Lean 4 Code: SpreadingOut

Mathlib4

September 6, 2025

1 Source Code

The following is the Lean 4 source code from `SpreadingOut.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.AlgebraicGeometry.Morphisms.FiniteType
import Mathlib.AlgebraicGeometry.Noetherian
import Mathlib.AlgebraicGeometry.Stalk
import Mathlib.AlgebraicGeometry.Properties

/-!
# Spreading out morphisms

Under certain conditions, a morphism on stalks `Spec  _ {X, x} ⟶ Spec  _ {Y, y}` can be spread
out into a neighborhood of `x`.

## Main result
Given `S`-schemes `X Y` and points `x : X` `y : Y` over `s : S`.
Suppose we have the following diagram of `S`-schemes
```
Spec _ {X, x} ⟶ X
    |
  Spec(φ)

Spec _ {Y, y} ⟶ Y
```
We would like to spread `Spec(φ)` out to an `S`-morphism on an open subscheme `U ⊆ X`
```
Spec _ {X, x} ⟶ U ⊆ X
    | |
  Spec(φ) |

Spec _ {Y, y} ⟶ Y
```
- `AlgebraicGeometry.spread_out_unique_of_isGermInjective`:
  The lift is "unique" if the germ map is injective at `x`.
- `AlgebraicGeometry.spread_out_of_isGermInjective`:
  The lift exists if `Y` is locally of finite type and the germ map is injective at `x`.

## TODO
```

```
Show that certain morphism properties can also be spread out.

-/

universe u

open CategoryTheory

namespace AlgebraicGeometry

variable {X Y S : Scheme.{u}} (f : X ⟶ Y) (sX : X ⟶ S) (sY : Y ⟶ S) {R A : CommRingCat.{u}}

/-- The germ map at `x` is injective if there exists some affine `U    x`
  such that the map `Γ(X, U) ⟶ X_x` is injective -/
class Scheme.IsGermInjectiveAt (X : Scheme.{u}) (x : X) : Prop where
  cond : ∃ (U : X.Opens) (hx : x ∈ U), IsAffineOpen U ∧ Function.Injective (X.presheaf.germ U x hx)

lemma injective_germ_basicOpen (U : X.Opens) (hU : IsAffineOpen U)
    (x : X) (hx : x ∈ U) (f : Γ(X, U))
    (hf : x ∈ X.basicOpen f)
    (H : Function.Injective (X.presheaf.germ U x hx)) :
    Function.Injective (X.presheaf.germ (X.basicOpen f) x hf) := by
  rw [RingHom.injective_iff_ker_eq_bot, RingHom.ker_eq_bot_iff_eq_zero] at H
  intro t ht
  have := hU.isLocalization_basicOpen f
  obtain ⟨t, s, rfl⟩ := IsLocalization.mk'_surjective (.powers f) t
  rw [← RingHom.mem_ker, IsLocalization.mk'_eq_mul_mk'_one, Ideal.mul_unit_mem_iff_mem,
    RingHom.mem_ker, RingHom.algebraMap_toAlgebra, TopCat.Presheaf.germ_res_apply] at ht
  swap; · exact @isUnit_of_invertible _ _ _ (@IsLocalization.invertible_mk'_one ..)
  rw [H _ ht, IsLocalization.mk'_zero]

lemma Scheme.exists_germ_injective (X : Scheme.{u}) (x : X) [X.IsGermInjectiveAt x] :
    ∃ (U : X.Opens) (hx : x ∈ U),
      IsAffineOpen U ∧ Function.Injective (X.presheaf.germ U x hx) :=
  Scheme.IsGermInjectiveAt.cond

lemma Scheme.exists_le_and_germ_injective (X : Scheme.{u}) (x : X) [X.IsGermInjectiveAt x]
    (V : X.Opens) (hxV : x ∈ V) :
    ∃ (U : X.Opens) (hx : x ∈ U),
      IsAffineOpen U ∧ U ≤ V ∧ Function.Injective (X.presheaf.germ U x hx) := by
  obtain ⟨U, hx, hU, H⟩ := Scheme.IsGermInjectiveAt.cond (x := x)
  obtain ⟨f, hf, hxf⟩ := hU.exists_basicOpen_le ⟨x, hxV⟩ hx
  exact ⟨X.basicOpen f, hxf, hU.basicOpen f, hf, injective_germ_basicOpen U hU x hx f hxf H⟩

instance (x : X) [X.IsGermInjectiveAt x] [IsOpenImmersion f] :
    Y.IsGermInjectiveAt (f.base x) := by
  obtain ⟨U, hxU, hU, H⟩ := X.exists_germ_injective x
  refine ⟨⟨f ''    U, ⟨x, hxU, rfl⟩, hU.image_of_isOpenImmersion f, ?_⟩⟩
  refine ((MorphismProperty.injective CommRingCat).cancel_right_of_respectsIso _
    (f.stalkMap x)).mp ?_
```

```
  refine ((MorphismProperty.injective CommRingCat).cancel_left_of_respectsIso
    (f.appIso U).inv _).mp ?_
  simpa

variable {f} in
lemma isGermInjectiveAt_iff_of_isOpenImmersion {x : X} [IsOpenImmersion f] :
    Y.IsGermInjectiveAt (f.base x) ↔ X.IsGermInjectiveAt x := by
  refine ⟨fun H     ?_, fun _     inferInstance⟩
  obtain ⟨U, hxU, hU, hU', H⟩ :=
    Y.exists_le_and_germ_injective (f.base x) (V := f.opensRange) ⟨x, rfl⟩
  obtain ⟨V, hV⟩ := (IsOpenImmersion.affineOpensEquiv f).surjective ⟨⟨U, hU⟩, hU'⟩
  obtain rfl : f ''    V = U := Subtype.eq_iff.mp (Subtype.eq_iff.mp hV)
  obtain ⟨y, hy, e : f.base y = f.base x⟩ := hxU
  obtain rfl := f.isOpenEmbedding.injective e
  refine ⟨V, hy, V.2, ?_⟩
  replace H := ((MorphismProperty.injective CommRingCat).cancel_right_of_respectsIso _
    (f.stalkMap y)).mpr H
  replace H := ((MorphismProperty.injective CommRingCat).cancel_left_of_respectsIso
    (f.appIso V).inv _).mpr H
  simpa using H

/--
The class of schemes such that for each `x : X`,
`Γ(X, U) ⟶ X_x` is injective for some affine `U` containing `x`.

This is typically satisfied when `X` is integral or locally Noetherian.
-/
abbrev Scheme.IsGermInjective (X : Scheme.{u}) := ∀ x : X, X.IsGermInjectiveAt x

lemma Scheme.IsGermInjective.of_openCover
    {X : Scheme.{u}} (𝒰 : X.OpenCover) [∀ i, (𝒰.obj i).IsGermInjective] : X.IsGermInjective := by
  intro x
  rw [← (𝒰.covers x).choose_spec]
  infer_instance

protected
lemma Scheme.IsGermInjective.Spec
    (H : ∀ I : Ideal R, I.IsPrime →
      ∃ f : R, f ∉ I ∧ ∀ (x y : R), y * x = 0 → y ∉ I → ∃ n, f ^ n * x = 0) :
    (Spec R).IsGermInjective := by
  refine fun p     ⟨?_⟩
  obtain ⟨f, hf, H⟩ := H p.asIdeal p.2
  refine ⟨PrimeSpectrum.basicOpen f, hf, ?_, ?_⟩
  · rw [← basicOpen_eq_of_affine]
    exact (isAffineOpen_top (Spec R)).basicOpen _
  rw [RingHom.injective_iff_ker_eq_bot, RingHom.ker_eq_bot_iff_eq_zero]
  intro x hx
  obtain ⟨x, s, rfl⟩ := IsLocalization.mk'_surjective
    (S := ((Spec.structureSheaf R).val.obj (.op <| PrimeSpectrum.basicOpen f))) (.powers f) x
  rw [← RingHom.mem_ker, IsLocalization.mk'_eq_mul_mk'_one, Ideal.mul_unit_mem_iff_mem,
    RingHom.mem_ker, RingHom.algebraMap_toAlgebra] at hx
  swap; · exact @isUnit_of_invertible _ _ _ (@IsLocalization.invertible_mk'_one ..)
  -- There is an `Opposite.unop (Opposite.op _)` in `hx` which doesn't seem
```

```
      removable using
  -- `simp`/`rw`.
  erw [StructureSheaf.germ_toOpen] at hx
  obtain ⟨⟨y, hy⟩, hy'⟩ := (IsLocalization.map_eq_zero_iff p.asIdeal.primeCompl
    ((Spec.structureSheaf R).presheaf.stalk p) _).mp hx
  obtain ⟨n, hn⟩ := H x y hy' hy
  refine (@IsLocalization.mk'_eq_zero_iff ..).mpr ?_
  exact ⟨⟨_, n, rfl⟩, hn⟩

instance (priority := 100) [IsIntegral X] : X.IsGermInjective := by
  refine fun x     ⟨⟨(X.affineCover.map x).opensRange, X.affineCover.covers x,
    (isAffineOpen_opensRange (X.affineCover.map x)), ?_⟩⟩
  have : Nonempty (X.affineCover.map x).opensRange := ⟨⟨_, X.affineCover.covers x⟩⟩
  have := (isAffineOpen_opensRange (X.affineCover.map x)).isLocalization_stalk
    ⟨_, X.affineCover.covers x⟩
  exact @IsLocalization.injective _ _ _ _ _ (show _ from _) this
    (Ideal.primeCompl_le_nonZeroDivisors _)

instance (priority := 100) [IsLocallyNoetherian X] : X.IsGermInjective := by
  suffices ∀ (R : CommRingCat.{u}) (_ : IsNoetherianRing R), (Spec R).IsGermInjective by
    refine @Scheme.IsGermInjective.of_openCover _ (X.affineOpenCover.openCover) (fun i     this _ ?_)
    have := isLocallyNoetherian_of_isOpenImmersion (X.affineOpenCover.map i)
    infer_instance
  refine fun R hR     Scheme.IsGermInjective.Spec fun I hI     ?_
  let J := RingHom.ker <| algebraMap R (Localization.AtPrime I)
  have hJ (x) : x ∈ J ↔ ∃ y : I.primeCompl, y * x = 0 :=
    IsLocalization.map_eq_zero_iff I.primeCompl _ x
  choose f hf using fun x     (hJ x).mp
  obtain ⟨s, hs⟩ := (isNoetherianRing_iff_ideal_fg R).mp    _    J
  have hs' : (s : Set R) ⊆ J := hs ▷ Ideal.subset_span
  refine ⟨_, (s.attach.prod fun x     f x (hs' x.2)).2, fun x y e hy     ⟨1, ?_⟩⟩
  rw [pow_one, mul_comm, ← smul_eq_mul, ← Submodule.mem_annihilator_span_singleton]
  refine SetLike.le_def.mp ?_ ((hJ x).mpr ⟨⟨y, hy⟩, e⟩)
  rw [← hs, Ideal.span_le]
  intro i hi
  rw [SetLike.mem_coe, Submodule.mem_annihilator_span_singleton, smul_eq_mul,
    mul_comm, ← smul_eq_mul, ← Submodule.mem_annihilator_span_singleton, Submonoid.coe_finset_prod]
  refine Ideal.mem_of_dvd _ (Finset.dvd_prod_of_mem _ (s.mem_attach ⟨i, hi⟩)) ?_
  rw [Submodule.mem_annihilator_span_singleton, smul_eq_mul]
  exact hf i _

/--
Let `x : X` and `f g : X ⟶ Y` be two morphisms such that `f x = g x`.
If `f` and `g` agree on the stalk of `x`, then they agree on an open neighborhood of `x`,
provided `X` is "germ-injective" at `x` (e.g. when it's integral or locally Noetherian).

TODO: The condition on `X` is unnecessary when `Y` is locally of finite type.
-/
@[stacks 0BX6]
lemma spread_out_unique_of_isGermInjective {x : X} [X.IsGermInjectiveAt x]
    (f g : X ⟶ Y) (e : f.base x = g.base x)
    (H : f.stalkMap x =
```

```
      Y.presheaf.stalkSpecializes (Inseparable.of_eq e.symm).specializes g.stalkMap x) :
    ∃ (U : X.Opens), x ∈ U ∧ U.ι      f = U.ι      g := by
  obtain ⟨_, ⟨V : Y.Opens, hV, rfl⟩, hxV, -⟩ :=
    (isBasis_affine_open Y).exists_subset_of_mem_open (Set.mem_univ (f.base x)) isOpen_univ
  have hxV' : g.base x ∈ V := e ▷ hxV
  obtain ⟨U, hxU, _, hUV, HU⟩ := X.exists_le_and_germ_injective x (f ⁻¹     V   g ⁻¹     V) ⟨hxV, hxV'⟩
  refine ⟨U, hxU, ?_⟩
  rw [← Scheme.Hom.resLE_comp_ι _ (hUV.trans inf_le_left),
    ← Scheme.Hom.resLE_comp_ι _ (hUV.trans inf_le_right)]
  congr 1
  have : IsAffine V := hV
  suffices ∀ (U₀ V₀) (eU : U = U₀) (eV : V = V₀),
      f.appLE V₀ U₀ (eU ▷ eV ▷ hUV.trans inf_le_left) =
        g.appLE V₀ U₀ (eU ▷ eV ▷ hUV.trans inf_le_right) by
    rw [← cancel_mono V.toScheme.isoSpec.hom]
    simp only [Scheme.isoSpec, asIso_hom, Scheme.toSpecΓ_naturality,
      Scheme.Hom.app_eq_appLE, Scheme.Hom.resLE_appLE]
    congr 2
    apply this <;> simp
  rintro U V rfl rfl
  have := ConcreteCategory.mono_of_injective _ HU
  rw [← cancel_mono (X.presheaf.germ U x hxU)]
  simp only [Scheme.Hom.appLE, Category.assoc, X.presheaf.germ_res', ← Scheme.stalkMap_germ, H]
  simp only [TopCat.Presheaf.germ_stalkSpecializes_assoc, Scheme.stalkMap_germ]

/--
A variant of `spread_out_unique_of_isGermInjective`
whose condition is an equality of scheme morphisms instead of ring homomorphisms.
-/
lemma spread_out_unique_of_isGermInjective' {x : X} [X.IsGermInjectiveAt x]
    (f g : X ⟶ Y)
    (e : X.fromSpecStalk x      f = X.fromSpecStalk x      g) :
    ∃ (U : X.Opens), x ∈ U ∧ U.ι      f = U.ι      g := by
  fapply spread_out_unique_of_isGermInjective
  · simpa using congr(($e).base (IsLocalRing.closedPoint _))
  · apply Spec.map_injective
    rw [← cancel_mono (Y.fromSpecStalk _)]
    simpa [Scheme.Spec_map_stalkSpecializes_fromSpecStalk]

lemma exists_lift_of_germInjective_aux {U : X.Opens} {x : X} (hxU)
    (φ : A ⟶ X.presheaf.stalk x) (φRA : R ⟶ A) (φRX : R ⟶ Γ(X, U))
    (hφRA : RingHom.FiniteType φRA.hom)
    (e : φRA      φ = φRX      X.presheaf.germ U x hxU) :
    ∃ (V : X.Opens) (hxV : x ∈ V),
      V ≤ U ∧ RingHom.range φ.hom ≤ RingHom.range (X.presheaf.germ V x hxV).hom := by
  letI := φRA.hom.toAlgebra
  obtain ⟨s, hs⟩ := hφRA
  choose W hxW f hf using fun t      X.presheaf.germ_exist x (φ t)
  have H : x ∈ s.inf W      U := by
    rw [← SetLike.mem_coe, TopologicalSpace.Opens.coe_inf,
      TopologicalSpace.Opens.coe_finset_inf]
    exact ⟨by simpa using fun x _      hxW x, hxU⟩
  refine ⟨s.inf W      U, H, inf_le_right, ?_⟩
  letI := φRX.hom.toAlgebra
```

```
  letI := (φRX      X.presheaf.germ U x hxU).hom.toAlgebra
  letI := (φRX      X.presheaf.map (homOfLE (inf_le_right (a := s.inf W))).op).hom.toAlgebra
  let φ' : A →    [R] X.presheaf.stalk x :=
    { φ.hom with commutes' := DFunLike.congr_fun (congr_arg CommRingCat.Hom.hom e) }
  let ψ : Γ(X, s.inf W      U) →    [R] X.presheaf.stalk x :=
    { (X.presheaf.germ _ x H).hom with commutes' := fun x X.presheaf.germ_res_apply _ _ _ _ }
  change AlgHom.range φ' ≤ AlgHom.range ψ
  rw [← Algebra.map_top, ← hs, AlgHom.map_adjoin, Algebra.adjoin_le_iff]
  rintro _ ⟨i, hi, rfl : φ i = _⟩
  refine ⟨X.presheaf.map (homOfLE (inf_le_left.trans (Finset.inf_le hi))).op (f i), ?_⟩
  exact (X.presheaf.germ_res_apply _ _ _ _).trans (hf _)

/--
Suppose `X` is a scheme, `x : X` such that the germ map at `x` is (locally) injective,
and `U` is a neighborhood of `x`.
Given a commutative diagram of `CommRingCat`
```
R ⟶ Γ(X, U)

A ⟶   _ {X, x}
```
such that `R` is of finite type over `A`, we may lift `A ⟶    _  {X, x}` to some `A ⟶ Γ(X, V)`.
-/
lemma exists_lift_of_germInjective {x : X} [X.IsGermInjectiveAt x] {U : X.Opens} (hxU : x ∈ U)
    (φ : A ⟶ X.presheaf.stalk x) (φRA : R ⟶ A) (φRX : R ⟶ Γ(X, U))
    (hφRA : RingHom.FiniteType φRA.hom)
    (e : φRA      φ = φRX      X.presheaf.germ U x hxU) :
    ∃ (V : X.Opens) (hxV : x ∈ V) (φ' : A ⟶ Γ(X, V)) (i : V ≤ U), IsAffineOpen V ∧
      φ = φ'      X.presheaf.germ V x hxV ∧ φRX      X.presheaf.map i.hom.op = φRA      φ' := by
  obtain ⟨V, hxV, iVU, hV⟩ := exists_lift_of_germInjective_aux hxU φ φRA φRX hφRA e
  obtain ⟨V', hxV', hV', iV'V, H⟩ := X.exists_le_and_germ_injective x V hxV
  let f := X.presheaf.germ V' x hxV'
  have hf' : RingHom.range (X.presheaf.germ V x hxV).hom ≤ RingHom.range f.hom := by
    rw [← X.presheaf.germ_res iV'V.hom _ hxV']
    exact Set.range_comp_subset_range (X.presheaf.map iV'V.hom.op) f
  let e := RingEquiv.ofLeftInverse H.hasLeftInverse.choose_spec
  refine ⟨V', hxV', CommRingCat.ofHom (e.symm.toRingHom.comp
    (φ.hom.codRestrict _ (fun x      hf' (hV ⟨x, rfl⟩)))), iV'V.trans iVU, hV', ?_, ?_⟩
  · ext a
    change φ a = (e (e.symm _)).1
    simp only [RingEquiv.apply_symm_apply]
    rfl
  · ext a
    apply e.injective
    change e _ = e (e.symm _)
    rw [RingEquiv.apply_symm_apply]
    ext
```

```
    change X.presheaf.germ _ _ _ (X.presheaf.map _ _) = (φRA     φ) a
    rw [TopCat.Presheaf.germ_res_apply,    φRA     φ = _  ]
    rfl

/--
Given ‘S‘-schemes ‘X Y‘ and points ‘x : X‘ ‘y : Y‘ over ‘s : S‘.
Suppose we have the following diagram of ‘S‘-schemes
‘‘‘
Spec   _  {X, x} ⟶ X
    |
  Spec(φ)

Spec   _  {Y, y} ⟶ Y
‘‘‘
Then the map ‘Spec(φ)‘ spreads out to an ‘S‘-morphism on an open subscheme ‘U ⊆ X‘,
‘‘‘
Spec   _  {X, x} ⟶ U ⊆ X
    |             |
  Spec(φ)         |

Spec   _  {Y, y} ⟶ Y
‘‘‘
provided that ‘Y‘ is locally of finite type over ‘S‘ and
‘X‘ is "germ-injective" at ‘x‘ (e.g. when it’s integral or locally Noetherian).

TODO: The condition on ‘X‘ is unnecessary when ‘Y‘ is locally of finite presentation.
-/
@[stacks 0BX6]
lemma spread_out_of_isGermInjective [LocallyOfFiniteType sY] {x : X}
    [X.IsGermInjectiveAt x] {y : Y}
    (e : sX.base x = sY.base y) (φ : Y.presheaf.stalk y ⟶ X.presheaf.stalk x)
    (h : sY.stalkMap y     φ =
      S.presheaf.stalkSpecializes (Inseparable.of_eq e).specializes
          sX.stalkMap x) :
    ∃ (U : X.Opens) (hxU : x ∈ U) (f : U.toScheme ⟶ Y),
      Spec.map φ     Y.fromSpecStalk y = U.fromSpecStalkOfMem x hxU     f ∧
      f     sY = U.ι     sX := by
  obtain ⟨_, ⟨U, hU, rfl⟩, hxU, -⟩ :=
    (isBasis_affine_open S).exists_subset_of_mem_open (Set.mem_univ (sX.base x))
        isOpen_univ
  have hyU : sY.base y ∈ U := e ▷ hxU
  obtain ⟨_, ⟨V : Y.Opens, hV, rfl⟩, hyV, iVU⟩ :=
    (isBasis_affine_open Y).exists_subset_of_mem_open hyU (sY ⁻¹    U).2
  have : sY.appLE U V iVU     Y.presheaf.germ V y hyV     φ =
      sX.app U     X.presheaf.germ (sX ⁻¹    U) x hxU := by
    rw [Scheme.Hom.appLE, Category.assoc, Y.presheaf.germ_res_assoc,
      ← Scheme.stalkMap_germ_assoc, h]
    simp
  obtain ⟨W, hxW, φ', i, hW, h₁, h₂⟩ :=
    exists_lift_of_germInjective (R := Γ(S, U)) (A := Γ(Y, V)) (U := sX ⁻¹    U)
        (x := x) hxU
    (Y.presheaf.germ _ y hyV     φ) (sY.appLE U V iVU) (sX.app U)
    (LocallyOfFiniteType.finiteType_of_affine_subset ⟨_, hU⟩ ⟨_, hV⟩ _) this
  refine ⟨W, hxW, W.toSpecΓ     Spec.map φ'     hV.fromSpec, ?_, ?_⟩
  · rw [W.fromSpecStalkOfMem_toSpecΓ_assoc x hxW, ← Spec.map_comp_assoc, ← h₁,
      Spec.map_comp, Category.assoc, ← IsAffineOpen.fromSpecStalk,
      IsAffineOpen.fromSpecStalk_eq_fromSpecStalk]
```

```
  · simp only [Category.assoc]
    rw [← IsAffineOpen.Spec_map_appLE_fromSpec sY hU hV iVU, ←
        Spec.map_comp_assoc, ← h₂,
      ← Scheme.Hom.appLE, ← hW.isoSpec_hom,
          IsAffineOpen.Spec_map_appLE_fromSpec sX hU hW i,
      ← Iso.eq_inv_comp, IsAffineOpen.isoSpec_inv_ι_assoc]

/--
Given `S`-schemes `X Y`, a point `x : X`, and a `S`-morphism `φ : Spec   _  {X,
    x} ⟶ Y`,
we may spread it out to an `S`-morphism `f : U ⟶ Y`
provided that `Y` is locally of finite type over `S` and
`X` is "germ-injective" at `x` (e.g. when it's integral or locally Noetherian).

TODO: The condition on `X` is unnecessary when `Y` is locally of finite
    presentation.
-/
lemma spread_out_of_isGermInjective' [LocallyOfFiniteType sY] {x : X}
    [X.IsGermInjectiveAt x]
    (φ : Spec (X.presheaf.stalk x) ⟶ Y)
    (h : φ      sY = X.fromSpecStalk x      sX) :
    ∃ (U : X.Opens) (hxU : x ∈ U) (f : U.toScheme ⟶ Y),
      φ = U.fromSpecStalkOfMem x hxU      f ∧ f      sY = U.ι      sX := by
  have := spread_out_of_isGermInjective sX sY ?_ (Scheme.stalkClosedPointTo φ) ?_
  · simpa only [Scheme.Spec_stalkClosedPointTo_fromSpecStalk] using this
  · rw [← Scheme.comp_base_apply, h, Scheme.comp_base_apply,
      Scheme.fromSpecStalk_closedPoint]
  · apply Spec.map_injective
    rw [← cancel_mono (S.fromSpecStalk _)]
    simpa only [Spec.map_comp, Category.assoc,
        Scheme.Spec_map_stalkMap_fromSpecStalk,
      Scheme.Spec_stalkClosedPointTo_fromSpecStalk_assoc,
      Scheme.Spec_map_stalkSpecializes_fromSpecStalk]

end AlgebraicGeometry
```

Listing 1: SpreadingOut.lean