Emily Riehl

Johns Hopkins University

Challenges in (auto)formalizing category theory

ItaLean 2025: Bridging Formal Mathematics and AI

Framing the question

If we believe in computer formalization as a paradigm for mathematical proof
— perhaps accelerated by collaborations with AI —
we must ask ourselves what advances are needed
for this to be feasible in all areas of mathematics.

By contrast, Kevin Buzzard, a mathematician at Imperial College London, did not find AlphaProof useful for translating the proof of Fermat's Last Theorem into Lean. The difference between his experience and my own probably lies in the number of custom definitions in our proofs. I had deliberately asked the PhD students for lemmas that mostly used standard definitions that were already in mathlib; Buzzard's proof development, however, was full of what he described as "bespoke definitions". "In my experience," he wrote, "no AI system is anywhere near useful to me right now."

— Talia Ringer, "Mathematicians put AI model AlphaProof to the test,"
Nature, 12 November 2025

Plan

A distinctive feature of category theory is that
the definitions and theorem statements are hard but the proofs are easy.

Once a result is stated at the correct level of abstraction,
the proof can be found by following your nose.

This talk will illustrate this principle with a few examples and then
discuss its implications for computer formalization by either humans or AI.

1. Examples from category theory

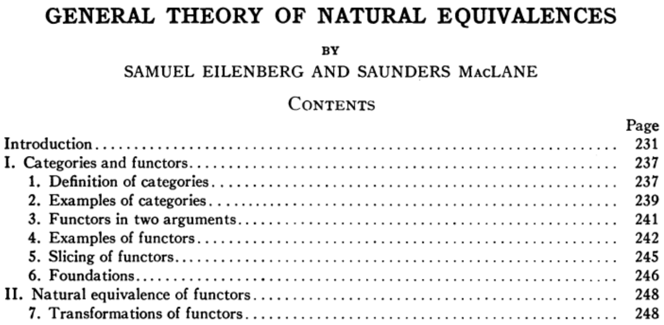
2. Challenges in (auto)formalization

1

Examples from category theory

A field born from a definition

Category theory was founded by Samuel Eilenberg and Saunders Mac Lane in their 1945 paper "General theory of natural equivalences" which invents a mathematical language to give a precise meaning to the term "natural equivalence".



GENERAL THEORY OF NATURAL EQUIVALENCES

BY

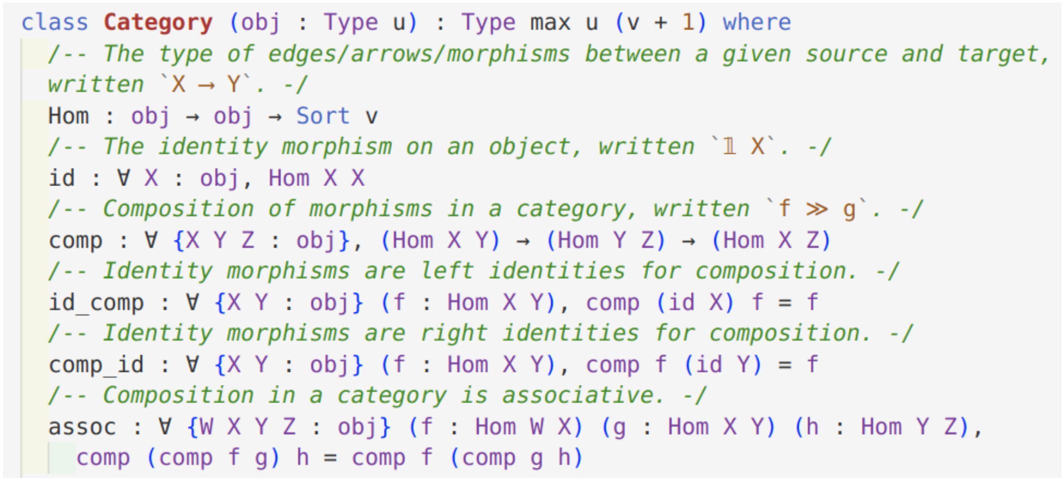
SAMUEL EILENBERG AND SAUNDERS MACLANE

CONTENTS

To define natural equivalences — now called natural isomorphisms —
one must first define categories and functors.

Categories

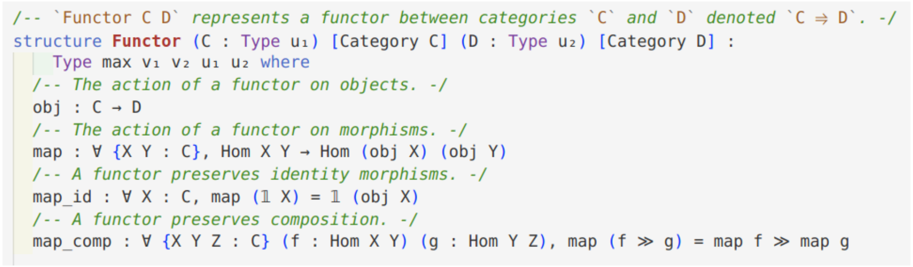
A category has a type of objects and a dependent type of arrows with composition and identities satisfying certain axioms.



```
class Category (obj : Type u) : Type max u (v + 1) where
  /-- The type of edges/arrows/morphisms between a given source and target,
  written `X → Y`. -/
  Hom : obj → obj → Sort v
  /-- The identity morphism on an object, written `𝟙 X`. -/
  id : ∀ X : obj, Hom X X
  /-- Composition of morphisms in a category, written `f ≫ g`. -/
  comp : ∀ {X Y Z : obj}, (Hom X Y) → (Hom Y Z) → (Hom X Z)
  /-- Identity morphisms are left identities for composition. -/
  id_comp : ∀ {X Y : obj} (f : Hom X Y), comp (id X) f = f
  /-- Identity morphisms are right identities for composition. -/
  comp_id : ∀ {X Y : obj} (f : Hom X Y), comp f (id Y) = f
  /-- Composition in a category is associative. -/
  assoc : ∀ {W X Y Z : obj} (f : Hom W X) (g : Hom X Y) (h : Hom Y Z),
    comp (comp f g) h = comp f (comp g h)
```

Functors

A functor is a mapping between categories, sending objects to objects and arrows to arrows while preserving all of the structure.



```
/-- `Functor C D` represents a functor between categories `C` and `D` denoted `C ⥤ D`. -/
structure Functor (C : Type u₁) [Category C] (D : Type u₂) [Category D] :
    Type max v₁ v₂ u₁ u₂ where
  /-- The action of a functor on objects. -/
  obj : C → D
  /-- The action of a functor on morphisms. -/
  map : ∀ {X Y : C}, Hom X Y → Hom (obj X) (obj Y)
  /-- A functor preserves identity morphisms. -/
  map_id : ∀ X : C, map (𝟙 X) = 𝟙 (obj X)
  /-- A functor preserves composition. -/
  map_comp : ∀ {X Y Z : C} (f : Hom X Y) (g : Hom Y Z), map (f ≫ g) = map f ≫ map g
```

$$\text{Ring} \xrightarrow{(-)^{\times}} \text{Type} \qquad\qquad \text{C} \xrightarrow{\text{Hom}(A,-)} \text{Type}$$

Examples:

$$\begin{tikzcd} \mathbb{Z} \arrow[r, mapsto] \arrow[d, "\text{mod}3"'] & \{1,-1\} \arrow[d, "\text{mod}3"] \\ \mathbb{Z}/3 \arrow[r, mapsto] & \{1,2\} \end{tikzcd} \qquad \begin{tikzcd} X \arrow[r, mapsto] \arrow[d, "f"'] & \text{Hom}(A,X) \arrow[d, "-\gg f"] \\ Y \arrow[r, mapsto] & \text{Hom}(A,Y) \end{tikzcd}$$

Natural transformations and corepresentable functors

A natural transformation is a mapping between parallel functors whose components are arrows in the target category indexed by objects in the source category.

```
structure NatTrans (F G : C ⥤ D) : Type max u₁ v₂ where
  /-- The component of a natural transformation. -/
  app : ∀ X : C, Hom (F.obj X) (G.obj X)
  /-- The naturality square for a given morphism. -/
  naturality : ∀ {X Y : C} (f : Hom X Y), F.map f ≫ app Y = app X ≫ G.map f
```

A natural isomorphism is a natural transformation whose components are isomorphisms.

Example:
$$\begin{tikzcd} \mathsf{Ring} \arrow[r, bend left, "\mathsf{Hom}(\mathbb{Z}[x,x^{-1}],-)"{name=U}] \arrow[r, bend right, "(-)^\times"'{name=D}] & \mathsf{Type} \arrow[from=U, to=D, Rightarrow, "\cong"'] \end{tikzcd}$$

$$\begin{tikzcd} \mathsf{Ring} & \mathsf{Type} \\ & \mathsf{Hom}(\mathbb{Z}[x,x^{-1}],R) \arrow[d, "\cong"', "\phi\mapsto\phi(x)"] \\ R \arrow[r, phantom, "\mapsto"] & {} \\ & R^\times \end{tikzcd}$$

Characterizing corepresentable functors

A functor $F\colon \mathsf{C} \to \mathsf{Type}$ is corepresentable if
F is naturally isomorphic to $\mathrm{Hom}(A, -)\colon \mathsf{C} \to \mathsf{Type}$ for some object $A : \mathsf{C}$.

Theorem. A functor $F\colon \mathsf{C} \to \mathsf{Type}$ is corepresentable if and only if
its category of elements $\mathrm{el}F$ has an initial object.

Given a functor $F\colon \mathsf{C} \to \mathsf{Type}$, its category of elements $\mathrm{el}F$ has

- objects given by pairs (X, x) with $X : \mathsf{C}$ and $x : FX$
- arrows $f : (X, x) \to (Y, y)$ given by arrows $f\colon X \to Y$ in C so that the function $Ff : FX \to FY$ sends x to y.

An object $I : \mathsf{C}$ is initial if for all objects $X : \mathsf{C}$ there exists a unique arrow $I \to X$.

Mathematical expositor's fallacy

Before continuing, we acknowledge the mathematical expositor's fallacy:

A mathematical proof can be understood in real time
provided all of the technical terms have been defined.

Nevertheless we continue ...

A proof by following your nose

Theorem. A functor $F\colon \mathsf{C} \to \mathsf{Type}$ is corepresentable if and only if its category of elements $\mathsf{el}F$ has an initial object.

Proof ($\Leftarrow$): Suppose (I, i) is an initial object in $\mathsf{el}F$. We will show that F is naturally isomorphic to the functor $\mathsf{Hom}(I, -)$, by defining a natural equivalence of types $\mathsf{Hom}(I, X) \simeq FX$.

- To define $\mathsf{Hom}(I, X) \to FX$, note that each arrow $f : \mathsf{Hom}(I, X)$ defines a function $Ff\colon FI \to FX$ by applying the functor F. We obtain an element of FX by applying Ff to the element $i : FI$.
- To define $FX \to \mathsf{Hom}(I, X)$, note that each $x : FX$ defines an object (X, x) in $\mathsf{el}F$. By initiality of (I, i) there is a unique arrow $f\colon (I, i) \to (X, x)$ in $\mathsf{el}F$, which gives the data of an arrow $f : \mathsf{Hom}(I, X)$ in C.

An arrow $f\colon (I, i) \to (X, x)$ in $\mathsf{el}F$ has the property that $Ff(i) = x$, so the second function is a right inverse of the first. By the uniqueness of arrows mapping out of an initial object, the second function is also a left inverse of the first. $\square$

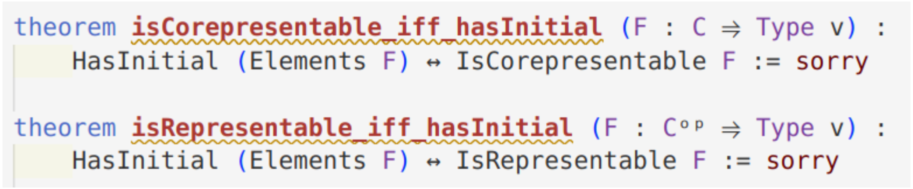
2

Challenges in (auto)formalization

Challenges for autoformalization agents

Theorem. A functor F: C $\to$ Type is corepresentable if and only if its category of elements el$F$ has an initial object.

The theorem we've just discussed is not currently in Mathlib, though all of the definitions references appear there. Thus it can be used to test the capacity of an autoformalization agent to prove a very easy theorem in category theory.

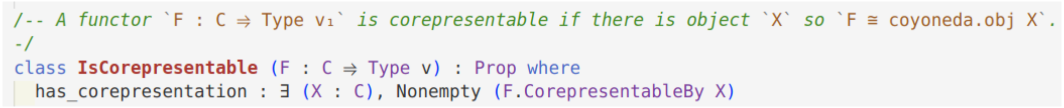


```
theorem isCorepresentable_iff_hasInitial (F : C ⥤ Type v) :
    HasInitial (Elements F) ↔ IsCorepresentable F := sorry

theorem isRepresentable_iff_hasInitial (F : Cᵒᵖ ⥤ Type v) :
    HasInitial (Elements F) ↔ IsRepresentable F := sorry
```

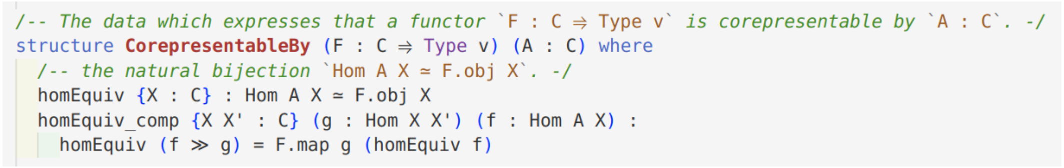
Proving theorems by constructing data

It is a property of a functor $F\colon \mathsf{C} \to \mathsf{Type}$ to be corepresentable:



```
/-- A functor `F : C ⥤ Type v₁` is corepresentable if there is object `X` so `F ≅ coyoneda.obj X`.
-/
class IsCorepresentable (F : C ⥤ Type v) : Prop where
  has_corepresentation : ∃ (X : C), Nonempty (F.CorepresentableBy X)
```

But to prove that a particular F has this property requires constructing explicit data, of an object $A : \mathsf{C}$ together with a natural isomorphism $\mathrm{Hom}(A, X) \simeq FX$ to show that the type F.`CorepresentableBy` A is nonempty:



```
/-- The data which expresses that a functor `F : C ⥤ Type v` is corepresentable by `A : C`. -/
structure CorepresentableBy (F : C ⥤ Type v) (A : C) where
  /-- the natural bijection `Hom A X ≃ F.obj X`. -/
  homEquiv {X : C} : Hom A X ≃ F.obj X
  homEquiv_comp {X X' : C} (g : Hom X X') (f : Hom A X) :
    homEquiv (f ≫ g) = F.map g (homEquiv f)
```

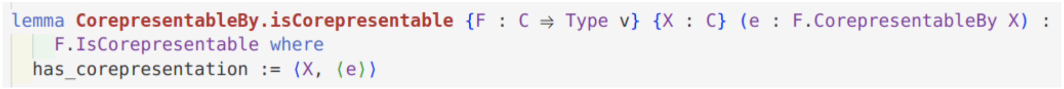
Thus, autoformalization agents need to be able to work with Lean's definitions "`def`" and not just its propositions "`theorem`".

Maintaining computable data

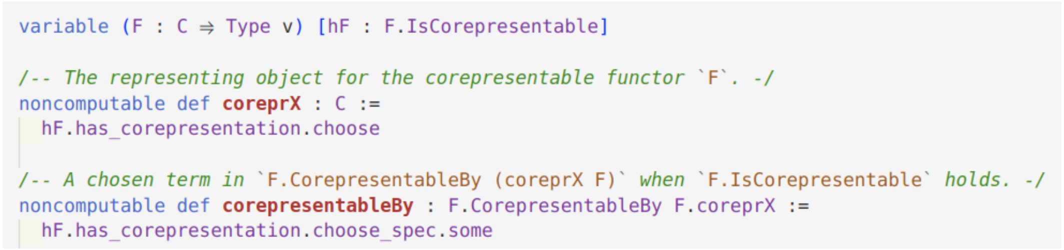
It's possible to translate back and forth between

- the proposition "F.IsCorepresentable" and
- the type "F.CorprepresentableBy A"

but with some loss of computability:



```lean
lemma CorepresentableBy.isCorepresentable {F : C ⥤ Type v} {X : C} (e : F.CorepresentableBy X) :
    F.IsCorepresentable where
  has_corepresentation := ⟨X, ⟨e⟩⟩
```



```lean
variable (F : C ⥤ Type v) [hF : F.IsCorepresentable]

/-- The representing object for the corepresentable functor `F`. -/
noncomputable def coreprX : C :=
  hF.has_corepresentation.choose

/-- A chosen term in `F.CorepresentableBy (coreprX F)` when `F.IsCorepresentable` holds. -/
noncomputable def corepresentableBy : F.CorepresentableBy F.coreprX :=
  hF.has_corepresentation.choose_spec.some
```

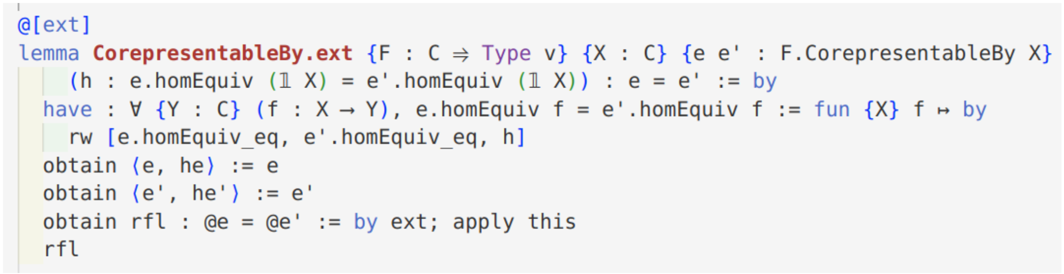
Good news: definitions as homotopy propositions

If an autoformalization agent constructs an element of an arbitrary type rather than a proof of a proposition, how do we know the element was the correct one?

The types that encode categorical definitions tend to be homotopy propositions in the sense of homotopy type theory:

A type `A` is a (homotopy) proposition if any two elements can be identified:

for all $x, y :$ `A` there is an equality $e : x = y$.

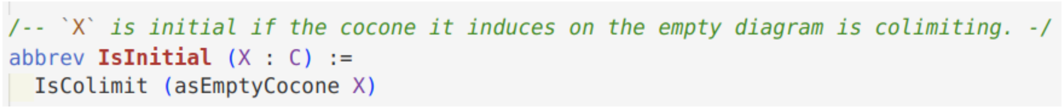


```
@[ext]
lemma CorepresentableBy.ext {F : C ⥤ Type v} {X : C} {e e' : F.CorepresentableBy X}
    (h : e.homEquiv (𝟙 X) = e'.homEquiv (𝟙 X)) : e = e' := by
  have : ∀ {Y : C} (f : X ⟶ Y), e.homEquiv f = e'.homEquiv f := fun {X} f ↦ by
    rw [e.homEquiv_eq, e'.homEquiv_eq, h]
  obtain ⟨e, he⟩ := e
  obtain ⟨e', he'⟩ := e'
  obtain rfl : @e = @e' := by ext; apply this
  rfl
```

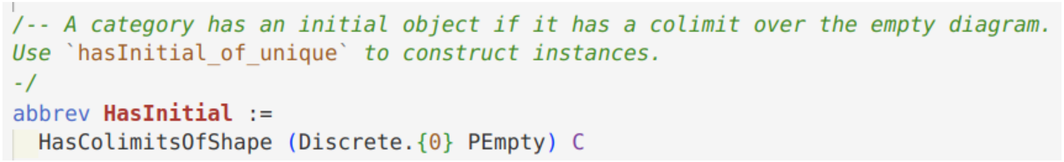
Working with complicated API

An object I : C is initial if for all objects X : C there exists a unique arrow $I \to X$.

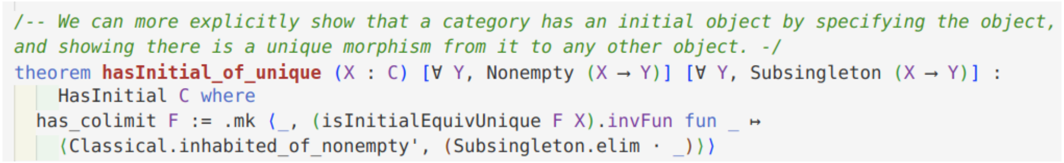
A simple definition in category theory often has a complicated definition in `mathlib`.



```
/-- `X` is initial if the cocone it induces on the empty diagram is colimiting. -/
abbrev IsInitial (X : C) :=
  IsColimit (asEmptyCocone X)
```



```
/-- A category has an initial object if it has a colimit over the empty diagram.
Use `hasInitial_of_unique` to construct instances.
-/
abbrev HasInitial :=
  HasColimitsOfShape (Discrete.{0} PEmpty) C
```



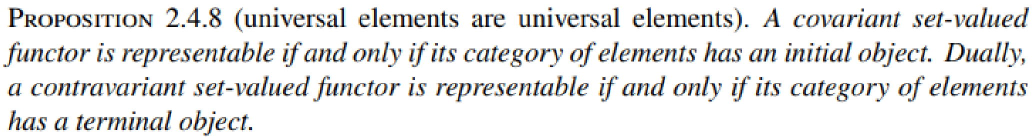
```
/-- We can more explicitly show that a category has an initial object by specifying the object,
and showing there is a unique morphism from it to any other object. -/
theorem hasInitial_of_unique (X : C) [∀ Y, Nonempty (X → Y)] [∀ Y, Subsingleton (X → Y)] :
    HasInitial C where
  has_colimit F := .mk ⟨_, (isInitialEquivUnique F X).invFun fun _ ↦
    ⟨Classical.inhabited_of_nonempty', (Subsingleton.elim · _)⟩⟩
```

Determining co- or contra-variance

Any category C has an opposite category C^{op} defined by turning around the arrows.

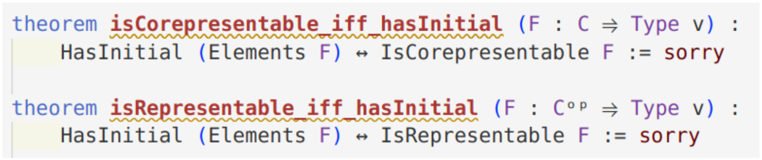
Note C is equivalent to $\mathsf{C}^{\mathrm{opop}}$ so any functor $F\colon \mathsf{C} \to \mathsf{Type}$ equally defines a functor $F\colon \mathsf{C}^{\mathrm{opop}} \to \mathsf{Type}$.

It is conventional to define $\mathsf{el}F$ differently for $F\colon \mathsf{C}^{\mathrm{op}} \to \mathsf{Type}$ so that there is a forgetful functor $U\colon \mathsf{el}F \to \mathsf{C}$, rather than to C^{op}. With these conventions:



Proposition 2.4.8 (universal elements are universal elements). *A covariant set-valued functor is representable if and only if its category of elements has an initial object. Dually, a contravariant set-valued functor is representable if and only if its category of elements has a terminal object.*

But this is not how this is handled in `mathlib`:

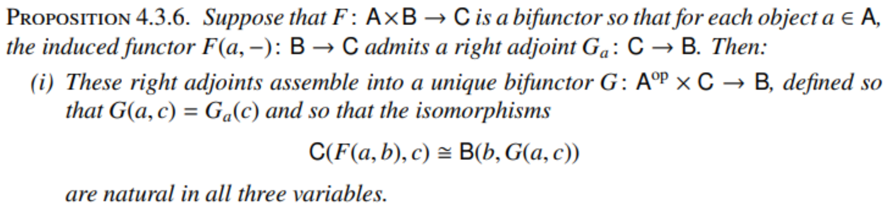


```
theorem isCorepresentable_iff_hasInitial (F : C ⥤ Type v) :
    HasInitial (Elements F) ↔ IsCorepresentable F := sorry

theorem isRepresentable_iff_hasInitial (F : Cᵒᵖ ⥤ Type v) :
    HasInitial (Elements F) ↔ IsRepresentable F := sorry
```

Extending nested structures

Another result that is not currently in `mathlib` is the following:



Proposition 4.3.6. *Suppose that $F\colon \mathsf{A}\times\mathsf{B}\to\mathsf{C}$ is a bifunctor so that for each object $a\in\mathsf{A}$, the induced functor $F(a,-)\colon\mathsf{B}\to\mathsf{C}$ admits a right adjoint $G_a\colon\mathsf{C}\to\mathsf{B}$. Then:*

(i) These right adjoints assemble into a unique bifunctor $G\colon\mathsf{A}^{\mathrm{op}}\times\mathsf{C}\to\mathsf{B}$, defined so that $G(a,c)=G_a(c)$ and so that the isomorphisms

$$\mathsf{C}(F(a,b),c)\cong\mathsf{B}(b,G(a,c))$$

are natural in all three variables.

The uniqueness part of this statement is rather subtle to formalize.

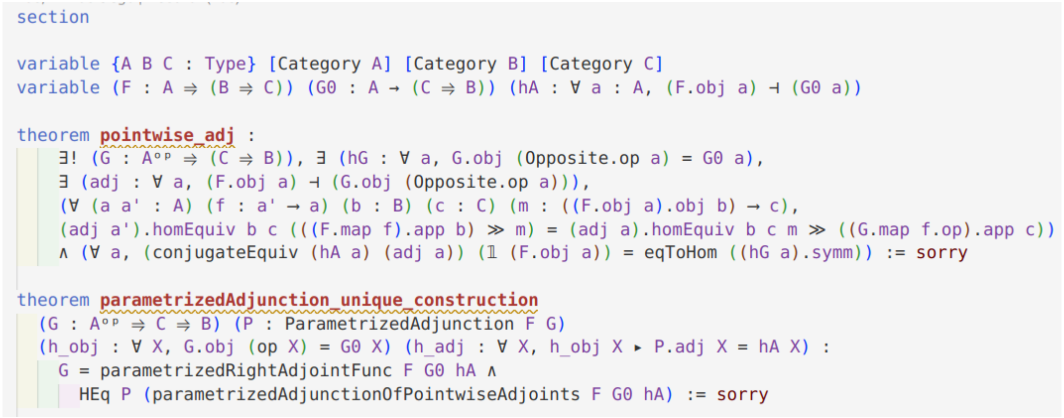
It asserts that

- given a function that for each object $a : \mathsf{A}$ defines a functor $G_a : \mathsf{C} \to \mathsf{B}$,
- together with the data of an adjunction $Fa \dashv G_a$ for each $a : \mathsf{A}$,

then there exists a unique functor $G\colon \mathsf{A}^{\mathrm{op}} \to \mathsf{C} \to \mathsf{B}$ extending the mapping on objects so that the adjoint hom equivalences are natural in A (as well as B and C).

Dependent type theory hell

Gauss and Aristotle each took different approaches to autoformalizing the statement:



```
section

variable {A B C : Type} [Category A] [Category B] [Category C]
variable (F : A ⥤ (B ⥤ C)) (G0 : A → (C ⥤ B)) (hA : ∀ a : A, (F.obj a) ⊣ (G0 a))

theorem pointwise_adj :
    ∃! (G : Aᵒᵖ ⥤ (C ⥤ B)), ∃ (hG : ∀ a, G.obj (Opposite.op a) = G0 a),
    ∃ (adj : ∀ a, (F.obj a) ⊣ (G.obj (Opposite.op a))),
    (∀ (a a' : A) (f : a' ⟶ a) (b : B) (c : C) (m : ((F.obj a).obj b) ⟶ c),
    (adj a').homEquiv b c (((F.map f).app b) ≫ m) = (adj a).homEquiv b c m ≫ ((G.map f.op).app c))
    ∧ (∀ a, (conjugateEquiv (hA a) (adj a)) (𝟙 (F.obj a)) = eqToHom ((hG a).symm)) := sorry

theorem parametrizedAdjunction_unique_construction
  (G : Aᵒᵖ ⥤ C ⥤ B) (P : ParametrizedAdjunction F G)
  (h_obj : ∀ X, G.obj (op X) = G0 X) (h_adj : ∀ X, h_obj X ▸ P.adj X = hA X) :
    G = parametrizedRightAdjointFunc F G0 hA ∧
      HEq P (parametrizedAdjunctionOfPointwiseAdjoints F G0 hA) := sorry
```

Both statements require heterogeneous equality between elements of different fibers of the constituent dependent types.

Invisible mathematics

The previous result is used to construct

- parametrized adjunctions involving a pair of bifunctors $F\colon \mathsf{A} \to \mathsf{B} \to \mathsf{C}$ and $G\colon \mathsf{A}^{\mathsf{op}} \to \mathsf{C} \to \mathsf{B}$ and
- two-variable adjunctions, involving a third bifunctor $H\colon \mathsf{B}^{\mathsf{op}} \to \mathsf{C} \to \mathsf{A}$.

The former notion is currently in Mathlib while the latter is not.

One of the most difficult results of an autoformalization challenge I built around these notions turned out to be the following lemma, which I thought was trivial:

Lemma. Any two variable adjunction between F, G, H defines three parametrized adjunctions, one for each pair of bifunctors.

To define the third of these parametrized adjunctions, we must first

- "flip" the arguments of $G : \mathsf{C} \to \mathsf{A}^{\mathsf{op}} \to \mathsf{B}$ and $H\colon \mathsf{C} \to \mathsf{B}^{\mathsf{op}} \to \mathsf{A}$,
- precompose H with the equivalence $\mathsf{C}^{\mathsf{opop}} \simeq \mathsf{C}$, and
- "op" the functor $G\colon \mathsf{C}^{\mathsf{op}} \to (\mathsf{A}^{\mathsf{op}} \to \mathsf{B})^{\mathsf{op}}$ and then postcompose with the equivalence $(\mathsf{A}^{\mathsf{op}} \to \mathsf{B})^{\mathsf{op}} \simeq (\mathsf{A} \to \mathsf{B}^{\mathsf{op}})$.

Present status

The autoformalization challenges I just described are very easy.

- The corepresentability theorem takes about 50 lines to formalize.
- The parametrized/two-variable adjunction theorem is left as an exercise in my book.

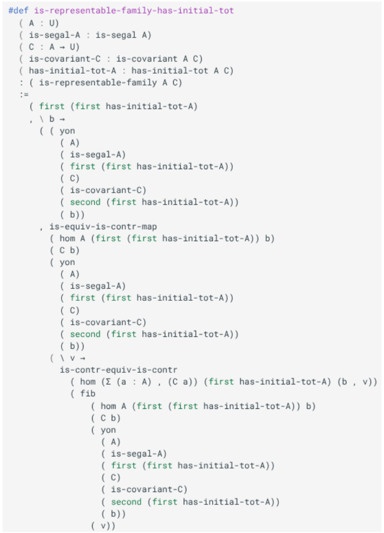
Success requires `mathlib`-accepted code.

My work involves an infinite-dimensional generalization of category theory — ∞-category theory — that is now several decades old.

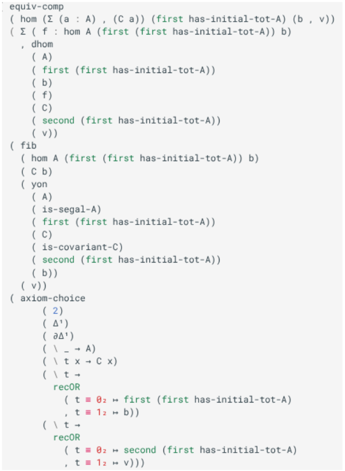
- Neither of these ∞-theorems are anywhere close to being stateable in `mathlib`.
- And even if they were, would it be worth the monumental effort that it would take to formalize a surely unreadable proof?

Aside: in the literature that uses ∞-category theory, it is common to cite results like this without proof.

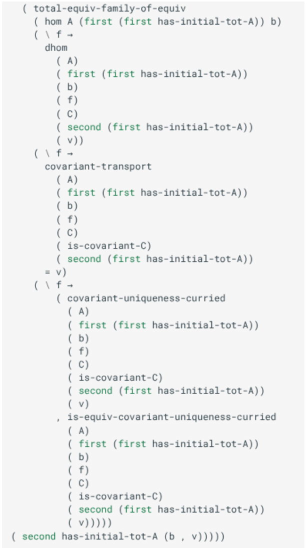
Future dream: other proof assistants / formal systems?



```
#def is-representable-family-has-initial-tot
  ( A : U)
  ( is-segal-A : is-segal A)
  ( C : A → U)
  ( is-covariant-C : is-covariant A C)
  ( has-initial-tot-A : has-initial-tot A C)
  : ( is-representable-family A C)
  :=
    ( first (first has-initial-tot-A)
    , \ b →
      ( ( yon
          ( A)
          ( is-segal-A)
          ( first (first has-initial-tot-A))
          ( C)
          ( is-covariant-C)
          ( second (first has-initial-tot-A))
          ( b))
      , is-equiv-is-contr-map
        ( hom A (first (first has-initial-tot-A)) b)
        ( C b)
        ( yon
          ( A)
          ( is-segal-A)
          ( first (first has-initial-tot-A))
          ( C)
          ( is-covariant-C)
          ( second (first has-initial-tot-A))
          ( b))
        ( \ v →
          is-contr-equiv-is-contr
            ( hom (Σ (a : A) , (C a)) (first has-initial-tot-A) (b , v))
            ( fib
                ( hom A (first (first has-initial-tot-A)) b)
                ( C b)
                ( yon
                  ( A)
                  ( is-segal-A)
                  ( first (first has-initial-tot-A))
                  ( C)
                  ( is-covariant-C)
                  ( second (first has-initial-tot-A))
                  ( b))
                ( v))
```



```
equiv-comp
( hom (Σ (a : A) , (C a)) (first has-initial-tot-A) (b , v))
( Σ ( f : hom A (first (first has-initial-tot-A)) b)
  , dhom
    ( A)
    ( first (first has-initial-tot-A))
    ( b)
    ( f)
    ( C)
    ( second (first has-initial-tot-A))
    ( v))
( fib
  ( hom A (first (first has-initial-tot-A)) b)
  ( C b)
  ( yon
    ( A)
    ( is-segal-A)
    ( first (first has-initial-tot-A))
    ( C)
    ( is-covariant-C)
    ( second (first has-initial-tot-A))
    ( b))
  ( v))
( axiom-choice
      ( 2)
      ( Δ¹)
      ( ∂Δ¹)
      ( \ _ → A)
      ( \ t x → C x)
      ( \ t →
        recOR
          ( t ≡ 0₂ ↦ first (first has-initial-tot-A)
          , t ≡ 1₂ ↦ b))
      ( \ t →
        recOR
          ( t ≡ 0₂ ↦ second (first has-initial-tot-A)
          , t ≡ 1₂ ↦ v)))
```



```
  ( total-equiv-family-of-equiv
    ( hom A (first (first has-initial-tot-A)) b)
    ( \ f →
      dhom
        ( A)
        ( first (first has-initial-tot-A))
        ( b)
        ( f)
        ( C)
        ( second (first has-initial-tot-A))
        ( v))
    ( \ f →
      covariant-transport
        ( A)
        ( first (first has-initial-tot-A))
        ( b)
        ( f)
        ( C)
        ( is-covariant-C)
        ( second (first has-initial-tot-A))
      = v)
    ( \ f →
        ( covariant-uniqueness-curried
          ( A)
          ( first (first has-initial-tot-A))
          ( b)
          ( f)
          ( C)
          ( is-covariant-C)
          ( second (first has-initial-tot-A))
          ( v)
        , is-equiv-covariant-uniqueness-curried
          ( A)
          ( first (first has-initial-tot-A))
          ( b)
          ( f)
          ( C)
          ( is-covariant-C)
          ( second (first has-initial-tot-A))
          ( v)))))
( second has-initial-tot-A (b , v)))))
```

The ∞-categorical version of the corepresentability theorem is formalized in `Rzk`.

Questions?

Grazie!