# Lean 4 Code: AffineSpace

Mathlib4

September 6, 2025

## 1 Source Code

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

```
/-
Copyright (c) 2024 Andrew Yang. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Andrew Yang
-/
import Mathlib.Algebra.MvPolynomial.Monad
import Mathlib.AlgebraicGeometry.Morphisms.Finite
import Mathlib.AlgebraicGeometry.Morphisms.FinitePresentation
import Mathlib.RingTheory.Spectrum.Prime.Polynomial
import Mathlib.AlgebraicGeometry.PullbackCarrier

/-!
# Affine space

## Main definitions

- `AlgebraicGeometry.AffineSpace`: `𝔸(n; S)` is the affine `n`-space over `S`.
- `AlgebraicGeometry.AffineSpace.coord`: The standard coordinate functions on the
    affine space.
- `AlgebraicGeometry.AffineSpace.homOfVector`:
  The morphism `X ⟶ 𝔸(n; S)` given by a `X ⟶ S` and a choice of `n`-coordinate
      functions.
- `AlgebraicGeometry.AffineSpace.homOverEquiv`:
  `S`-morphisms into `Spec 𝔸(n; S)` are equivalent to the choice of `n` global
      sections.
- `AlgebraicGeometry.AffineSpace.SpecIso`: `𝔸(n; Spec R) ≅ Spec R[n]`

-/

open CategoryTheory Limits MvPolynomial

noncomputable section

namespace AlgebraicGeometry

universe v u

variable (n : Type v) (S : Scheme.{max u v})

local notation3 "ℤ[" n "]" => CommRingCat.of (MvPolynomial n (ULift ℤ))
local notation3 "ℤ[" n "].{" u "," v "}" => CommRingCat.of (MvPolynomial n
    (ULift.{max u v} ℤ))

```

```
/-- `𝔸(n; S)` is the affine `n`-space over `S`.
Note that `n` is an arbitrary index type (e.g. `Fin m`). -/
def AffineSpace (n : Type v) (S : Scheme.{max u v}) : Scheme.{max u v} :=
  pullback (terminal.from S) (terminal.from (Spec ℤ[n].{u, v}))

namespace AffineSpace

/-- `𝔸(n; S)` is the affine `n`-space over `S`. -/
scoped [AlgebraicGeometry] notation "𝔸("n"; "S")" => AffineSpace n S

variable {n} in
lemma of_mvPolynomial_int_ext {R} {f g : ℤ[n] ⟶ R} (h : ∀ i, f (.X i) = g (.X i)) : f = g := by
  suffices f.hom.comp (MvPolynomial.mapEquiv _ ULift.ringEquiv.symm).toRingHom =
      g.hom.comp (MvPolynomial.mapEquiv _ ULift.ringEquiv.symm).toRingHom by
    ext x
    · obtain ⟨x⟩ := x
      simpa [-map_intCast, -eq_intCast] using DFunLike.congr_fun this (C x)
    · simpa [-map_intCast, -eq_intCast] using DFunLike.congr_fun this (X x)
  ext1
  · exact RingHom.ext_int _ _
  · simpa using h _


@[simps -isSimp]
instance over : 𝔸(n; S).CanonicallyOver S where
  hom := pullback.fst _ _

/-- The map from the affine `n`-space over `S` to the integral model `Spec ℤ[n]`. -/
def toSpecMvPoly : 𝔸(n; S) ⟶ Spec ℤ[n].{u, v} := pullback.snd _ _

variable {X : Scheme.{max u v}}

/--
Morphisms into `Spec ℤ[n]` are equivalent the choice of `n` global sections.
Use `homOverEquiv` instead.
-/
@[simps]
def toSpecMvPolyIntEquiv : (X ⟶ Spec ℤ[n]) ≃ (n → Γ(X, ⊤)) where
  toFun f i := f.appTop ((Scheme.ΓSpecIso ℤ[n]).inv (.X i))
  invFun v := X.toSpecΓ     Spec.map
    (CommRingCat.ofHom (MvPolynomial.eval₂Hom ((algebraMap ℤ _).comp
        ULift.ringEquiv.toRingHom) v))
  left_inv f := by
    apply (ΓSpec.adjunction.homEquiv _ _).symm.injective
    apply Quiver.Hom.unop_inj
    rw [Adjunction.homEquiv_symm_apply, Adjunction.homEquiv_symm_apply]
    simp only [Functor.rightOp_obj, Scheme.Γ_obj, Scheme.Spec_obj,
        algebraMap_int_eq,
      RingEquiv.toRingHom_eq_coe, TopologicalSpace.Opens.map_top,
          Functor.rightOp_map, op_comp,
      Scheme.Γ_map, unop_comp, Quiver.Hom.unop_op, Scheme.comp_app,
          Scheme.toSpecΓ_appTop,
      Scheme.ΓSpecIso_naturality, ΓSpec.adjunction_counit_app, Category.assoc,
      Iso.cancel_iso_inv_left, ← Iso.eq_inv_comp]
    apply of_mvPolynomial_int_ext
    intro i
    rw [ConcreteCategory.hom_ofHom, coe_eval₂Hom, eval₂_X]
```

```
    rfl
  right_inv v := by
    ext i
    simp only [algebraMap_int_eq, RingEquiv.toRingHom_eq_coe, TopologicalSpace.Opens.map_top,
      Scheme.comp_app, Scheme.toSpecΓ_appTop, Scheme.ΓSpecIso_naturality, CommRingCat.comp_apply,
      CommRingCat.coe_of]
    -- TODO: why does `simp` not apply this lemma?
    rw [CommRingCat.hom_inv_apply]
    simp

lemma toSpecMvPolyIntEquiv_comp {X Y : Scheme} (f : X ⟶ Y) (g : Y ⟶ Spec ℤ[n]) (i) :
    toSpecMvPolyIntEquiv n (f      g) i = f.appTop (toSpecMvPolyIntEquiv n g i) := rfl

variable {n} in
/-- The standard coordinates of `𝔸(n; S)`. -/
def coord (i : n) : Γ(𝔸(n; S), ⊤) := toSpecMvPolyIntEquiv _ (toSpecMvPoly n S) i

section homOfVector

variable {n S}

/-- The morphism `X ⟶ 𝔸(n; S)` given by a `X ⟶ S` and a choice of `n`-coordinate functions. -/
def homOfVector (f : X ⟶ S) (v : n → Γ(X, ⊤)) : X ⟶ 𝔸(n; S) :=
  pullback.lift f ((toSpecMvPolyIntEquiv n).symm v) (by simp)

variable (f : X ⟶ S) (v : n → Γ(X, ⊤))

@[reassoc (attr := simp)]
lemma homOfVector_over : homOfVector f v      𝔸(n; S)      S = f :=
  pullback.lift_fst _ _ _

@[reassoc]
lemma homOfVector_toSpecMvPoly :
    homOfVector f v      toSpecMvPoly n S = (toSpecMvPolyIntEquiv n).symm v :=
  pullback.lift_snd _ _ _

@[simp]
lemma homOfVector_appTop_coord (i) :
    (homOfVector f v).appTop (coord S i) = v i := by
  rw [coord, ← toSpecMvPolyIntEquiv_comp, homOfVector_toSpecMvPoly,
    Equiv.apply_symm_apply]

@[ext 1100]
lemma hom_ext {f g : X ⟶ 𝔸(n; S)}
    (h₁ : f      𝔸(n; S)      S = g      𝔸(n; S)      S)
    (h₂ : ∀ i, f.appTop (coord S i) = g.appTop (coord S i)) : f = g := by
  apply pullback.hom_ext h₁
  change f      toSpecMvPoly _ _ = g      toSpecMvPoly _ _
  apply (toSpecMvPolyIntEquiv n).injective
  ext i
  rw [toSpecMvPolyIntEquiv_comp, toSpecMvPolyIntEquiv_comp]
  exact h₂ i

@[reassoc]
```

```
lemma comp_homOfVector {X Y : Scheme} (v : n → Γ(Y, ⊤)) (f : X ⟶ Y) (g : Y ⟶ S) :
    f     homOfVector g v = homOfVector (f     g) (f.appTop ∘ v) := by
  ext1 <;> simp

end homOfVector

variable [X.Over S]

variable {n}

instance (v : n → Γ(X, ⊤)) : (homOfVector (X     S) v).IsOver S where

/-- `S`-morphisms into `Spec 𝔸(n; S)` are equivalent to the choice of `n` global sections. -/
@[simps]
def homOverEquiv : { f : X ⟶ 𝔸(n; S) // f.IsOver S } ≃ (n → Γ(X, ⊤)) where
  toFun f i := f.1.appTop (coord S i)
  invFun v := ⟨homOfVector (X     S) v, inferInstance⟩
  left_inv f := by
    ext : 2
    · simp [f.2.1]
    · rw [homOfVector_appTop_coord]
  right_inv v := by ext i; simp [-TopologicalSpace.Opens.map_top, homOfVector_appTop_coord]

variable (n) in
/--
The affine space over an affine base is isomorphic to the spectrum of the polynomial ring.
Also see `AffineSpace.SpecIso`.
-/
@[simps -isSimp hom inv]
def isoOfIsAffine [IsAffine S] :
    𝔸(n; S) ≅ Spec(MvPolynomial n Γ(S, ⊤)) where
      hom := 𝔸(n; S).toSpecΓ     Spec.map (CommRingCat.ofHom
        (eval₂Hom ((𝔸(n; S)     S).appTop).hom (coord S)))
      inv := homOfVector (Spec.map (CommRingCat.ofHom C)     S.isoSpec.inv)
        ((Scheme.ΓSpecIso (.of (MvPolynomial n Γ(S, ⊤)))).inv ∘ MvPolynomial.X)
      hom_inv_id := by
        ext1
        · simp only [Category.assoc, homOfVector_over, Category.id_comp]
          rw [← Spec.map_comp_assoc, ← CommRingCat.ofHom_comp, eval₂Hom_comp_C,
            CommRingCat.ofHom_hom, ← Scheme.toSpecΓ_naturality_assoc]
          simp [Scheme.isoSpec]
        · simp only [Category.assoc, Scheme.comp_app, Scheme.comp_coeBase,
            TopologicalSpace.Opens.map_comp_obj, TopologicalSpace.Opens.map_top,
            Scheme.toSpecΓ_appTop, Scheme.ΓSpecIso_naturality, CommRingCat.comp_apply,
            homOfVector_appTop_coord, Function.comp_apply, CommRingCat.coe_of, Scheme.id_app,
            CommRingCat.id_apply]
          -- TODO: why does `simp` not apply this?
          rw [CommRingCat.hom_inv_apply]
          exact eval₂_X _ _ _
      inv_hom_id := by
        apply ext_of_isAffine
        simp only [Scheme.comp_coeBase, TopologicalSpace.Opens.map_comp_obj,
          TopologicalSpace.Opens.map_top, Scheme.comp_app, Scheme.toSpecΓ_appTop,
```

```
          Scheme.ΓSpecIso_naturality, Category.assoc, Scheme.id_app, ← Iso.eq_inv_comp,
          Category.comp_id]
        ext : 1
        apply ringHom_ext'
        · change _ = (CommRingCat.ofHom C     _).hom
          rw [CommRingCat.hom_comp, RingHom.comp_assoc, CommRingCat.hom_ofHom, eval₂Hom_comp_C,
            ← CommRingCat.hom_comp, ← CommRingCat.hom_ext_iff,
            ← cancel_mono (Scheme.ΓSpecIso _).hom]
          rw [← Scheme.comp_appTop, homOfVector_over, Scheme.comp_appTop]
          simp only [Category.assoc, Scheme.ΓSpecIso_naturality, CommRingCat.of_carrier,
            ← Scheme.toSpecΓ_appTop]
          rw [← Scheme.comp_appTop_assoc, Scheme.isoSpec, asIso_inv, IsIso.hom_inv_id]
          simp
        · intro i
          rw [CommRingCat.comp_apply, ConcreteCategory.hom_ofHom, coe_eval₂Hom]
          simp only [eval₂_X]
          exact homOfVector_appTop_coord _ _ _

@[simp]
lemma isoOfIsAffine_hom_appTop [IsAffine S] :
    (isoOfIsAffine n S).hom.appTop =
      (Scheme.ΓSpecIso _).hom     CommRingCat.ofHom
        (eval₂Hom ((𝔸(n; S)     S).appTop).hom (coord S)) := by
  simp [isoOfIsAffine_hom]

@[simp]
lemma isoOfIsAffine_inv_appTop_coord [IsAffine S] (i) :
    (isoOfIsAffine n S).inv.appTop (coord _ i) = (Scheme.ΓSpecIso (.of _)).inv (.X i) :=
  homOfVector_appTop_coord _ _ _

@[reassoc (attr := simp)]
lemma isoOfIsAffine_inv_over [IsAffine S] :
    (isoOfIsAffine n S).inv     𝔸(n; S)     S = Spec.map (CommRingCat.ofHom C)     S.isoSpec.inv :=
  pullback.lift_fst _ _ _

instance [IsAffine S] : IsAffine 𝔸(n; S) := .of_isIso (isoOfIsAffine n S).hom

variable (n) in
/-- The affine space over an affine base is isomorphic to the spectrum of the polynomial ring. -/
def SpecIso (R : CommRingCat.{max u v}) :
    𝔸(n; Spec R) ≅ Spec(MvPolynomial n R) :=
  isoOfIsAffine _ _         Scheme.Spec.mapIso (MvPolynomial.mapEquiv _
    (Scheme.ΓSpecIso R).symm.commRingCatIsoToRingEquiv).toCommRingCatIso.op

@[simp]
lemma SpecIso_hom_appTop (R : CommRingCat.{max u v}) :
    (SpecIso n R).hom.appTop = (Scheme.ΓSpecIso _).hom
      CommRingCat.ofHom (eval₂Hom ((Scheme.ΓSpecIso _).inv
        (𝔸(n; Spec R)     Spec R).appTop).hom (coord (Spec R))) := by
  simp only [SpecIso, Iso.trans_hom, Functor.mapIso_hom, Iso.op_hom,
    Scheme.Spec_map, Quiver.Hom.unop_op, TopologicalSpace.Opens.map_top, Scheme.comp_app,
```

```
    isoOfIsAffine_hom_appTop, Scheme.ΓSpecIso_naturality_assoc]
  congr 1
  ext : 1
  apply ringHom_ext'
  · ext; simp
  · simp

@[simp]
lemma SpecIso_inv_appTop_coord (R : CommRingCat.{max u v}) (i) :
    (SpecIso n R).inv.appTop (coord _ i) = (Scheme.ΓSpecIso (.of _)).inv (.X i)
        := by
  simp only [SpecIso, Iso.trans_inv, Functor.mapIso_inv, Iso.op_inv,
      Scheme.Spec_map,
    Quiver.Hom.unop_op, TopologicalSpace.Opens.map_top, Scheme.comp_app,
        CommRingCat.comp_apply]
  rw [isoOfIsAffine_inv_appTop_coord, ← CommRingCat.comp_apply, ←
      Scheme.ΓSpecIso_inv_naturality,
      CommRingCat.comp_apply]
  congr 1
  exact map_X _ _

@[reassoc (attr := simp)]
lemma SpecIso_inv_over (R : CommRingCat.{max u v}) :
    (SpecIso n R).inv    𝔸(n; Spec R)    Spec R = Spec.map (CommRingCat.ofHom
        C) := by
  simp only [SpecIso, Iso.trans_inv, Functor.mapIso_inv, Iso.op_inv,
      Scheme.Spec_map,
    Quiver.Hom.unop_op, Category.assoc, isoOfIsAffine_inv_over,
        Scheme.isoSpec_Spec_inv,
    ← Spec.map_comp]
  congr 1
  rw [Iso.inv_comp_eq]
  ext : 2
  exact map_C _ _

section functorial

variable (n) in
/-- `𝔸(n; S)` is functorial w.r.t. `S`. -/
def map {S T : Scheme.{max u v}} (f : S ⟶ T) : 𝔸(n; S) ⟶ 𝔸(n; T) :=
  homOfVector (𝔸(n; S)    S    f) (coord S)

@[reassoc (attr := simp)]
lemma map_over {S T : Scheme.{max u v}} (f : S ⟶ T) : map n f    𝔸(n; T)    T
    = 𝔸(n; S)    S    f :=
  pullback.lift_fst _ _ _

@[simp]
lemma map_appTop_coord {S T : Scheme.{max u v}} (f : S ⟶ T) (i) :
    (map n f).appTop (coord T i) = coord S i :=
  homOfVector_appTop_coord _ _ _

@[reassoc (attr := simp)]
lemma map_toSpecMvPoly {S T : Scheme.{max u v}} (f : S ⟶ T) :
    map n f    toSpecMvPoly n T = toSpecMvPoly n S := by
  apply (toSpecMvPolyIntEquiv _).injective
  ext i
  rw [toSpecMvPolyIntEquiv_comp, ← coord, map_appTop_coord, coord]
```

```
@[simp]
lemma map_id : map n (      S) =       𝔸(n; S) := by
  ext1 <;> simp

@[reassoc, simp]
lemma map_comp {S S' S'' : Scheme} (f : S ⟶ S') (g : S' ⟶ S'') :
    map n (f      g) = map n f      map n g := by
  ext1
  · simp
  · simp

lemma map_Spec_map {R S : CommRingCat.{max u v}} (φ : R ⟶ S) :
    map n (Spec.map φ) =
      (SpecIso n S).hom      Spec.map (CommRingCat.ofHom (MvPolynomial.map φ.hom))

        (SpecIso n R).inv := by
  rw [← Iso.inv_comp_eq]
  ext1
  · simp only [map_over, Category.assoc, SpecIso_inv_over, SpecIso_inv_over_assoc,
      ← Spec.map_comp, ← CommRingCat.ofHom_comp]
    rw [map_comp_C, CommRingCat.ofHom_comp, CommRingCat.ofHom_hom]
  · simp only [TopologicalSpace.Opens.map_top, Scheme.comp_app, CommRingCat.comp_apply]
    conv_lhs => enter[2]; tactic => exact map_appTop_coord _ _
    conv_rhs => enter[2]; tactic => exact SpecIso_inv_appTop_coord _ _
    rw [SpecIso_inv_appTop_coord, ← CommRingCat.comp_apply, ← Scheme.ΓSpecIso_inv_naturality,
        CommRingCat.comp_apply, ConcreteCategory.hom_ofHom, map_X]

/-- The map between affine spaces over affine bases is
isomorphic to the natural map between polynomial rings. -/
def mapSpecMap {R S : CommRingCat.{max u v}} (φ : R ⟶ S) :
    Arrow.mk (map n (Spec.map φ)) ≅
      Arrow.mk (Spec.map (CommRingCat.ofHom (MvPolynomial.map (σ := n) φ.hom))) :=
  Arrow.isoMk (SpecIso n S) (SpecIso n R) (by have := (SpecIso n R).inv_hom_id; simp [map_Spec_map])

lemma isPullback_map {S T : Scheme.{max u v}} (f : S ⟶ T) :
    IsPullback (map n f) (𝔸(n; S)      S) (𝔸(n; T)      T) f := by
  refine (IsPullback.paste_horiz_iff (.flip <| .of_hasPullback _ _) (map_over f)).mp ?_
  simp only [terminal.comp_from, ]
  convert (IsPullback.of_hasPullback _ _).flip
  rw [← toSpecMvPoly, ← toSpecMvPoly, map_toSpecMvPoly]

/-- `𝔸(n; S)` is functorial w.r.t. `n`. -/
def reindex {n m : Type v} (i : m → n) (S : Scheme.{max u v}) : 𝔸(n; S) ⟶ 𝔸(m; S) :=
  homOfVector (𝔸(n; S)      S) (coord S ∘ i)

@[simp, reassoc]
lemma reindex_over {n m : Type v} (i : m → n) (S : Scheme.{max u v}) :
    reindex i S      𝔸(m; S)      S = 𝔸(n; S)      S :=
  pullback.lift_fst _ _ _

@[simp]
lemma reindex_appTop_coord {n m : Type v} (i : m → n) (S : Scheme.{max u v}) (j : m) :
    (reindex i S).appTop (coord S j) = coord S (i j) :=
```

```
  homOfVector_appTop_coord _ _ _

@[simp]
lemma reindex_id : reindex id S =      𝔸(n; S) := by
  ext1 <;> simp

@[simp, reassoc]
lemma reindex_comp {n₁ n₂ n₃ : Type v} (i : n₁ → n₂) (j : n₂ → n₃) (S : Scheme.{max u v}) :
    reindex (j ∘ i) S = reindex j S      reindex i S := by
  have H₁ : reindex (j ∘ i) S      𝔸(n₁; S)      S = (reindex j S      reindex i S)      𝔸(n₁; S)      S := by
    simp
  have H₂ (k) : (reindex (j ∘ i) S).appTop (coord S k) =
      (reindex j S).appTop ((reindex i S).appTop (coord S k)) := by
    rw [reindex_appTop_coord, reindex_appTop_coord, reindex_appTop_coord]
    rfl
  exact hom_ext H₁ H₂

@[reassoc (attr := simp)]
lemma map_reindex {n₁ n₂ : Type v} (i : n₁ → n₂) {S T : Scheme.{max u v}} (f : S ⟶ T) :
    map n₂ f      reindex i T = reindex i S      map n₁ f := by
  apply hom_ext <;> simp

/-- The affine space as a functor. -/
@[simps]
def functor : (Type v)           Scheme.{max u v}      Scheme.{max u v} where
  obj n := { obj := AffineSpace n.unop, map := map n.unop, map_id := map_id, map_comp := map_comp }
  map {n m} i := { app := reindex i.unop, naturality := fun _ _      map_reindex i.unop }
  map_id n := by ext: 2; exact reindex_id _
  map_comp f g := by ext: 2; dsimp; exact reindex_comp _ _ _

end functorial
section instances

instance : IsAffineHom (𝔸(n; S)      S) := MorphismProperty.pullback_fst _ _ inferInstance

instance : Surjective (𝔸(n; S)      S) := MorphismProperty.pullback_fst _ _ <| by
  have := isIso_of_isTerminal specULiftZIsTerminal terminalIsTerminal (terminal.from _)
  rw [← terminal.comp_from (Spec.map (CommRingCat.ofHom C)),
    MorphismProperty.cancel_right_of_respectsIso (P := @Surjective)]
  exact ⟨MvPolynomial.comap_C_surjective⟩

instance [Finite n] : LocallyOfFinitePresentation (𝔸(n; S)      S) :=
  MorphismProperty.pullback_fst _ _ <| by
  have := isIso_of_isTerminal specULiftZIsTerminal.{max u v} terminalIsTerminal (terminal.from _)
  rw [← terminal.comp_from (Spec.map (CommRingCat.ofHom C)),
    MorphismProperty.cancel_right_of_respectsIso (P := @LocallyOfFinitePresentation),
    HasRingHomProperty.Spec_iff (P := @LocallyOfFinitePresentation), RingHom.FinitePresentation]
  convert (inferInstanceAs (Algebra.FinitePresentation (ULift ℤ) ℤ[n]))
  exact Algebra.algebra_ext _ _ fun _      rfl
```

```
lemma isOpenMap_over : IsOpenMap (𝔸(n; S)     S).base := by
  change topologically @IsOpenMap _
  wlog hS : ∃ R, S = Spec R
  · refine (IsLocalAtTarget.iff_of_openCover (P := topologically @IsOpenMap)
      S.affineCover).mpr ?_
    intro i
    have := this (n := n) (S.affineCover.obj i) ⟨_, rfl⟩
    rwa [← (isPullback_map (n := n)  (S.affineCover.map i)).isoPullback_hom_snd,
      MorphismProperty.cancel_left_of_respectsIso (P := topologically
          @IsOpenMap)] at this
  obtain ⟨R, rfl⟩ := hS
  rw [← MorphismProperty.cancel_left_of_respectsIso (P := topologically
      @IsOpenMap)
    (SpecIso n R).inv, SpecIso_inv_over]
  exact MvPolynomial.isOpenMap_comap_C

open MorphismProperty in
instance [IsEmpty n] : IsIso (𝔸(n; S)     S) := pullback_fst
    (P := isomorphisms _) _ _ <| by
  rw [← terminal.comp_from (Spec.map (CommRingCat.ofHom C))]
  apply IsStableUnderComposition.comp_mem
  · rw [HasAffineProperty.iff_of_isAffine (P := isomorphisms _), ← isomorphisms,
      ← arrow_mk_iso_iff (isomorphisms _) (arrowIsoΓSpecOfIsAffine _)]
    exact ⟨inferInstance, (ConcreteCategory.isIso_iff_bijective _).mpr
      ⟨C_injective n _, C_surjective _⟩⟩
  · exact isIso_of_isTerminal specULiftZIsTerminal terminalIsTerminal
      (terminal.from _)

lemma isIntegralHom_over_iff_isEmpty : IsIntegralHom (𝔸(n; S)     S) ↔ IsEmpty S
    ∨ IsEmpty n := by
  constructor
  · intro h
    cases isEmpty_or_nonempty S
    · exact .inl    _
    refine .inr ?_
    wlog hS : ∃ R, S = Spec R
    · obtain ⟨x⟩ :=   Nonempty    S
      obtain ⟨y, hy⟩ := S.affineCover.covers x
      exact this (S.affineCover.obj x)
          (MorphismProperty.IsStableUnderBaseChange.of_isPullback
        (isPullback_map (S.affineCover.map x)) h) ⟨y⟩ ⟨_, rfl⟩
    obtain ⟨R, rfl⟩ := hS
    have : Nontrivial R := (subsingleton_or_nontrivial R).resolve_left fun H
        not_isEmpty_of_nonempty (Spec R) (inferInstanceAs (IsEmpty (PrimeSpectrum
          R)))
    constructor
    intro i
    have := RingHom.toMorphismProperty_respectsIso_iff.mp
        RingHom.isIntegral_respectsIso.{max u v}
    rw [← MorphismProperty.cancel_left_of_respectsIso @IsIntegralHom (SpecIso n
        R).inv,
      SpecIso_inv_over, HasAffineProperty.iff_of_isAffine (P := @IsIntegralHom)]
          at h
    obtain ⟨p : Polynomial R, hp, hp'⟩ :=
      (MorphismProperty.arrow_mk_iso_iff (RingHom.toMorphismProperty
          RingHom.IsIntegral)
        (arrowIsoΓSpecOfIsAffine _)).mpr h.2 (X i)
    have : (rename fun _     i).comp (pUnitAlgEquiv.{_, v} _).symm.toAlgHom p = 0
```

```
      := by
    simp [← hp', ← algebraMap_eq]
  rw [AlgHom.comp_apply, map_eq_zero_iff _ (rename_injective _ (fun _ _ _
      rfl))] at this
  simp only [AlgEquiv.toAlgHom_eq_coe, AlgHom.coe_coe,
      EmbeddingLike.map_eq_zero_iff] at this
  simp [this] at hp
· rintro (_ | _) <;> infer_instance

lemma not_isIntegralHom [Nonempty S] [Nonempty n] : ¬ IsIntegralHom (𝔸(n; S)
    S) := by
  simp [isIntegralHom_over_iff_isEmpty]

lemma spec_le_iff (R : CommRingCat) (p q : Spec R) : p ≤ q ↔ q.asIdeal ≤
    p.asIdeal := by
  aesop (add simp PrimeSpectrum.le_iff_specializes)

/--
One should bear this equality in mind when breaking the `Spec R/ PrimeSpectrum R`
    abstraction
boundary, since these instances are not definitionally equal.
-/
example (R : CommRingCat) :
    inferInstance (α := Preorder (Spec R)) = inferInstance (α := Preorder
        (PrimeSpectrum R)      ) := by
  aesop (add simp spec_le_iff)

end instances

end AffineSpace

end AlgebraicGeometry
```

Listing 1: AffineSpace.lean