Lean 4 Code: RationalMap

Mathlib4

September 6, 2025

1 Source Code

The following is the Lean 4 source code from `RationalMap.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.SpreadingOut
import Mathlib.AlgebraicGeometry.FunctionField
import Mathlib.AlgebraicGeometry.Morphisms.Separated
/-!

# Rational maps between schemes

## Main definitions

* `AlgebraicGeometry.Scheme.PartialMap`: A partial map from `X` to `Y` (`X.PartialMap Y`) is
  a morphism into `Y` defined on a dense open subscheme of `X`.
* `AlgebraicGeometry.Scheme.PartialMap.equiv`:
  Two partial maps are equivalent if they are equal on a dense open subscheme.
* `AlgebraicGeometry.Scheme.RationalMap`:
  A rational map from `X` to `Y` (`X    Y`) is an equivalence class of partial maps.
* `AlgebraicGeometry.Scheme.RationalMap.equivFunctionFieldOver`:
  Given `S`-schemes `X` and `Y` such that `Y` is locally of finite type and `X` is integral,
  `S`-morphisms `Spec K(X) ⟶ Y` correspond bijectively to `S`-rational maps from `X` to `Y`.
* `AlgebraicGeometry.Scheme.RationalMap.toPartialMap`:
  If `X` is integral and `Y` is separated, then any `f : X    Y` can be realized as a partial
  map on `f.domain`, the domain of definition of `f`.
-/

universe u

open CategoryTheory hiding Quotient

namespace AlgebraicGeometry

variable {X Y Z S : Scheme.{u}} (sX : X ⟶ S) (sY : Y ⟶ S)

namespace Scheme

```

```
/--
A partial map from `X` to `Y` (`X.PartialMap Y`) is a morphism into `Y`
defined on a dense open subscheme of `X`.
-/
structure PartialMap (X Y : Scheme.{u}) where
  /-- The domain of definition of a partial map. -/
  domain : X.Opens
  dense_domain : Dense (domain : Set X)
  /-- The underlying morphism of a partial map. -/
  hom :   domain   ⟶ Y

variable (S) in
/-- A partial map is a `S`-map if the underlying morphism is. -/
abbrev PartialMap.IsOver [X.Over S] [Y.Over S] (f : X.PartialMap Y) :=
  f.hom.IsOver S

namespace PartialMap

lemma ext_iff (f g : X.PartialMap Y) :
    f = g ↔ ∃ e : f.domain = g.domain, f.hom = (X.isoOfEq e).hom     g.hom := by
  constructor
  · rintro rfl
    simp only [exists_true_left, Scheme.isoOfEq_rfl, Iso.refl_hom,
        Category.id_comp]
  · obtain ⟨U, hU, f⟩ := f
    obtain ⟨V, hV, g⟩ := g
    rintro ⟨rfl : U = V, e⟩
    congr 1
    simpa using e

@[ext]
lemma ext (f g : X.PartialMap Y) (e : f.domain = g.domain)
    (H : f.hom = (X.isoOfEq e).hom     g.hom) : f = g := by
  rw [ext_iff]
  exact ⟨e, H⟩

/-- The restriction of a partial map to a smaller domain. -/
@[simps hom domain]
noncomputable
def restrict (f : X.PartialMap Y) (U : X.Opens)
    (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) : X.PartialMap Y where
  domain := U
  dense_domain := hU
  hom := X.homOfLE hU'     f.hom

@[simp]
lemma restrict_id (f : X.PartialMap Y) : f.restrict f.domain f.dense_domain
   le_rfl = f := by
  ext1 <;> simp [restrict_domain]

lemma restrict_id_hom (f : X.PartialMap Y) :
    (f.restrict f.domain f.dense_domain le_rfl).hom = f.hom := by
  simp

@[simp]
lemma restrict_restrict (f : X.PartialMap Y)
    (U : X.Opens) (hU : Dense (U : Set X)) (hU' : U ≤ f.domain)
    (V : X.Opens) (hV : Dense (V : Set X)) (hV' : V ≤ U) :
    (f.restrict U hU hU').restrict V hV hV' = f.restrict V hV (hV'.trans hU') :=
```

```
      by
  ext1 <;> simp [restrict_domain]

lemma restrict_restrict_hom (f : X.PartialMap Y)
    (U : X.Opens) (hU : Dense (U : Set X)) (hU' : U ≤ f.domain)
    (V : X.Opens) (hV : Dense (V : Set X)) (hV' : V ≤ U) :
    ((f.restrict U hU hU').restrict V hV hV').hom = (f.restrict V hV (hV'.trans
      hU')).hom := by
  simp

instance [X.Over S] [Y.Over S] (f : X.PartialMap Y) [f.IsOver S]
    (U : X.Opens) (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) :
    (f.restrict U hU hU').IsOver S where

/-- The composition of a partial map and a morphism on the right. -/
@[simps]
def compHom (f : X.PartialMap Y) (g : Y ⟶ Z) : X.PartialMap Z where
  domain := f.domain
  dense_domain := f.dense_domain
  hom := f.hom      g

instance [X.Over S] [Y.Over S] [Z.Over S] (f : X.PartialMap Y) (g : Y ⟶ Z)
    [f.IsOver S] [g.IsOver S] : (f.compHom g).IsOver S where

/-- A scheme morphism as a partial map. -/
@[simps]
def _root_.AlgebraicGeometry.Scheme.Hom.toPartialMap (f : X.Hom Y) :
    X.PartialMap Y := ⟨⊤, dense_univ, X.topIso.hom      f⟩

instance [X.Over S] [Y.Over S] (f : X ⟶ Y) [f.IsOver S] : f.toPartialMap.IsOver
   S where

lemma isOver_iff [X.Over S] [Y.Over S] {f : X.PartialMap Y} :
    f.IsOver S ↔ (f.compHom (Y      S)).hom = f.domain.ι      X      S := by
  simp

lemma isOver_iff_eq_restrict [X.Over S] [Y.Over S] {f : X.PartialMap Y} :
    f.IsOver S ↔ f.compHom (Y      S) = (X      S).toPartialMap.restrict _
      f.dense_domain (by simp) := by
  simp [PartialMap.ext_iff]

/-- If `x` is in the domain of a partial map `f`, then `f` restricts to a map
   from `Spec    _x   `. -/
noncomputable
def fromSpecStalkOfMem (f : X.PartialMap Y) {x} (hx : x ∈ f.domain) :
    Spec (X.presheaf.stalk x) ⟶ Y :=
  f.domain.fromSpecStalkOfMem x hx      f.hom

/-- A partial map restricts to a map from `Spec K(X)`. -/
noncomputable
abbrev fromFunctionField [IrreducibleSpace X] (f : X.PartialMap Y) :
    Spec X.functionField ⟶ Y :=
  f.fromSpecStalkOfMem
    ((genericPoint_specializes _).mem_open f.domain.2
      f.dense_domain.nonempty.choose_spec)

lemma fromSpecStalkOfMem_restrict (f : X.PartialMap Y)
    {U : X.Opens} (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) {x} (hx : x ∈ U) :
    (f.restrict U hU hU').fromSpecStalkOfMem hx = f.fromSpecStalkOfMem (hU' hx)
```

```
      := by
  dsimp only [fromSpecStalkOfMem, restrict, Scheme.Opens.fromSpecStalkOfMem]
  have e : ⟨x, hU' hx⟩ = (X.homOfLE hU').base ⟨x, hx⟩ := by
    rw [Scheme.homOfLE_base]
    rfl
  rw [Category.assoc, ← Spec_map_stalkMap_fromSpecStalk_assoc,
    ← Spec_map_stalkSpecializes_fromSpecStalk (Inseparable.of_eq e).specializes,
    ← TopCat.Presheaf.stalkCongr_inv _ (Inseparable.of_eq e)]
  simp only [← Category.assoc, ← Spec.map_comp]
  congr 3
  rw [Iso.eq_inv_comp, ← Category.assoc, IsIso.comp_inv_eq, IsIso.eq_inv_comp,
    stalkMap_congr_hom _ _ (X.homOfLE_ι hU').symm]
  simp only [TopCat.Presheaf.stalkCongr_hom]
  rw [← stalkSpecializes_stalkMap_assoc, stalkMap_comp]

lemma fromFunctionField_restrict (f : X.PartialMap Y) [IrreducibleSpace X]
    {U : X.Opens} (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) :
    (f.restrict U hU hU').fromFunctionField = f.fromFunctionField :=
  fromSpecStalkOfMem_restrict f _ _ _

/--
Given `S`-schemes `X` and `Y` such that `Y` is locally of finite type and
`X` is irreducible germ-injective at `x` (e.g. when `X` is integral),
any `S`-morphism `Spec         ⟶ Y` spreads out to a partial map from `X` to `Y`.
-/
noncomputable
def ofFromSpecStalk [IrreducibleSpace X] [LocallyOfFiniteType sY] {x : X} [X.IsGermInjectiveAt x]
    (φ : Spec (X.presheaf.stalk x) ⟶ Y) (h : φ      sY = X.fromSpecStalk x      sX) : X.PartialMap Y where
  hom := (spread_out_of_isGermInjective' sX sY φ h).choose_spec.choose_spec.choose
  domain := (spread_out_of_isGermInjective' sX sY φ h).choose
  dense_domain := (spread_out_of_isGermInjective' sX sY φ h).choose.2.dense
    ⟨_, (spread_out_of_isGermInjective' sX sY φ h).choose_spec.choose⟩

lemma ofFromSpecStalk_comp [IrreducibleSpace X] [LocallyOfFiniteType sY]
    {x : X} [X.IsGermInjectiveAt x] (φ : Spec (X.presheaf.stalk x) ⟶ Y)
    (h : φ      sY = X.fromSpecStalk x      sX) :
    (ofFromSpecStalk sX sY φ h).hom      sY = (ofFromSpecStalk sX sY φ h).domain.ι      sX :=
  (spread_out_of_isGermInjective' sX sY φ h).choose_spec.choose_spec.choose_spec.2

lemma mem_domain_ofFromSpecStalk [IrreducibleSpace X] [LocallyOfFiniteType sY]
    {x : X} [X.IsGermInjectiveAt x] (φ : Spec (X.presheaf.stalk x) ⟶ Y)
    (h : φ      sY = X.fromSpecStalk x      sX) : x ∈ (ofFromSpecStalk sX sY φ h).domain :=
  (spread_out_of_isGermInjective' sX sY φ h).choose_spec.choose

lemma fromSpecStalkOfMem_ofFromSpecStalk [IrreducibleSpace X] [LocallyOfFiniteType sY]
    {x : X} [X.IsGermInjectiveAt x] (φ : Spec (X.presheaf.stalk x) ⟶ Y)
    (h : φ      sY = X.fromSpecStalk x      sX) :
    (ofFromSpecStalk sX sY φ h).fromSpecStalkOfMem (mem_domain_ofFromSpecStalk sX sY φ h) = φ :=
  (spread_out_of_isGermInjective' sX sY φ h).choose_spec.choose_spec.choose_spec.1.symm

@[simp]
lemma fromSpecStalkOfMem_compHom (f : X.PartialMap Y) (g : Y ⟶ Z) (x) (hx) :
```

```lean
    (f.compHom g).fromSpecStalkOfMem (x := x) hx = f.fromSpecStalkOfMem hx     g := by
  simp [fromSpecStalkOfMem]

@[simp]
lemma fromSpecStalkOfMem_toPartialMap (f : X ⟶ Y) (x) :
    f.toPartialMap.fromSpecStalkOfMem (x := x) trivial = X.fromSpecStalk x     f := by
  simp [fromSpecStalkOfMem]

/-- Two partial maps are equivalent if they are equal on a dense open subscheme. -/
protected noncomputable
def equiv (f g : X.PartialMap Y) : Prop :=
  ∃ (W : X.Opens) (hW : Dense (W : Set X)) (hWl : W ≤ f.domain) (hWr : W ≤ g.domain),
    (f.restrict W hW hWl).hom = (g.restrict W hW hWr).hom

lemma equivalence_rel : Equivalence (@Scheme.PartialMap.equiv X Y) where
  refl f := ⟨f.domain, f.dense_domain, by simp⟩
  symm {f g} := by
    intro ⟨W, hW, hWl, hWr, e⟩
    exact ⟨W, hW, hWr, hWl, e.symm⟩
  trans {f g h} := by
    intro ⟨W₁, hW₁, hW₁l, hW₁r, e₁⟩ ⟨W₂, hW₂, hW₂l, hW₂r, e₂⟩
    refine ⟨W₁     W₂, hW₁.inter_of_isOpen_left hW₂ W₁.2, inf_le_left.trans hW₁l,
      inf_le_right.trans hW₂r, ?_⟩
    dsimp at e₁ e₂
    simp only [restrict_domain, restrict_hom, ← X.homOfLE_homOfLE (U := W₁     W₂) inf_le_left hW₁l,
      Category.assoc, e₁, ← X.homOfLE_homOfLE (U := W₁     W₂) inf_le_right hW₂r, ← e₂]
    simp only [homOfLE_homOfLE_assoc]

instance : Setoid (X.PartialMap Y) := ⟨@PartialMap.equiv X Y, equivalence_rel⟩

lemma restrict_equiv (f : X.PartialMap Y) (U : X.Opens)
    (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) : (f.restrict U hU hU').equiv f :=
  ⟨U, hU, le_rfl, hU', by simp⟩

lemma equiv_of_fromSpecStalkOfMem_eq [IrreducibleSpace X]
    {x : X} [X.IsGermInjectiveAt x] (f g : X.PartialMap Y)
    (hxf : x ∈ f.domain) (hxg : x ∈ g.domain)
    (H : f.fromSpecStalkOfMem hxf = g.fromSpecStalkOfMem hxg) : f.equiv g := by
  have hdense : Dense ((f.domain     g.domain) : Set X) :=
    f.dense_domain.inter_of_isOpen_left g.dense_domain f.domain.2
  have := (isGermInjectiveAt_iff_of_isOpenImmersion (f := (f.domain     g.domain).ι)
    (x := ⟨x, hxf, hxg⟩)).mp    _
  have := spread_out_unique_of_isGermInjective' (X := (f.domain     g.domain).toScheme)
    (X.homOfLE inf_le_left     f.hom) (X.homOfLE inf_le_right     g.hom) (x := ⟨x, hxf, hxg⟩) ?_
  · obtain ⟨U, hxU, e⟩ := this
    refine ⟨(f.domain     g.domain).ι ''    U, ((f.domain     g.domain).ι ''    U).2.dense
      ⟨_, ⟨_, hxU, rfl⟩⟩,
      ((Set.image_subset_range _ _).trans_eq (Subtype.range_val)).trans
```

```
        inf_le_left,
      ((Set.image_subset_range _ _).trans_eq (Subtype.range_val)).trans inf_le_right, ?_⟩
    rw [← cancel_epi (Scheme.Hom.isoImage _ _).hom]
    simp only [restrict_hom, ← Category.assoc] at e
    convert e using 2 <;> rw [← cancel_mono (Scheme.Opens.ι _)] <;> simp
  · rw [← f.fromSpecStalkOfMem_restrict hdense inf_le_left ⟨hxf, hxg⟩,
      ← g.fromSpecStalkOfMem_restrict hdense inf_le_right ⟨hxf, hxg⟩] at H
    simpa only [fromSpecStalkOfMem, restrict_domain, Opens.fromSpecStalkOfMem, Spec.map_inv,
      restrict_hom, Category.assoc, IsIso.eq_inv_comp, IsIso.hom_inv_id_assoc] using H

instance (U : X.Opens) [IsReduced X] : IsReduced U := isReduced_of_isOpenImmersion U.ι

lemma Opens.isDominant_ι {U : X.Opens} (hU : Dense (X := X) U) : IsDominant U.ι :=
  ⟨by simpa [DenseRange] using hU⟩

lemma Opens.isDominant_homOfLE {U V : X.Opens} (hU : Dense (X := X) U) (hU' : U ≤ V) :
    IsDominant (X.homOfLE hU') :=
  have : IsDominant (X.homOfLE hU'     Opens.ι _) := by simpa using Opens.isDominant_ι hU
  IsDominant.of_comp_of_isOpenImmersion (g := Opens.ι _) _

/-- Two partial maps from reduced schemes to separated schemes are equivalent if and only if
they are equal on **any** open dense subset. -/
lemma equiv_iff_of_isSeparated_of_le [X.Over S] [Y.Over S] [IsReduced X]
    [IsSeparated (Y     S)] {f g : X.PartialMap Y} [f.IsOver S] [g.IsOver S]
    {W : X.Opens} (hW : Dense (X := X) W) (hWl : W ≤ f.domain) (hWr : W ≤ g.domain) : f.equiv g ↔
      (f.restrict W hW hWl).hom = (g.restrict W hW hWr).hom := by
  refine ⟨fun ⟨V, hV, hVl, hVr, e⟩     ?_, fun e     ⟨_, _, _, _, e⟩⟩
  have : IsDominant (X.homOfLE (inf_le_left : W     V ≤ W)) :=
    Opens.isDominant_homOfLE (hW.inter_of_isOpen_left hV W.2) _
  apply ext_of_isDominant_of_isSeparated' S (X.homOfLE (inf_le_left : W     V ≤ W))
  simpa using congr(X.homOfLE (inf_le_right : W     V ≤ V)     $e)

/-- Two partial maps from reduced schemes to separated schemes are equivalent if and only if
they are equal on the intersection of the domains. -/
lemma equiv_iff_of_isSeparated [X.Over S] [Y.Over S] [IsReduced X]
    [IsSeparated (Y     S)] {f g : X.PartialMap Y}
    [f.IsOver S] [g.IsOver S] : f.equiv g ↔
      (f.restrict _ (f.2.inter_of_isOpen_left g.2 f.domain.2) inf_le_left).hom =
      (g.restrict _ (f.2.inter_of_isOpen_left g.2 f.domain.2) inf_le_right).hom :=
  equiv_iff_of_isSeparated_of_le (S := S) _ _ _

/-- Two partial maps from reduced schemes to separated schemes with the same domain are equivalent
if and only if they are equal. -/
lemma equiv_iff_of_domain_eq_of_isSeparated [X.Over S] [Y.Over S] [IsReduced X]
    [IsSeparated (Y     S)] {f g : X.PartialMap Y} (hfg : f.domain = g.domain)
    [f.IsOver S] [g.IsOver S] : f.equiv g ↔ f = g := by
  rw [equiv_iff_of_isSeparated_of_le (S := S) f.dense_domain le_rfl hfg.le]
  obtain ⟨Uf, _, f⟩ := f
```

```
  obtain ⟨Ug, _, g⟩ := g
  obtain rfl : Uf = Ug := hfg
  simp

/-- A partial map from a reduced scheme to a separated scheme is equivalent to a morphism
if and only if it is equal to the restriction of the morphism. -/
lemma equiv_toPartialMap_iff_of_isSeparated [X.Over S] [Y.Over S] [IsReduced X]
    [IsSeparated (Y     S)] {f : X.PartialMap Y} {g : X ⟶ Y}
    [f.IsOver S] [g.IsOver S] : f.equiv g.toPartialMap ↔
      f.hom = f.domain.ι     g := by
  rw [equiv_iff_of_isSeparated (S := S), ← cancel_epi (X.isoOfEq (inf_top_eq f.domain)).hom]
  simp
  rfl

end PartialMap

/-- A rational map from `X` to `Y` (`X     Y`) is an equivalence class of partial maps,
where two partial maps are equivalent if they are equal on a dense open subscheme. -/
def RationalMap (X Y : Scheme.{u}) : Type u :=
  @Quotient (X.PartialMap Y) inferInstance

/-- The notation for rational maps. -/
scoped[AlgebraicGeometry] infix:10 "     " => Scheme.RationalMap

/-- A partial map as a rational map. -/
def PartialMap.toRationalMap (f : X.PartialMap Y) : X     Y := Quotient.mk _ f

/-- A scheme morphism as a rational map. -/
abbrev Hom.toRationalMap (f : X.Hom Y) : X     Y := f.toPartialMap.toRationalMap

variable (S) in
/-- A rational map is a `S`-map if some partial map in the equivalence class is a `S`-map. -/
class RationalMap.IsOver [X.Over S] [Y.Over S] (f : X     Y) : Prop where
  exists_partialMap_over : ∃ g : X.PartialMap Y, g.IsOver S ∧ g.toRationalMap = f

lemma PartialMap.toRationalMap_surjective : Function.Surjective (@toRationalMap X Y) :=
  Quotient.exists_rep

lemma RationalMap.exists_rep (f : X     Y) : ∃ g : X.PartialMap Y, g.toRationalMap = f :=
  Quotient.exists_rep f

lemma PartialMap.toRationalMap_eq_iff {f g : X.PartialMap Y} :
    f.toRationalMap = g.toRationalMap ↔ f.equiv g :=
  Quotient.eq

@[simp]
lemma PartialMap.restrict_toRationalMap (f : X.PartialMap Y) (U : X.Opens)
    (hU : Dense (U : Set X)) (hU' : U ≤ f.domain) :
    (f.restrict U hU hU').toRationalMap = f.toRationalMap :=
  toRationalMap_eq_iff.mpr (f.restrict_equiv U hU hU')

instance [X.Over S] [Y.Over S] (f : X.PartialMap Y) [f.IsOver S] :
```

```
    f.toRationalMap.IsOver S :=
  ⟨f,    _    , rfl⟩

variable (S) in
lemma RationalMap.exists_partialMap_over [X.Over S] [Y.Over S] (f : X      Y)
    [f.IsOver S] :
    ∃ g : X.PartialMap Y, g.IsOver S ∧ g.toRationalMap = f :=
  IsOver.exists_partialMap_over

/-- The composition of a rational map and a morphism on the right. -/
def RationalMap.compHom (f : X      Y) (g : Y ⟶ Z) : X      Z := by
  refine Quotient.map (PartialMap.compHom · g) ?_ f
  intro f₁ f₂ ⟨W, hW, hWl, hWr, e⟩
  refine ⟨W, hW, hWl, hWr, ?_⟩
  simp only [PartialMap.restrict_domain, PartialMap.restrict_hom,
      PartialMap.compHom_domain,
    PartialMap.compHom_hom] at e
  rw [reassoc_of% e]

@[simp]
lemma RationalMap.compHom_toRationalMap (f : X.PartialMap Y) (g : Y ⟶ Z) :
    (f.compHom g).toRationalMap = f.toRationalMap.compHom g := rfl

instance [X.Over S] [Y.Over S] [Z.Over S] (f : X      Y) (g : Y ⟶ Z)
    [f.IsOver S] [g.IsOver S] : (f.compHom g).IsOver S where
  exists_partialMap_over := by
    obtain ⟨f, hf, rfl⟩ := f.exists_partialMap_over S
    exact ⟨f.compHom g, inferInstance, rfl⟩

variable (S) in
lemma PartialMap.exists_restrict_isOver [X.Over S] [Y.Over S] (f : X.PartialMap Y)
    [f.toRationalMap.IsOver S] : ∃ U hU hU', (f.restrict U hU hU').IsOver S := by
  obtain ⟨f', hf₁, hf₂⟩ := RationalMap.IsOver.exists_partialMap_over (S := S) (f
      := f.toRationalMap)
  obtain ⟨U, hU, hUl, hUr, e⟩ := PartialMap.toRationalMap_eq_iff.mp hf₂
  exact ⟨U, hU, hUr, by rw [IsOver, ← e]; infer_instance⟩

lemma RationalMap.isOver_iff [X.Over S] [Y.Over S] {f : X      Y} :
    f.IsOver S ↔ f.compHom (Y      S) = (X      S).toRationalMap := by
  constructor
  · intro h
    obtain ⟨g, hg, e⟩ := f.exists_partialMap_over S
    rw [← e, Hom.toRationalMap, ← compHom_toRationalMap,
        PartialMap.isOver_iff_eq_restrict.mp hg,
      PartialMap.restrict_toRationalMap]
  · intro e
    obtain ⟨f, rfl⟩ := PartialMap.toRationalMap_surjective f
    obtain ⟨U, hU, hUl, hUr, e⟩ := PartialMap.toRationalMap_eq_iff.mp e
    exact ⟨⟨f.restrict U hU hUl, by simpa using e, by simp⟩⟩

lemma PartialMap.isOver_toRationalMap_iff_of_isSeparated [X.Over S] [Y.Over S]
    [IsReduced X]
    [S.IsSeparated] {f : X.PartialMap Y} :
    f.toRationalMap.IsOver S ↔ f.IsOver S := by
  refine ⟨fun _      ?_, fun _      inferInstance⟩
  obtain ⟨U, hU, hU', H⟩ := f.exists_restrict_isOver (S := S)
  rw [isOver_iff]
  have : IsDominant (X.homOfLE hU') := Opens.isDominant_homOfLE hU _
  exact ext_of_isDominant (ι := X.homOfLE hU') (by simpa using H.1)
```

```
section functionField

/-- A rational map restricts to a map from `Spec K(X)`. -/
noncomputable
def RationalMap.fromFunctionField [IrreducibleSpace X] (f : X      Y) :
    Spec X.functionField ⟶ Y := by
  refine Quotient.lift PartialMap.fromFunctionField ?_ f
  intro f g ⟨W, hW, hWl, hWr, e⟩
  have : f.restrict W hW hWl = g.restrict W hW hWr := by ext1; rfl; rw [e]; simp
  rw [← f.fromFunctionField_restrict hW hWl, this, g.fromFunctionField_restrict]

@[simp]
lemma RationalMap.fromFunctionField_toRationalMap [IrreducibleSpace X] (f : X.PartialMap Y) :
    f.toRationalMap.fromFunctionField = f.fromFunctionField := rfl

/--
Given `S`-schemes `X` and `Y` such that `Y` is locally of finite type and `X` is integral,
any `S`-morphism `Spec K(X) ⟶ Y` spreads out to a rational map from `X` to `Y`.
-/
noncomputable
def RationalMap.ofFunctionField [IsIntegral X] [LocallyOfFiniteType sY]
    (f : Spec X.functionField ⟶ Y) (h : f      sY = X.fromSpecStalk _      sX) : X      Y :=
  (PartialMap.ofFromSpecStalk sX sY f h).toRationalMap

lemma RationalMap.fromFunctionField_ofFunctionField [IsIntegral X] [LocallyOfFiniteType sY]
    (f : Spec X.functionField ⟶ Y) (h : f      sY = X.fromSpecStalk _      sX) :
    (ofFunctionField sX sY f h).fromFunctionField = f :=
  PartialMap.fromSpecStalkOfMem_ofFromSpecStalk sX sY _ _

lemma RationalMap.eq_of_fromFunctionField_eq [IsIntegral X] (f g : X.RationalMap Y)
    (H : f.fromFunctionField = g.fromFunctionField) : f = g := by
  obtain ⟨f, rfl⟩ := f.exists_rep
  obtain ⟨g, rfl⟩ := g.exists_rep
  refine PartialMap.toRationalMap_eq_iff.mpr ?_
  exact PartialMap.equiv_of_fromSpecStalkOfMem_eq _ _ _ _ H

/--
Given `S`-schemes `X` and `Y` such that `Y` is locally of finite type and `X` is integral,
`S`-morphisms `Spec K(X) ⟶ Y` correspond bijectively to `S`-rational maps from `X` to `Y`.
-/
noncomputable
def RationalMap.equivFunctionField [IsIntegral X] [LocallyOfFiniteType sY] :
    { f : Spec X.functionField ⟶ Y // f      sY = X.fromSpecStalk _      sX } ≃
      { f : X      Y // f.compHom sY = sX.toRationalMap } where
  toFun f := ⟨.ofFunctionField sX sY f f.2, PartialMap.toRationalMap_eq_iff.mpr
      ⟨_, PartialMap.dense_domain _, le_rfl, le_top, by simp [PartialMap.ofFromSpecStalk_comp]⟩⟩
  invFun f := ⟨f.1.fromFunctionField, by
    obtain ⟨f, hf⟩ := f
    obtain ⟨f, rfl⟩ := f.exists_rep
    simpa [fromFunctionField_toRationalMap] using
```

```
      congr(RationalMap.fromFunctionField $hf)⟩
  left_inv f := Subtype.ext (RationalMap.fromFunctionField_ofFunctionField _ _ _ _)
  right_inv f := Subtype.ext (RationalMap.eq_of_fromFunctionField_eq
      (ofFunctionField sX sY f.1.fromFunctionField _) f
      (RationalMap.fromFunctionField_ofFunctionField _ _ _ _))

/--
Given `S`-schemes `X` and `Y` such that `Y` is locally of finite type and `X` is integral,
`S`-morphisms `Spec K(X) ⟶ Y` correspond bijectively to `S`-rational maps from `X` to `Y`.
-/
noncomputable
def RationalMap.equivFunctionFieldOver [X.Over S] [Y.Over S] [IsIntegral X]
    [LocallyOfFiniteType (Y    S)] :
    { f : Spec X.functionField ⟶ Y // f.IsOver S } ≃ { f : X     Y // f.IsOver S } :=
  ((Equiv.subtypeEquivProp (by simp only [Hom.isOver_iff]; rfl)).trans
    (RationalMap.equivFunctionField (X     S) (Y     S))).trans
      (Equiv.subtypeEquivProp (by ext f; rw [RationalMap.isOver_iff]))

end functionField

section domain

/-- The domain of definition of a rational map. -/
def RationalMap.domain (f : X     Y) : X.Opens :=
  sSup { PartialMap.domain g | (g) (_ : g.toRationalMap = f) }

lemma PartialMap.le_domain_toRationalMap (f : X.PartialMap Y) :
    f.domain ≤ f.toRationalMap.domain :=
  le_sSup ⟨f, rfl, rfl⟩

lemma RationalMap.mem_domain {f : X     Y} {x} :
    x ∈ f.domain ↔ ∃ g : X.PartialMap Y, x ∈ g.domain ∧ g.toRationalMap = f :=
  TopologicalSpace.Opens.mem_sSup.trans (by simp [@and_comm (x ∈ _)])

lemma RationalMap.dense_domain (f : X     Y) : Dense (X := X) f.domain :=
  f.inductionOn (fun g     g.dense_domain.mono g.le_domain_toRationalMap)

/-- The open cover of the domain of `f : X     Y`,
consisting of all the domains of the partial maps in the equivalence class. -/
noncomputable
def RationalMap.openCoverDomain (f : X     Y) : f.domain.toScheme.OpenCover where
  J := { PartialMap.domain g | (g) (_ : g.toRationalMap = f) }
  obj U := U.1.toScheme
  map U := X.homOfLE (le_sSup U.2)
  f x := ⟨_, (TopologicalSpace.Opens.mem_sSup.mp x.2).choose_spec.1⟩
  covers x := ⟨⟨x.1, (TopologicalSpace.Opens.mem_sSup.mp x.2).choose_spec.2⟩, Subtype.ext (by simp)⟩

/-- If `f : X     Y` is a rational map from a reduced scheme to a separated scheme,
then `f` can be represented as a partial map on its domain of definition. -/
noncomputable
def RationalMap.toPartialMap [IsReduced X] [Y.IsSeparated] (f : X     Y) : X.PartialMap Y := by
  refine ⟨f.domain, f.dense_domain, f.openCoverDomain.glueMorphisms
```

```
    (fun x    (X.isoOfEq x.2.choose_spec.2).inv    x.2.choose.hom) ?_⟩
  intro x y
  let g (x : f.openCoverDomain.J) := x.2.choose
  have hg₁ (x) : (g x).toRationalMap = f := x.2.choose_spec.1
  have hg₂ (x) : (g x).domain = x.1 := x.2.choose_spec.2
  refine (cancel_epi (isPullback_opens_inf_le (le_sSup x.2) (le_sSup y.2)).isoPullback.hom).mp ?_
  simp only [openCoverDomain, IsPullback.isoPullback_hom_fst_assoc,
    IsPullback.isoPullback_hom_snd_assoc]
  change _    _    (g x).hom = _    _    (g y).hom
  simp_rw [← cancel_epi (X.isoOfEq congr($(hg₂ x)    $(hg₂ y))).hom, ← Category.assoc]
  convert (PartialMap.equiv_iff_of_isSeparated (S := ⊤_ _) (f := g x) (g := g y)).mp ?_ using 1
  · dsimp; congr 1; simp [g, ← cancel_mono (Opens.ι _)]
  · dsimp; congr 1; simp [g, ← cancel_mono (Opens.ι _)]
  · rw [← PartialMap.toRationalMap_eq_iff, hg₁, hg₁]

lemma PartialMap.toPartialMap_toRationalMap_restrict [IsReduced X] [Y.IsSeparated]
    (f : X.PartialMap Y) : (f.toRationalMap.toPartialMap.restrict _ f.dense_domain
      f.le_domain_toRationalMap).hom = f.hom := by
  dsimp [RationalMap.toPartialMap]
  refine (f.toRationalMap.openCoverDomain.ι_glueMorphisms _ _ ⟨_, f, rfl, rfl⟩).trans ?_
  generalize_proofs _ _ H _
  have : H.choose = f := (equiv_iff_of_domain_eq_of_isSeparated (S := ⊤_ _) H.choose_spec.2).mp
    (toRationalMap_eq_iff.mp H.choose_spec.1)
  exact ((ext_iff _ _).mp this.symm).choose_spec.symm

@[simp]
lemma RationalMap.toRationalMap_toPartialMap [IsReduced X] [Y.IsSeparated]
    (f : X    Y) : f.toPartialMap.toRationalMap = f := by
  obtain ⟨f, rfl⟩ := PartialMap.toRationalMap_surjective f
  trans (f.toRationalMap.toPartialMap.restrict _
    f.dense_domain f.le_domain_toRationalMap).toRationalMap
  · simp
  · congr 1
    exact PartialMap.ext _ f rfl (by simpa using f.toPartialMap_toRationalMap_restrict)

instance [IsReduced X] [Y.IsSeparated] [S.IsSeparated] [X.Over S] [Y.Over S]
    (f : X    Y) [f.IsOver S] : f.toPartialMap.IsOver S := by
  rw [← PartialMap.isOver_toRationalMap_iff_of_isSeparated, f.toRationalMap_toPartialMap]
  infer_instance

end domain

end Scheme

end AlgebraicGeometry
```

Listing 1: RationalMap.lean