate

```
Definition StructQ := {|
  obj := (T : □) × Str T;
  hom (T₁, S₁) (T₂, S₂) := (f : T₁ → T₂) × ??? f S₁ S₂
              (* How do we get this piece? ^ *)
              (* spoiler: it's in the title *)
|}
```

Objective

We rarely care about structures and not their morphisms. Yet formalizing morphisms feels repetitive, mechanical, boring, *automatable*.

The user writes a `Record Str (A : □) := { (* ops, sorts, axioms, ... *) }`

Then we want to generate

```
Definition StructQ := {|
  obj := (T : □) × Str T;
  hom (T₁, S₁) (T₂, S₂) := (f : T₁ → T₂) × ??? f S₁ S₂
                (* How do we get this piece? ^ *)
                (* spoiler: it's in the title *)
|}
```

Eventually adding `id`, `comp`, `idl`, `idr` and `assoc` as well.

Beware of the pipeline

External parametricity refresher: metaprogram $[\![\cdot]\!] : \mathrm{Tm} \to \mathrm{Tm}$ such that $[\![\square]\!] = \lambda(A\ B : \square).A \to B \to \square$ and for $\Gamma \vdash M : T$, we have $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]\ M\ M'$.
Extends to top-level declarations.

Beware of the pipeline

External parametricity refresher: metaprogram $[\![\cdot]\!] : \mathrm{Tm} \to \mathrm{Tm}$ such that $[\![\square]\!] = \lambda(A\ B : \square).A \to B \to \square$ and for $\Gamma \vdash M : T$, we have $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]\ M\ M'$.
Extends to top-level declarations.

```
Record Str (A : □) := { _ }
```

Beware of the pipeline

External parametricity refresher: metaprogram $[\![\cdot]\!] : \mathrm{Tm} \to \mathrm{Tm}$ such that $[\![\square]\!] = \lambda(A\ B : \square).A \to B \to \square$ and for $\Gamma \vdash M : T$, we have $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]\ M\ M'$.
Extends to top-level declarations.

```
Record Str (A : □) := { _ }
```

↧ parametricity translation

```
Record Strᵣ A A' (Aᵣ : A → A' → □) (S : Str A) (S' : Str A') := { _ }
```

Beware of the pipeline

External parametricity refresher: metaprogram $[\![\cdot]\!] : \mathrm{Tm} \to \mathrm{Tm}$ such that $[\![\Box]\!] = \lambda(A\ B : \Box).A \to B \to \Box$ and for $\Gamma \vdash M : T$, we have $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]\ M\ M'$.
Extends to top-level declarations.

```
Record Str (A : □) := { _ }
```

↧ parametricity translation

```
Record Strᵣ A A' (Aᵣ : A → A' → □) (S : Str A) (S' : Str A') := { _ }
```

↧ reparameterize by a function, use `Aᵣ := λ a a'. (f a = a')`

```
Record Strₘ A A' (f : A → A') (S : Str A) (S' : Str A') := { _ }
```

Beware of the pipeline

External parametricity refresher: metaprogram $[\![\cdot]\!] : \mathrm{Tm} \to \mathrm{Tm}$ such that $[\![\Box]\!] = \lambda(A\ B : \Box).A \to B \to \Box$ and for $\Gamma \vdash M : T$, we have $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]\ M\ M'$.
Extends to top-level declarations.

```
Record Str (A : □) := { _ }
```

↧ parametricity translation

```
Record Strr A A' (Ar : A → A' → □) (S : Str A) (S' : Str A') := { _ }
```

↧ reparameterize by a function, use `Ar := λ a a'. (f a = a')`

```
Record Strm A A' (f : A → A') (S : Str A) (S' : Str A') := { _ }
```

↧ get rid of ~~junk~~ contractible data

```
Record Strh A A' (f : A → A') (S : Str A) (S' : Str A') := { _ }
```

It's Morphing time

This works for algebraic theories, e.g.

```
Record SGr (A : □) := { ·: A → A → A; assoc: ∀ x y z, x·(y·z) = (x·y)·z; }
```

It's Morphing time

This works for algebraic theories, e.g.

```
Record SGr (A : □) := { ·: A → A → A; assoc: ∀ x y z, x·(y·z) = (x·y)·z; }
```

↧ parametricity translation

```
Record SGrᵣ A A' (Aᵣ : A → A' → □) ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ᵣ : ∀ x x' (xᵣ : Aᵣ x x') y y' (yᵣ : Aᵣ y y'), Aᵣ (x · y) (x' ·' y')
 ; assocᵣ : (* long type relating a and a' *) ;}
```

It's Morphing time

This works for algebraic theories, e.g.

```
Record SGr (A : □) := { ·: A → A → A; assoc: ∀ x y z, x·(y·z) = (x·y)·z; }
```

↧ parametricity translation

```
Record SGrᵣ A A' (Aᵣ : A → A' → □) ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ᵣ : ∀ x x' (xᵣ : Aᵣ x x') y y' (yᵣ : Aᵣ y y'), Aᵣ (x · y) (x' ·' y')
 ; assocᵣ : (* long type relating a and a' *) ;}
```

↧ reparameterize by a function, use `Aᵣ := λ a a'. (f a = a')`

```
Record SGrₘ A A' (f : A → A') ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ₘ : ∀ x x' (xᵣ : f x = x') y y' (yᵣ : f y = y'), f (x · y) = (x' ·' y')
 ; assocₘ : (* long type relating a and a' *) ;}
```

🪣✨💅

```
Record SGrₘ A A' (f : A → A') ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ₘ : ∀ x x' (xᵣ : f x = x') y y' (yᵣ : f y = y'), f (x · y) = (x' ·' y')
 ; assocₘ : (* long type relating a and a' *); }
```

🪣✨💅

```
Record SGrₘ A A' (f : A → A') ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ₘ : ∀ x x' (xᵣ : f x = x') y y' (yᵣ : f y = y'), f (x · y) = (x' ·' y')
 ; assocₘ : (* long type relating a and a' *); }
```

Contracting singletons: replace $(x, x', x_r : fx = x')$ by (x, fx, refl)

```
⇝ ·ₕ : ∀ x y, f (x · y) = f x ·' f y
```

🪣✨💅

```
Record SGrₘ A A' (f : A → A') ((·, a) : SGr A) ((·', a') : SGr A') :=
 { ·ₘ : ∀ x x' (xᵣ : f x = x') y y' (yᵣ : f y = y'), f (x · y) = (x' ·' y')
 ; assocₘ : (* long type relating a and a' *); }
```

Contracting singletons: replace $(x, x', x_r : fx = x')$ by (x, fx, refl)

⇝ `·ₕ : ∀ x y, f (x · y) = f x ·' f y`

Assuming carriers are sets: the type of `assocₕ` is equivalent to a coherence square of identifications in `A'`, so the whole field can be dropped completely.

$$\begin{array}{ccc} f(x \cdot (y \cdot z)) & \overset{\text{“}\cdot_h\,\cdot_h\text{”}}{=\!=} & f(x) \cdot' (f(y) \cdot' f(z)) \\ {\scriptstyle \mathrm{ap}_f(a)}\,\| & & \|\,{\scriptstyle a'} \\ f((x \cdot y) \cdot z) & \underset{\text{“}\cdot_h\,\cdot_h\text{”}}{=\!=} & (f(x) \cdot' f(y)) \cdot f(z) \end{array}$$

Morphing structures with a GAT behind them

```
Record SGrₕ A A' (f : A → A') ((·, _) : SGr A) ((·', _) : SGr A') :=
 { ·ₕ : ∀ x y, f (x · y) = f x ·' f y ;}
```

Works as expected, looks promising!

Morphing structures with a GAT behind them

```
Record SGrₕ A A' (f : A → A') ((·, _) : SGr A) ((·', _) : SGr A') :=
 { ·ₕ : ∀ x y, f (x · y) = f x ·' f y ;}
```

Works as expected, looks promising! Unfortunately it's useless for GATs: equipping the carrier with e.g. an order `le : A → A → Prop` translates to
`leₕ : ∀ x y, le x y → le' (f x) (f y) → Prop`, while we want
`leₕ : ∀ x y, le x y → le' (f x) (f y)`.

Morphing structures with a GAT behind them

```
Record SGrₕ A A' (f : A → A') ((·, _) : SGr A) ((·', _) : SGr A') :=
 { ·ₕ : ∀ x y, f (x · y) = f x ·' f y ;}
```

Works as expected, looks promising! Unfortunately it's useless for GATs: equipping the carrier with e.g. an order `le : A → A → Prop` translates to
`leₕ : ∀ x y, le x y → le' (f x) (f y) → Prop`, while we want
`leₕ : ∀ x y, le x y → le' (f x) (f y)`.

Fortunately, this can be achieved by guiding the parametricity translation with annotations from the Trocq framework.

⚠️ **TRADE OFFER** ⚠️ i receive: $(4, 0)$; you receive: a function

`Prop` and □ may be annotated by a pair of indices, which manifest as extra data imposed on the translated type of relations.

⚠️ TRADE OFFER ⚠️ i receive: $(4, 0)$; you receive: a function

`Prop` and □ may be annotated by a pair of indices, which manifest as extra data imposed on the translated type of relations.

$$[\![\square^{(4,0)}]\!]\ A\ B = ⧈^{(4,0)}\ A\ B = (R : A \to B \to \square) \times (m : A \to B) \times (Rab \simeq (ma = b))$$

$$[\![\square^{(0,4)}]\!]\ A\ B = ⧈^{(0,4)}\ A\ B = (R : A \to B \to \square) \times (m : B \to A) \times (Rab \simeq (a = mb))$$

$$[\![\square^{(4,4)}]\!]\ A\ B = ⧈^{(4,4)}\ A\ B = (R : A \to B \to \square) \times (e : A \simeq B) \quad \times (Rab \simeq (ea = b))$$

⚠️ TRADE OFFER ⚠️ i receive: $(4, 0)$; you receive: a function

`Prop` and $\square$ may be annotated by a pair of indices, which manifest as extra data imposed on the translated type of relations.

$$[\![\square^{(4,0)}]\!]\ A\ B = \circledcirc^{(4,0)}\ A\ B = (R : A \to B \to \square) \times (m : A \to B) \times (Rab \simeq (ma = b))$$

$$[\![\square^{(0,4)}]\!]\ A\ B = \circledcirc^{(0,4)}\ A\ B = (R : A \to B \to \square) \times (m : B \to A) \times (Rab \simeq (a = mb))$$

$$[\![\square^{(4,4)}]\!]\ A\ B = \circledcirc^{(4,4)}\ A\ B = (R : A \to B \to \square) \times (e : A \simeq B) \quad \times (Rab \simeq (ea = b))$$

Declaring `le : A → A → Prop40` generates the expected order preserving morphisms. Declaring it as `Prop04` generates order reflecting maps.

The quest for continuous functions

```
Record Top (A : □) := { open : (A → Prop??) → Prop??; (* axioms *) ;}
```

The quest for continuous functions

```
Record Top (A : □) := { open : (A → Prop??) → Prop??; (* axioms *) ;}
```

The inner `Prop` picks elements of a subset, we want to talk about *exactly* preimages. The outer one picks open subsets, which should be reflected.

The quest for continuous functions

```
Record Top (A : □) := { open : (A → Prop44) → Prop04; (* axioms *) ;}
```

↧ Trocq parametricity translation, instantiation

```
Record Topₘ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₘ : ∀ (U : A → Prop) (U' : A' → Prop)
             (Uᵣ : ∀ x x' (xᵣ : f x = x'), ◘_P^{4,4} (U a) (U' a')),
           ◘_P^{0,4} (open U) (open' U') ; (* translation of axioms *)}
```

The quest for continuous functions

```
Record Top (A : □) := { open : (A → Prop44) → Prop04; (* axioms *) ;}
```

↧ Trocq parametricity translation, instantiation

```
Record Topₘ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₘ : ∀ (U : A → Prop) (U' : A' → Prop)
             (Uᵣ : ∀ x x' (xᵣ : f x = x'), ⧈_P^{4,4} (U a) (U' a')),
           ⧈_P^{0,4} (open U) (open' U') ; (* translation of axioms *)}
```

Topology axioms have the shape `... -> open ?`, so their translations have the shape `... -> openₘ ? ?`. But we have a proof that that's equivalent to `? = m ?`, equality in `open`, which is contractible.

Contracting `(x', xᵣ)` leaves `Uᵣ : ∀ x, U x ≃ U' (f x)`, so we may contract `(U, Uᵣ)`.

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

✅ … on paper

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

✅ … on paper

⌚ full implementation in rocq-elpi

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

✅ … on paper

⌚ full implementation in rocq-elpi

⌚ generating categories, jww Rafaël Bocquet and Kenji Maillard

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

✅ … on paper

⌚ full implementation in rocq-elpi

⌚ generating categories, jww Rafaël Bocquet and Kenji Maillard

Thank you.

Conclusion

```
Record Topₕ A A' (f : A → A') (S : Top A) (S' : Top A') :=
 { openₕ : ∀ (U' : A' → Prop), open' U' → open (U' ∘ f) ;}
```

✅ Algebraic theories

✅ GATs

✅ GATs with different variances

✅ (some?) non algebraic structures

✅ … on paper

⌚ full implementation in rocq-elpi

⌚ generating categories, jww Rafaël Bocquet and Kenji Maillard

I don't have a podcast,
but maybe check out
Mikan on codeberg.

Thank you.