Lean 4 Code: AffineTransitionLimit

Mathlib4

September 6, 2025

1 Source Code

The following is the Lean 4 source code from `AffineTransitionLimit.lean`:

```lean
/-
Copyright (c) 2025 Andrew Yang. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Andrew Yang
-/
import Mathlib.AlgebraicGeometry.IdealSheaf.Functorial
import Mathlib.AlgebraicGeometry.Morphisms.Separated
import Mathlib.CategoryTheory.Filtered.Final

/-!

# Inverse limits of schemes with affine transition maps

In this file, we develop API for inverse limits of schemes with affine transition maps,
following EGA IV 8 and https://stacks.math.columbia.edu/tag/01YT.

-/

universe uI u

open AlgebraicGeometry CategoryTheory Limits

-- We refrain from considering diagrams in the over category since inverse limits in the over
-- category is isomorphic to limits in `Scheme`. Instead we use `D ⟶ (Functor.const I).obj S` to
-- say that the diagram is over the base scheme `S`.
variable {I : Type u} [Category.{u} I] {S X : Scheme.{u}} (D : I     Scheme.{u})
  (t : D ⟶ (Functor.const I).obj S) (f : X ⟶ S) (c : Cone D) (hc : IsLimit c)

include hc in
/--
Suppose we have a cofiltered diagram of nonempty quasi-compact schemes,
whose transition maps are affine. Then the limit is also nonempty.
-/
@[stacks 01Z2]
lemma Scheme.nonempty_of_isLimit [IsCofilteredOrEmpty I]
    [∀ {i j} (f : i ⟶ j), IsAffineHom (D.map f)] [∀ i, Nonempty (D.obj i)]
    [∀ i, CompactSpace (D.obj i)] :
    Nonempty c.pt := by
  classical
  cases isEmpty_or_nonempty I
  · have e := (isLimitEquivIsTerminalOfIsEmpty _ _ hc).uniqueUpToIso specULiftZIsTerminal
    exact Nonempty.map e.inv.base inferInstance
  · have i := Nonempty.some    Nonempty    I
    have : IsCofiltered I := ⟨⟩
    let 𝒰 := (D.obj i).affineCover.finiteSubcover
    have (i' : _) : IsAffine (𝒰.obj i') := inferInstanceAs (IsAffine (Spec _))
    obtain ⟨j, H⟩ :
        ∃ j : 𝒰.J, ∀ {i'} (f : i' ⟶ i), Nonempty ((𝒰.pullbackCover (D.map f)).obj j) := by
      simp_rw [← not_isEmpty_iff]
      by_contra! H
      choose i' f hf using H
      let g (j) := IsCofiltered.infTo (insert i (Finset.univ.image i'))
        (Finset.univ.image fun j : 𝒰.J     ⟨_, _, by simp, by simp, f j⟩) (X := j)
      have (j : 𝒰.J) : IsEmpty ((𝒰.pullbackCover (D.map (g i (by simp)))).obj j) := by
        let F : (𝒰.pullbackCover (D.map (g i (by simp)))).obj j ⟶
            (𝒰.pullbackCover (D.map (f j))).obj j :=
          pullback.map _ _ _ _ (D.map (g _ (by simp))) (     _) (     _) (by
            rw [← D.map_comp, IsCofiltered.infTo_commutes]
            · simp [g]
            · simp
            · exact Finset.mem_image_of_mem _ (Finset.mem_univ _)) (by simp)
        exact Function.isEmpty F.base
      obtain ⟨x, -⟩ :=
        (𝒰.pullbackCover (D.map (g i (by simp)))).covers (Nonempty.some inferInstance)
      exact (this _).elim x
    let F := Over.post D     Over.pullback (𝒰.map j)     Over.forget _
    have (i' : _) : IsAffine (F.obj i') :=
      have : IsAffineHom (pullback.snd (D.map i'.hom) (𝒰.map j)) :=
        MorphismProperty.pullback_snd _ _ inferInstance
      isAffine_of_isAffineHom (pullback.snd (D.map i'.hom) (𝒰.map j))
    have (i' : _) : Nonempty (F.obj i') := H i'.hom
    let e : F ⟶ (F     Scheme.Γ.rightOp)     Scheme.Spec := Functor.whiskerLeft F ΓSpec.adjunction.unit
    have (i : _) : IsIso (e.app i) := IsAffine.affine
    have : IsIso e := NatIso.isIso_of_isIso_app e
    let c' : LimitCone F := ⟨_, (IsLimit.postcomposeInvEquiv (asIso e) _).symm
      (isLimitOfPreserves Scheme.Spec (limit.isLimit (F     Scheme.Γ.rightOp)))⟩
    have : Nonempty c'.1.pt := by
      apply (config := { allowSynthFailures := true }) PrimeSpectrum.instNonemptyOfNontrivial
      have (i' : _) : Nontrivial ((F     Scheme.Γ.rightOp).leftOp.obj i') := by
        apply (config := { allowSynthFailures := true }) Scheme.component_nontrivial
        simp
      exact CommRingCat.FilteredColimits.nontrivial
        (isColimitCoconeLeftOpOfCone _ (limit.isLimit (F     Scheme.Γ.rightOp)))
    let α : F ⟶ Over.forget _     D := Functor.whiskerRight
      (Functor.whiskerLeft (Over.post D) (Over.mapPullbackAdj (𝒰.map j)).counit) (Over.forget _)
    exact this.map (((Functor.Initial.isLimitWhiskerEquiv (Over.forget i) c).symm hc).lift
      ((Cones.postcompose α).obj c'.1)).base

include hc in
open Scheme.IdealSheafData in
```

```
/--
Suppose we have a cofiltered diagram of schemes whose transition maps are affine. The limit of
a family of compatible nonempty quasicompact closed sets in the diagram is also nonempty.
-/
lemma exists_mem_of_isClosed_of_nonempty
    [IsCofilteredOrEmpty I]
    [∀ {i j} (f : i ⟶ j), IsAffineHom (D.map f)]
    (Z : ∀ (i : I), Set (D.obj i))
    (hZc : ∀ (i : I), IsClosed (Z i))
    (hZne : ∀ i, (Z i).Nonempty)
    (hZcpt : ∀ i, IsCompact (Z i))
    (hmapsTo : ∀ {i i' : I} (f : i ⟶ i'), Set.MapsTo (D.map f).base (Z i) (Z i')) :
    ∃ (s : c.pt), ∀ i, (c.π.app i).base s ∈ Z i := by
  let D' : I     Scheme :=
  { obj i := (vanishingIdeal ⟨Z i, hZc i⟩).subscheme
    map {X Y} f := subschemeMap _ _ (D.map f) (by
      rw [map_vanishingIdeal, ← le_support_iff_le_vanishingIdeal]
      simpa [(hZc _).closure_subset_iff] using (hmapsTo f).subset_preimage)
    map_id _ := by simp [← cancel_mono (subschemeι _)]
    map_comp _ _ := by simp [← cancel_mono (subschemeι _)] }
  let ι : D' ⟶ D := { app i := subschemeι _, naturality _ _ _ := by simp [D'] }
  haveI {i j} (f : i ⟶ j) : IsAffineHom (D'.map f) := by
    suffices IsAffineHom (D'.map f     ι.app j) from .of_comp _ (ι.app j)
    simp only [subschemeMap_subschemeι, D', ι]
    infer_instance
  haveI _ (i) : Nonempty (D'.obj i) := Set.nonempty_coe_sort.mpr (hZne i)
  haveI _ (i) : CompactSpace (D'.obj i) := isCompact_iff_compactSpace.mp (hZcpt i)
  let c' : Cone D' :=
  { pt := (    i, (vanishingIdeal ⟨Z i, hZc i⟩).comap (c.π.app i)).subscheme
    π :=
    { app i := subschemeMap _ _ (c.π.app i) (by simp [le_map_iff_comap_le, le_iSup_of_le i])
      naturality {i j} f := by simp [D', ← cancel_mono (subschemeι _)] } }
  let hc' : IsLimit c' :=
  { lift s := IsClosedImmersion.lift (subschemeι _) (hc.lift ((Cones.postcompose ι).obj s)) (by
      suffices ∀ i, vanishingIdeal ⟨Z i, hZc i⟩ ≤ (s.π.app i     ι.app i).ker by
        simpa [← le_map_iff_comap_le, ← Scheme.Hom.ker_comp]
      refine fun i     .trans ?_ (Scheme.Hom.le_ker_comp _ _)
      simp [ι])
    fac s i := by simp [← cancel_mono (subschemeι _), c', ι]
    uniq s m hm := by
      rw [← cancel_mono (subschemeι _)]
      refine hc.hom_ext fun i     ?_
      simp [ι, c', ← hm] }
  have : Nonempty (    i, (vanishingIdeal ⟨Z i, hZc i⟩).comap (c.π.app i)).support :=
    Scheme.nonempty_of_isLimit D' c' hc'
  simpa using this

include hc in
/--
A variant of `exists_mem_of_isClosed_of_nonempty` where the closed sets are only defined
for the objects over a given `j : I`.
-/
```

```
@[stacks 01Z3]
lemma exists_mem_of_isClosed_of_nonempty'
    [IsCofilteredOrEmpty I]
    [∀ {i j} (f : i ⟶ j), IsAffineHom (D.map f)]
    {j : I}
    (Z : ∀ (i : I), (i ⟶ j) → Set (D.obj i))
    (hZc : ∀ i hij, IsClosed (Z i hij))
    (hZne : ∀ i hij, (Z i hij).Nonempty)
    (hZcpt : ∀ i hij, IsCompact (Z i hij))
    (hstab : ∀ (i i' : I) (hi'i : i' ⟶ i) (hij : i ⟶ j),
      Set.MapsTo (D.map hi'i).base (Z i' (hi'i     hij)) (Z i hij)) :
    ∃ (s : c.pt), ∀ i hij, (c.π.app i).base s ∈ Z i hij := by
  have {i₁ i₂ : Over j} (f : i₁ ⟶ i₂) : IsAffineHom ((Over.forget j     D).map f) := by
    dsimp; infer_instance
  simpa [Over.forall_iff] using exists_mem_of_isClosed_of_nonempty (Over.forget j D) _
    ((Functor.Initial.isLimitWhiskerEquiv (Over.forget j) c).symm hc)
    (fun i     Z i.left i.hom) (fun _     hZc _ _)  (fun _     hZne _ _)  (fun _     hZcpt _ _)
    (fun {i₁ i₂} f     by dsimp; rw [← Over.w f]; exact hstab ..)
```

Listing 1: AffineTransitionLimit.lean